p-adics

Release 10.2

The Sage Development Team

Dec 06, 2023

CONTENTS

CHAPTER

ONE

INTRODUCTION TO THE P-ADICS

This tutorial outlines what you need to know in order to use p-adics in Sage effectively.

Our goal is to create a rich structure of different options that will reflect the mathematical structures of the p-adics. This is very much a work in progress: some of the classes that we eventually intend to include have not yet been written, and some of the functionality for classes in existence has not yet been implemented. In addition, while we strive for perfect code, bugs (both subtle and not-so-subtle) continue to evade our clutches. As a user, you serve an important role. By writing non-trivial code that uses the p-adics, you both give us insight into what features are actually used and also expose problems in the code for us to fix.

Our design philosophy has been to create a robust, usable interface working first, with simple-minded implementations underneath. We want this interface to stabilize rapidly, so that users' code does not have to change. Once we get the framework in place, we can go back and work on the algorithms and implementations underneath. All of the current p-adic code is currently written in pure Python, which means that it does not have the speed advantage of compiled code. Thus our p-adics can be painfully slow at times when you're doing real computations. However, finding and fixing bugs in Python code is *far* easier than finding and fixing errors in the compiled alternative within Sage (Cython), and Python code is also faster and easier to write. We thus have significantly more functionality implemented and working than we would have if we had chosen to focus initially on speed. And at some point in the future, we will go back and improve the speed. Any code you have written on top of our p-adics will then get an immediate performance enhancement.

If you do find bugs, have feature requests or general comments, please email sage-support@groups.google.com or roed@math.harvard.edu.

1.1 Terminology and types of p-adics

To write down a general p-adic element completely would require an infinite amount of data. Since computers do not have infinite storage space, we must instead store finite approximations to elements. Thus, just as in the case of floating point numbers for representing reals, we have to store an element to a finite precision level. The different ways of doing this account for the different types of p-adics.

We can think of p-adics in two ways. First, as a projective limit of finite groups:

$$\mathbf{Z}_p = \lim_{\leftarrow n} \mathbf{Z}/p^n\mathbf{Z}.$$

Secondly, as Cauchy sequences of rationals (or integers, in the case of $\mathbf{Z}_p$) under the p-adic metric. Since we only need to consider these sequences up to equivalence, this second way of thinking of the p-adics is the same as considering power series in p with integral coefficients in the range 0 to $p-1$. If we only allow nonnegative powers of p then these power series converge to elements of $\mathbf{Z}_p$, and if we allow bounded negative powers of p then we get $\mathbf{Q}_p$.

Both of these representations give a natural way of thinking about finite approximations to a p-adic element. In the first representation, we can just stop at some point in the projective limit, giving an element of $\mathbf{Z}/p^n\mathbf{Z}$. As $\mathbf{Z}_p/p^n\mathbf{Z}_p \cong \mathbf{Z}/p^n\mathbf{Z}$, this is equivalent to specifying our element modulo $p^n\mathbf{Z}_p$.

The *absolute precision* of a finite approximation $\bar{x} \in \mathbf{Z}/p^n\mathbf{Z}$ to $x \in \mathbf{Z}_p$ is the non-negative integer n.

In the second representation, we can achieve the same thing by truncating a series

$$a_0 + a_1 p + a_2 p^2 + \cdots$$

at p^n, yielding

$$a_0 + a_1 p + \cdots + a_{n-1} p^{n-1} + O(p^n).$$

As above, we call this n the absolute precision of our element.

Given any $x \in \mathbf{Q}_p$ with $x \neq 0$, we can write $x = p^v u$ where $v \in \mathbf{Z}$ and $u \in \mathbf{Z}_p^\times$. We could thus also store an element of $\mathbf{Q}_p$ (or $\mathbf{Z}_p$) by storing v and a finite approximation of u. This motivates the following definition: the *relative precision* of an approximation to x is defined as the absolute precision of the approximation minus the valuation of x. For example, if $x = a_k p^k + a_{k+1} p^{k+1} + \cdots + a_{n-1} p^{n-1} + O(p^n)$ then the absolute precision of x is n, the valuation of x is k and the relative precision of x is $n - k$.

There are three different representations of $\mathbf{Z}_p$ in Sage and one representation of $\mathbf{Q}_p$:

- the fixed modulus ring
- the capped absolute precision ring
- the capped relative precision ring, and
- the capped relative precision field.

1.1.1 Fixed Modulus Rings

The first, and simplest, type of $\mathbf{Z}_p$ is basically a wrapper around $\mathbf{Z}/p^n\mathbf{Z}$, providing a unified interface with the rest of the p-adics. You specify a precision, and all elements are stored to that absolute precision. If you perform an operation that would normally lose precision, the element does not track that it no longer has full precision.

The fixed modulus ring provides the lowest level of convenience, but it is also the one that has the lowest computational overhead. Once we have ironed out some bugs, the fixed modulus elements will be those most optimized for speed.

As with all of the implementations of $\mathbf{Z}_p$, one creates a new ring using the constructor `Zp`, and passing in `'fixed-mod'` for the `type` parameter. For example,

```
sage: R = Zp(5, prec = 10, type = 'fixed-mod', print_mode = 'series')
sage: R
5-adic Ring of fixed modulus 5^10
```

One can create elements as follows:

```
sage: a = R(375)
sage: a
3*5^3
sage: b = R(105)
sage: b
5 + 4*5^2
```

Now that we have some elements, we can do arithmetic in the ring.

```
sage: a + b
5 + 4*5^2 + 3*5^3
sage: a * b
3*5^4 + 2*5^5 + 2*5^6
```

Floor division (//) divides even though the result isn't really known to the claimed precision; note that division isn't defined:

```
sage: a // 5
3*5^2
```

```
sage: a / 5
Traceback (most recent call last):
...
ValueError: cannot invert non-unit
```

Since elements don't actually store their actual precision, one can only divide by units:

```
sage: a / 2
4*5^3 + 2*5^4 + 2*5^5 + 2*5^6 + 2*5^7 + 2*5^8 + 2*5^9
sage: a / b
Traceback (most recent call last):
...
ValueError: cannot invert non-unit
```

If you want to divide by a non-unit, do it using the `//` operator:

```
sage: a // b
3*5^2 + 3*5^3 + 2*5^5 + 5^6 + 4*5^7 + 2*5^8 + 3*5^9
```

1.1.2 Capped Absolute Rings

The second type of implementation of $\mathbf{Z}_p$ is similar to the fixed modulus implementation, except that individual elements track their known precision. The absolute precision of each element is limited to be less than the precision cap of the ring, even if mathematically the precision of the element would be known to greater precision (see Appendix A for the reasons for the existence of a precision cap).

Once again, use `Zp` to create a capped absolute p-adic ring.

```
sage: R = Zp(5, prec = 10, type = 'capped-abs', print_mode = 'series')
sage: R
5-adic Ring with capped absolute precision 10
```

We can do similar things as in the fixed modulus case:

```
sage: a = R(375)
sage: a
3*5^3 + O(5^10)
sage: b = R(105)
sage: b
5 + 4*5^2 + O(5^10)
sage: a + b
```

(continues on next page)

(continued from previous page)

```
5 + 4*5^2 + 3*5^3 + O(5^10)
sage: a * b
3*5^4 + 2*5^5 + 2*5^6 + O(5^10)
sage: c = a // 5
sage: c
3*5^2 + O(5^9)
```

Note that when we divided by 5, the precision of c dropped. This lower precision is now reflected in arithmetic.

```
sage: c + b
5 + 2*5^2 + 5^3 + O(5^9)
```

Division is allowed: the element that results is a capped relative field element, which is discussed in the next section:

```
sage: 1 / (c + b)
5^-1 + 3 + 2*5 + 5^2 + 4*5^3 + 4*5^4 + 3*5^6 + O(5^7)
```

1.1.3 Capped Relative Rings and Fields

Instead of restricting the absolute precision of elements (which doesn't make much sense when elements have negative valuations), one can cap the relative precision of elements. This is analogous to floating point representations of real numbers. As in the reals, multiplication works very well: the valuations add and the relative precision of the product is the minimum of the relative precisions of the inputs. Addition, however, faces similar issues as floating point addition: relative precision is lost when lower order terms cancel.

To create a capped relative precision ring, use Zp as before. To create capped relative precision fields, use Qp.

```
sage: R = Zp(5, prec = 10, type = 'capped-rel', print_mode = 'series')
sage: R
5-adic Ring with capped relative precision 10
sage: K = Qp(5, prec = 10, type = 'capped-rel', print_mode = 'series')
sage: K
5-adic Field with capped relative precision 10
```

We can do all of the same operations as in the other two cases, but precision works a bit differently: the maximum precision of an element is limited by the precision cap of the ring.

```
sage: a = R(375)
sage: a
3*5^3 + O(5^13)
sage: b = K(105)
sage: b
5 + 4*5^2 + O(5^11)
sage: a + b
5 + 4*5^2 + 3*5^3 + O(5^11)
sage: a * b
3*5^4 + 2*5^5 + 2*5^6 + O(5^14)
sage: c = a // 5
sage: c
3*5^2 + O(5^12)
sage: c + 1
1 + 3*5^2 + O(5^10)
```

As with the capped absolute precision rings, we can divide, yielding a capped relative precision field element.

```
sage: 1 / (c + b)
5^-1 + 3 + 2*5 + 5^2 + 4*5^3 + 4*5^4 + 3*5^6 + 2*5^7 + 5^8 + O(5^9)
```

1.1.4 Unramified Extensions

One can create unramified extensions of $\mathbf{Z}_p$ and $\mathbf{Q}_p$ using the functions `Zq` and `Qq`.

In addition to requiring a prime power as the first argument, `Zq` also requires a name for the generator of the residue field. One can specify this name as follows:

```
sage: R.<c> = Zq(125, prec=20); R                                          #␣
↪needs sage.libs.ntl
5-adic Unramified Extension Ring in c defined by x^3 + 3*x + 3
```

1.1.5 Eisenstein Extensions

It is also possible to create Eisenstein extensions of $\mathbf{Z}_p$ and $\mathbf{Q}_p$. In order to do so, create the ground field first:

```
sage: R = Zp(5, 2)
```

Then define the polynomial yielding the desired extension.:

```
sage: S.<x> = ZZ[]
sage: f = x^5 - 25*x^3 + 15*x - 5
```

Finally, use the `ext` function on the ground field to create the desired extension.:

```
sage: W.<w> = R.ext(f)                                                     #␣
↪needs sage.libs.ntl sage.rings.padics
```

You can do arithmetic in this Eisenstein extension:

```
sage: (1 + w)^7                                                            #␣
↪needs sage.libs.ntl sage.rings.padics
1 + 2*w + w^2 + w^5 + 3*w^6 + 3*w^7 + 3*w^8 + w^9 + O(w^10)
```

Note that the precision cap increased by a factor of 5, since the ramification index of this extension over $\mathbf{Z}_p$ is 5.

CHAPTER

TWO

FACTORY

This file contains the constructor classes and functions for p-adic rings and fields.

AUTHORS:

- David Roe

sage.rings.padics.factory.**QpCR**(*p*, *prec=None*, **args*, ***kwds*)

A shortcut function to create capped relative p-adic fields.

Same functionality as `Qp()`. See documentation for `Qp()` for a description of the input parameters.

EXAMPLES:

```
sage: QpCR(5, 40)
5-adic Field with capped relative precision 40
```

sage.rings.padics.factory.**QpER**(*p*, *prec=None*, *halt=None*, *secure=False*, **args*, ***kwds*)

A shortcut function to create relaxed p-adic fields.

See *ZpER()* for more information about this model of precision.

EXAMPLES:

```
sage: R = QpER(2); R                                                    #␣
↪needs sage.libs.flint
2-adic Field handled with relaxed arithmetics
```

sage.rings.padics.factory.**QpFP**(*p*, *prec=None*, **args*, ***kwds*)

A shortcut function to create floating point p-adic fields.

Same functionality as `Qp()`. See documentation for `Qp()` for a description of the input parameters.

EXAMPLES:

```
sage: QpFP(5, 40)
5-adic Field with floating precision 40
```

sage.rings.padics.factory.**QpLC**(*p*, *prec=None*, **args*, ***kwds*)

A shortcut function to create p-adic fields with lattice precision.

See *ZpLC()* for more information about this model of precision.

EXAMPLES:

```
sage: R = QpLC(2)
sage: R
2-adic Field with lattice-cap precision
```

`sage.rings.padics.factory.QpLF(p, prec=None, *args, **kwds)`

A shortcut function to create p-adic fields with lattice precision.

See `ZpLC()` for more information about this model of precision.

EXAMPLES:

```
sage: R = QpLF(2)
sage: R
2-adic Field with lattice-float precision
```

`class sage.rings.padics.factory.Qp_class`

Bases: `UniqueFactory`

A creation function for p-adic fields.

INPUT:

- `p` – integer: the p in $\mathbf{Q}_p$
- `prec` – integer (default: `20`) the precision cap of the field. In the lattice capped case, `prec` can either be a pair (`relative_cap`, `absolute_cap`) or an integer (understood at relative cap). In the relaxed case, `prec` can be either a pair (`default_prec`, `halting_prec`) or an integer (understood at default precision). Except in the floating point case, individual elements keep track of their own precision. See TYPES and PRECISION below.
- `type` – string (default: `'capped-rel'`) Valid types are `'capped-rel'`, `'floating-point'`, `'lattice-cap'`, `'lattice-float'`. See TYPES and PRECISION below
- `print_mode` – string (default: `None`). Valid modes are 'series', 'val-unit', 'terse', 'digits', and 'bars'. See PRINTING below
- `names` – string or tuple (defaults to a string representation of p). What to use whenever p is printed.
- `ram_name` – string. Another way to specify the name; for consistency with the `Qq` and `Zq` and extension functions.
- `print_pos` – bool (default `None`) Whether to only use positive integers in the representations of elements. See PRINTING below.
- `print_sep` – string (default `None`) The separator character used in the `'bars'` mode. See PRINTING below.
- `print_alphabet` – tuple (default `None`) The encoding into digits for use in the 'digits' mode. See PRINTING below.
- `print_max_terms` – integer (default `None`) The maximum number of terms shown. See PRINTING below.
- `show_prec` – a boolean or a string (default `None`) Specify how the precision is printed. See PRINTING below.
- `check` – bool (default `True`) whether to check if p is prime. Non-prime input may cause seg-faults (but can also be useful for base n expansions for example)
- `label` – string (default `None`) used for lattice precision to create parents with different lattices.

OUTPUT:

- The corresponding p-adic field.

TYPES AND PRECISION:

There are two main types of precision for a p-adic element. The first is relative precision, which gives the number of known p-adic digits:

```
sage: R = Qp(5, 20, 'capped-rel', 'series'); a = R(675); a
2*5^2 + 5^4 + O(5^22)
sage: a.precision_relative()
20
```

The second type of precision is absolute precision, which gives the power of p that this element is defined modulo:

```
sage: a.precision_absolute()
22
```

There are several types of p-adic fields, depending on the methods used for tracking precision. Namely, we have:

- capped relative fields (`type='capped-rel'`)
- capped absolute fields (`type='capped-abs'`)
- fixed modulus fields (`type='fixed-mod'`)
- floating point fields (`type='floating-point'`)
- lattice precision fields (`type='lattice-cap'` or `type='lattice-float'`)
- exact fields with relaxed arithmetics (`type='relaxed'`)

In the capped relative case, the relative precision of an element is restricted to be at most a certain value, specified at the creation of the field. Individual elements also store their own precision, so the effect of various arithmetic operations on precision is tracked. When you cast an exact element into a capped relative field, it truncates it to the precision cap of the field.

```
sage: R = Qp(5, 5, 'capped-rel', 'series'); a = R(4006); a
1 + 5 + 2*5^3 + 5^4 + O(5^5)
sage: b = R(4025); b
5^2 + 2*5^3 + 5^4 + 5^5 + O(5^7)
sage: a + b
1 + 5 + 5^2 + 4*5^3 + 2*5^4 + O(5^5)
```

In the floating point case, elements do not track their precision, but the relative precision of elements is truncated during arithmetic to the precision cap of the field.

In the lattice case, precision on elements is tracked by a global lattice that is updated after every operation, yielding better precision behavior at the cost of higher memory and runtime usage. We refer to the documentation of the function *ZpLC()* for a small demonstration of the capabilities of this precision model.

Finally, the model for relaxed p-adics is quite different from any of the other types. In addition to storing a finite approximation, one also stores a method for increasing the precision. A quite interesting feature with relaxed p-adics is the possibility to create (in some cases) self-referent numbers, that are numbers whose n-th digit is defined by the previous ones. We refer to the documentation of the function `ZpL()` for a small demonstration of the capabilities of this precision model.

PRINTING:

There are many different ways to print p-adic elements. The way elements of a given field print is controlled by options passed in at the creation of the field. There are five basic printing modes (series, val-unit, terse, digits and bars), as well as various options that either hide some information in the print representation or sometimes make print representations more compact. Note that the printing options affect whether different p-adic fields are considered equal.

1. **series**: elements are displayed as series in p.

```
sage: R = Qp(5, print_mode='series'); a = R(70700); a
3*5^2 + 3*5^4 + 2*5^5 + 4*5^6 + O(5^22)
sage: b = R(-70700); b
2*5^2 + 4*5^3 + 5^4 + 2*5^5 + 4*5^7 + 4*5^8 + 4*5^9 + 4*5^10 + 4*5^11
 + 4*5^12 + 4*5^13 + 4*5^14 + 4*5^15 + 4*5^16 + 4*5^17 + 4*5^18
 + 4*5^19 + 4*5^20 + 4*5^21 + O(5^22)
```

print_pos controls whether negatives can be used in the coefficients of powers of p.

```
sage: S = Qp(5, print_mode='series', print_pos=False); a = S(70700); a
-2*5^2 + 5^3 - 2*5^4 - 2*5^5 + 5^7 + O(5^22)
sage: b = S(-70700); b
2*5^2 - 5^3 + 2*5^4 + 2*5^5 - 5^7 + O(5^22)
```

print_max_terms limits the number of terms that appear.

```
sage: T = Qp(5, print_mode='series', print_max_terms=4); b = R(-70700);␣
↪repr(b)
'2*5^2 + 4*5^3 + 5^4 + 2*5^5 + ... + O(5^22)'
```

names affects how the prime is printed.

```
sage: U.<p> = Qp(5); p
p + O(p^21)
```

show_prec determines how the precision is printed. It can be either 'none' (or equivalently `False`), 'bigoh' (or equivalently `True`) or 'bigoh'. The default is `False` for the `'floating-point'` type and `True` for all other types.

```
sage: Qp(5)(6)
1 + 5 + O(5^20)
sage: Qp(5, show_prec='none')(6)
1 + 5

sage: QpFP(5)(6)
1 + 5
```

print_sep and *print_alphabet* have no effect in series mode.

Note that print options affect equality:

```
sage: R == S, R == T, R == U, S == T, S == U, T == U
(False, False, False, False, False, False)
```

2. **val-unit**: elements are displayed as p^k*u:

```
sage: R = Qp(5, print_mode='val-unit'); a = R(70700); a
5^2 * 2828 + O(5^22)
sage: b = R(-707/5); b
5^-1 * 95367431639918 + O(5^19)
```

print_pos controls whether to use a balanced representation or not.

```
sage: S = Qp(5, print_mode='val-unit', print_pos=False); b = S(-70700); b
5^2 * (-2828) + O(5^22)
```

names affects how the prime is printed.

```
sage: T = Qp(5, print_mode='val-unit', names='pi'); a = T(70700); a
pi^2 * 2828 + O(pi^22)
```

show_prec determines how the precision is printed. It can be either 'none' (or equivalently `False`) or 'bigoh' (or equivalently `True`). The default is `False` for the `'floating-point'` type and `True` for all other types.

```
sage: Qp(5, print_mode='val-unit', show_prec=False)(30)
5 * 6
```

print_max_terms, *print_sep* and *print_alphabet* have no effect.

Equality again depends on the printing options:

```
sage: R == S, R == T, S == T
(False, False, False)
```

3. **terse**: elements are displayed as an integer in base 10 or the quotient of an integer by a power of p (still in base 10):

```
sage: R = Qp(5, print_mode='terse'); a = R(70700); a
70700 + O(5^22)
sage: b = R(-70700); b
2384185790944925 + O(5^22)
sage: c = R(-707/5); c
95367431639918/5 + O(5^19)
```

The denominator, as of version 3.3, is always printed explicitly as a power of p, for predictability.

```
sage: d = R(707/5^2); d
707/5^2 + O(5^18)
```

print_pos controls whether to use a balanced representation or not.

```
sage: S = Qp(5, print_mode='terse', print_pos=False); b = S(-70700); b
-70700 + O(5^22)
sage: c = S(-707/5); c
-707/5 + O(5^19)
```

name affects how the name is printed.

```
sage: T.<unif> = Qp(5, print_mode='terse'); c = T(-707/5); c
95367431639918/unif + O(unif^19)
sage: d = T(-707/5^10); d
95367431639918/unif^10 + O(unif^10)
```

show_prec determines how the precision is printed. It can be either 'none' (or equivalently `False`) or 'bigoh' (or equivalently `True`). The default is `False` for the `'floating-point'` type and `True` for all other types.

```
sage: Qp(5, print_mode='terse', show_prec=False)(6)
6
```

print_max_terms, *print_sep* and *print_alphabet* have no effect.

Equality depends on printing options:

```
sage: R == S, R == T, S == T
(False, False, False)
```

4. **digits**: elements are displayed as a string of base p digits

   Restriction: you can only use the digits printing mode for small primes. Namely, p must be less than the length of the alphabet tuple (default alphabet has length 62).

```
sage: R = Qp(5, print_mode='digits'); a = R(70700); repr(a)
'...0000000000000004230300'
sage: b = R(-70700); repr(b)
'...4444444444444440214200'
sage: c = R(-707/5); repr(c)
'...44444444444444443413.3'
sage: d = R(-707/5^2); repr(d)
'...444444444444444341.33'
```

   Observe that the significant 0's are printed even if they are located in front of the number. On the contrary, unknown digits located after the comma appears as question marks. The precision can therefore be read in this mode as well. Here are more examples:

```
sage: p = 7
sage: K = Qp(p, prec=10, print_mode='digits')
sage: repr(K(1))
'...0000000001'
sage: repr(K(p^2))
'...0000000000100'
sage: repr(K(p^-5))
'...00000.00001'
sage: repr(K(p^-20))
'...?.??????????0000000001'
```

   print_max_terms limits the number of digits that are printed. Note that if the valuation of the element is very negative, more digits will be printed.

```
sage: S = Qp(5, print_max_terms=4); S(-70700)
2*5^2 + 4*5^3 + 5^4 + 2*5^5 + ... + O(5^22)
sage: S(-707/5^2)
3*5^-2 + 3*5^-1 + 1 + 4*5 + ... + O(5^18)
sage: S(-707/5^6)
3*5^-6 + 3*5^-5 + 5^-4 + 4*5^-3 + ... + O(5^14)
sage: S(-707/5^6,absprec=-2)
3*5^-6 + 3*5^-5 + 5^-4 + 4*5^-3 + O(5^-2)
sage: S(-707/5^4)
3*5^-4 + 3*5^-3 + 5^-2 + 4*5^-1 + ... + O(5^16)
```

   print_alphabet controls the symbols used to substitute for digits greater than 9.

Defaults to ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'):

```
sage: T = Qp(5, print_mode='digits', print_alphabet=('1','2','3','4','5'));␣
↪repr(T(-70700))
'...5555555555555551325311'
```

show_prec determines how the precision is printed. It can be either 'none' (or equivalently `False`), 'dots' (or equivalently `True`) or 'bigoh'. The default is `False` for the `'floating-point'` type and `True` for all other types.

```
sage: repr(Zp(5, print_mode='digits', show_prec=True)(6))
'...00000000000000000011'

sage: repr(Zp(5, print_mode='digits', show_prec='bigoh')(6))
'11 + O(5^20)'
```

print_pos, *name* and *print_sep* have no effect.

Equality depends on printing options:

```
sage: R == S, R == T, S == T
(False, False, False)
```

5. **bars**: elements are displayed as a string of base p digits with separators:

```
sage: R = Qp(5, print_mode='bars'); a = R(70700); repr(a)
'...4|2|3|0|3|0|0'
sage: b = R(-70700); repr(b)
'...4|4|4|4|4|4|4|4|4|4|4|4|4|4|4|0|2|1|4|2|0|0'
sage: d = R(-707/5^2); repr(d)
'...4|4|4|4|4|4|4|4|4|4|4|4|4|4|4|3|4|1|.|3|3'
```

Again, note that it's not possible to read off the precision from the representation in this mode.

print_pos controls whether the digits can be negative.

```
sage: S = Qp(5, print_mode='bars',print_pos=False); b = S(-70700); repr(b)
'...-1|0|2|2|-1|2|0|0'
```

print_max_terms limits the number of digits that are printed. Note that if the valuation of the element is very negative, more digits will be printed.

```
sage: T = Qp(5, print_max_terms=4); T(-70700)
2*5^2 + 4*5^3 + 5^4 + 2*5^5 + ... + O(5^22)
sage: T(-707/5^2)
3*5^-2 + 3*5^-1 + 1 + 4*5 + ... + O(5^18)
sage: T(-707/5^6)
3*5^-6 + 3*5^-5 + 5^-4 + 4*5^-3 + ... + O(5^14)
sage: T(-707/5^6,absprec=-2)
3*5^-6 + 3*5^-5 + 5^-4 + 4*5^-3 + O(5^-2)
sage: T(-707/5^4)
3*5^-4 + 3*5^-3 + 5^-2 + 4*5^-1 + ... + O(5^16)
```

print_sep controls the separation character.

```
sage: U = Qp(5, print_mode='bars', print_sep=']['); a = U(70700); repr(a)
'...4][2][3][0][3][0][0'
```

show_prec determines how the precision is printed. It can be either 'none' (or equivalently `False`), 'dots' (or equivalently `True`) or 'bigoh' The default is `False` for the `'floating-point'` type and `True` for all other types.

```
sage: repr(Qp(5, print_mode='bars', show_prec='bigoh')(6))
'...1|1 + O(5^20)'
```

name and *print_alphabet* have no effect.

Equality depends on printing options:

```
sage: R == S, R == T, R == U, S == T, S == U, T == U
(False, False, False, False, False, False)
```

EXAMPLES:

```
sage: K = Qp(15, check=False); a = K(999); a
9 + 6*15 + 4*15^2 + O(15^20)
```

create_key(*p*, *prec=None*, *type='capped-rel'*, *print_mode=None*, *names=None*, *ram_name=None*, *print_pos=None*, *print_sep=None*, *print_alphabet=None*, *print_max_terms=None*, *show_prec=None*, *check=True*, *label=None*)

Creates a key from input parameters for `Qp`.

See the documentation for `Qp` for more information.

create_object(*version*, *key*)

Creates an object using a given key.

See the documentation for `Qp` for more information.

sage.rings.padics.factory.**Qq**(*q*, *prec=None*, *type='capped-rel'*, *modulus=None*, *names=None*, *print_mode=None*, *ram_name=None*, *res_name=None*, *print_pos=None*, *print_sep=None*, *print_max_ram_terms=None*, *print_max_unram_terms=None*, *print_max_terse_terms=None*, *show_prec=None*, *check=True*, *implementation='FLINT'*)

Given a prime power $q = p^n$, return the unique unramified extension of $\mathbf{Q}_p$ of degree n.

INPUT:

- `q` – integer, list, tuple or `Factorization` object. If `q` is an integer, it is the prime power q in $\mathbf{Q}_q$. If `q` is a `Factorization` object, it is the factorization of the prime power q. As a tuple it is the pair `(p, n)`, and as a list it is a single element list `[(p, n)]`.
- `prec` – integer (default: `20`) the precision cap of the field. Individual elements keep track of their own precision. See TYPES and PRECISION below.
- `type` – string (default: `'capped-rel'`) Valid types are `'capped-rel'`, `'floating-point'`, `'lattice-cap'` and `'lattice-float'`. See TYPES and PRECISION below
- `modulus` – polynomial (default `None`) A polynomial defining an unramified extension of $\mathbf{Q}_p$. See MODULUS below.

- `names` – string or tuple (`None` is only allowed when $q = p$). The name of the generator, reducing to a generator of the residue field.
- `print_mode` – string (default: `None`). Valid modes are `'series'`, `'val-unit'`, `'terse'`, and `'bars'`. See PRINTING below.
- `ram_name` – string (defaults to string representation of p if None). `ram_name` controls how the prime is printed. See PRINTING below.
- `res_name` – string (defaults to `None`, which corresponds to adding a `'0'` to the end of the name). Controls how elements of the residue field print.
- `print_pos` – bool (default `None`) Whether to only use positive integers in the representations of elements. See PRINTING below.
- `print_sep` – string (default `None`) The separator character used in the `'bars'` mode. See PRINTING below.
- `print_max_ram_terms` – integer (default `None`) The maximum number of powers of p shown. See PRINTING below.
- `print_max_unram_terms` – integer (default `None`) The maximum number of entries shown in a coefficient of p. See PRINTING below.
- `print_max_terse_terms` – integer (default `None`) The maximum number of terms in the polynomial representation of an element (using `'terse'`). See PRINTING below.
- `show_prec` – bool (default `None`) whether to show the precision for elements. See PRINTING below.
- `check` – bool (default `True`) whether to check inputs.

OUTPUT:

- The corresponding unramified p-adic field.

TYPES AND PRECISION:

There are two types of precision for a p-adic element. The first is relative precision, which gives the number of known p-adic digits:

```
sage: R.<a> = Qq(25, 20, 'capped-rel', print_mode='series'); b = 25*a; b        #␣
↪needs sage.libs.ntl
a*5^2 + O(5^22)
sage: b.precision_relative()                                                    #␣
↪needs sage.libs.ntl
20
```

The second type of precision is absolute precision, which gives the power of p that this element is defined modulo:

```
sage: b.precision_absolute()                                                    #␣
↪needs sage.libs.ntl
22
```

There are two types of unramified p-adic fields: capped relative fields, floating point fields.

In the capped relative case, the relative precision of an element is restricted to be at most a certain value, specified at the creation of the field. Individual elements also store their own precision, so the effect of various arithmetic operations on precision is tracked. When you cast an exact element into a capped relative field, it truncates it to the precision cap of the field.

```
sage: R.<a> = Qq(9, 5, 'capped-rel', print_mode='series'); b = (1+2*a)^4; b      #␣
↪needs sage.libs.ntl
2 + (2*a + 2)*3 + (2*a + 1)*3^2 + O(3^5)
sage: c = R(3249); c                                                              #␣
↪needs sage.libs.ntl
3^2 + 3^4 + 3^5 + 3^6 + O(3^7)
sage: b + c                                                                       #␣
↪needs sage.libs.ntl
2 + (2*a + 2)*3 + (2*a + 2)*3^2 + 3^4 + O(3^5)
```

In the floating point case, elements do not track their precision, but the relative precision of elements is truncated during arithmetic to the precision cap of the field.

MODULUS:

The modulus needs to define an unramified extension of $\mathbf{Q}_p$: when it is reduced to a polynomial over $\mathbf{F}_p$ it should be irreducible.

The modulus can be given in a number of forms.

1. A **polynomial**.

   The base ring can be $\mathbf{Z}$, $\mathbf{Q}$, $\mathbf{Z}_p$, $\mathbf{Q}_p$, $\mathbf{F}_p$.

```
sage: # needs sage.libs.ntl
sage: P.<x> = ZZ[]
sage: R.<a> = Qq(27, modulus = x^3 + 2*x + 1); R.modulus()
(1 + O(3^20))*x^3 + O(3^20)*x^2 + (2 + O(3^20))*x + 1 + O(3^20)
sage: P.<x> = QQ[]
sage: S.<a> = Qq(27, modulus = x^3 + 2*x + 1)
sage: P.<x> = Zp(3)[]
sage: T.<a> = Qq(27, modulus = x^3 + 2*x + 1)
sage: P.<x> = Qp(3)[]
sage: U.<a> = Qq(27, modulus = x^3 + 2*x + 1)
sage: P.<x> = GF(3)[]                                                           ␣
↪     # needs sage.rings.finite_rings
sage: V.<a> = Qq(27, modulus = x^3 + 2*x + 1)
```

Which form the modulus is given in has no effect on the unramified extension produced:

```
sage: R == S, S == T, T == U, U == V                                            ␣
↪     # needs sage.libs.ntl
(True, True, True, False)
```

unless the precision of the modulus differs. In the case of V, the modulus is only given to precision 1, so the resulting field has a precision cap of 1.

```
sage: # needs sage.libs.ntl
sage: V.precision_cap()
1
sage: U.precision_cap()
20
sage: P.<x> = Qp(3)[]
sage: modulus = x^3 + (2 + O(3^7))*x + (1 + O(3^10))
sage: modulus
```

(continues on next page)

(continued from previous page)

```
(1 + O(3^20))*x^3 + (2 + O(3^7))*x + 1 + O(3^10)
sage: W.<a> = Qq(27, modulus = modulus); W.precision_cap()
7
```

2. The modulus can also be given as a **symbolic expression**.

```
sage: x = var('x')
↪# needs sage.symbolic
sage: X.<a> = Qq(27, modulus = x^3 + 2*x + 1); X.modulus()
↪# needs sage.symbolic
(1 + O(3^20))*x^3 + O(3^20)*x^2 + (2 + O(3^20))*x + 1 + O(3^20)
sage: X == R
↪# needs sage.libs.ntl sage.symbolic
True
```

By default, the polynomial chosen is the standard lift of the generator chosen for $\mathbf{F}_q$.

```
sage: GF(125, 'a').modulus()                                                    ␣
↪    # needs sage.rings.finite_rings
x^3 + 3*x + 3
sage: Y.<a> = Qq(125); Y.modulus()                                              ␣
↪    # needs sage.libs.ntl
(1 + O(5^20))*x^3 + O(5^20)*x^2 + (3 + O(5^20))*x + 3 + O(5^20)
```

However, you can choose another polynomial if desired (as long as the reduction to $\mathbf{F}_p[x]$ is irreducible).

```
sage: P.<x> = ZZ[]
sage: Z.<a> = Qq(125, modulus = x^3 + 3*x^2 + x + 1); Z.modulus()              ␣
↪    # needs sage.libs.ntl
(1 + O(5^20))*x^3 + (3 + O(5^20))*x^2 + (1 + O(5^20))*x + 1 + O(5^20)
sage: Y == Z                                                                    ␣
↪    # needs sage.libs.ntl
False
```

PRINTING:

There are many different ways to print p-adic elements. The way elements of a given field print is controlled by options passed in at the creation of the field. There are four basic printing modes (`'series'`, `'val-unit'`, `'terse'` and `'bars'`; `'digits'` is not available), as well as various options that either hide some information in the print representation or sometimes make print representations more compact. Note that the printing options affect whether different p-adic fields are considered equal.

1. **series**: elements are displayed as series in p.

```
sage: R.<a> = Qq(9, 20, 'capped-rel', print_mode='series'); (1+2*a)^4
↪# needs sage.libs.ntl
2 + (2*a + 2)*3 + (2*a + 1)*3^2 + O(3^20)
sage: -3*(1+2*a)^4
↪# needs sage.libs.ntl
3 + a*3^2 + 3^3 + (2*a + 2)*3^4 + (2*a + 2)*3^5 + (2*a + 2)*3^6 + (2*a + 2)*3^7
 + (2*a + 2)*3^8 + (2*a + 2)*3^9 + (2*a + 2)*3^10 + (2*a + 2)*3^11
```

(continues on next page)

(continued from previous page)

```
 + (2*a + 2)*3^12 + (2*a + 2)*3^13 + (2*a + 2)*3^14 + (2*a + 2)*3^15
 + (2*a + 2)*3^16 + (2*a + 2)*3^17 + (2*a + 2)*3^18 + (2*a + 2)*3^19
 + (2*a + 2)*3^20 + O(3^21)
sage: ~(3*a+18)
↪# needs sage.libs.ntl
(a + 2)*3^-1 + 1 + 2*3 + (a + 1)*3^2 + 3^3 + 2*3^4 + (a + 1)*3^5 + 3^6 + 2*3^7
 + (a + 1)*3^8 + 3^9 + 2*3^10 + (a + 1)*3^11 + 3^12 + 2*3^13 + (a + 1)*3^14
 + 3^15 + 2*3^16 + (a + 1)*3^17 + 3^18 + O(3^19)
```

print_pos controls whether negatives can be used in the coefficients of powers of p.

```
sage: S.<b> = Qq(9, print_mode='series', print_pos=False); (1+2*b)^4
↪    # needs sage.libs.ntl
-1 - b*3 - 3^2 + (b + 1)*3^3 + O(3^20)
sage: -3*(1+2*b)^4
↪    # needs sage.libs.ntl
3 + b*3^2 + 3^3 + (-b - 1)*3^4 + O(3^21)
```

ram_name controls how the prime is printed.

```
sage: T.<d> = Qq(9, print_mode='series', ram_name='p'); 3*(1+2*d)^4
↪    # needs sage.libs.ntl
2*p + (2*d + 2)*p^2 + (2*d + 1)*p^3 + O(p^21)
```

print_max_ram_terms limits the number of powers of p that appear.

```
sage: U.<e> = Qq(9, print_mode='series', print_max_ram_terms=4); repr(-
↪3*(1+2*e)^4)          # needs sage.libs.ntl
'3 + e*3^2 + 3^3 + (2*e + 2)*3^4 + ... + O(3^21)'
```

print_max_unram_terms limits the number of terms that appear in a coefficient of a power of p.

```
sage: # needs sage.libs.ntl
sage: V.<f> = Qq(128, prec = 8, print_mode='series'); repr((1+f)^9)
'(f^3 + 1) + (f^5 + f^4 + f^3 + f^2)*2 + (f^6 + f^5 + f^4 + f + 1)*2^2 + (f^
↪5 + f^4 + f^2 + f + 1)*2^3 + (f^6 + f^5 + f^4 + f^3 + f^2 + f + 1)*2^4 +
↪(f^5 + f^4)*2^5 + (f^6 + f^5 + f^4 + f^3 + f + 1)*2^6 + (f + 1)*2^7 + O(2^
↪8)'
sage: V.<f> = Qq(128, prec = 8, print_mode='series', print_max_unram_terms
↪= 3); repr((1+f)^9)
'(f^3 + 1) + (f^5 + f^4 + ... + f^2)*2 + (f^6 + f^5 + ... + 1)*2^2 + (f^5 +
↪f^4 + ... + 1)*2^3 + (f^6 + f^5 + ... + 1)*2^4 + (f^5 + f^4)*2^5 + (f^6 +
↪f^5 + ... + 1)*2^6 + (f + 1)*2^7 + O(2^8)'
sage: V.<f> = Qq(128, prec = 8, print_mode='series', print_max_unram_terms
↪= 2); repr((1+f)^9)
'(f^3 + 1) + (f^5 + ... + f^2)*2 + (f^6 + ... + 1)*2^2 + (f^5 + ... + 1)*2^
↪3 + (f^6 + ... + 1)*2^4 + (f^5 + f^4)*2^5 + (f^6 + ... + 1)*2^6 + (f +
↪1)*2^7 + O(2^8)'
sage: V.<f> = Qq(128, prec = 8, print_mode='series', print_max_unram_terms
↪= 1); repr((1+f)^9)
'(f^3 + ...) + (f^5 + ...)*2 + (f^6 + ...)*2^2 + (f^5 + ...)*2^3 + (f^6 + ..
↪.)*2^4 + (f^5 + ...)*2^5 + (f^6 + ...)*2^6 + (f + ...)*2^7 + O(2^8)'
```

(continues on next page)

(continued from previous page)

```
sage: V.<f> = Qq(128, prec = 8, print_mode='series', print_max_unram_terms␣
↪= 0); repr((1+f)^9 - 1 - f^3)
'(...)*2 + (...)*2^2 + (...)*2^3 + (...)*2^4 + (...)*2^5 + (...)*2^6 + (...
↪)*2^7 + O(2^8)'
```

show_prec determines how the precision is printed. It can be either 'none' (or equivalently `False`), 'bigoh' (or equivalently `True`). The default is `False` for the `'floating-point'` type and `True` for all other types.

```
sage: U.<e> = Qq(9, 2, show_prec=False); repr(-3*(1+2*e)^4)                 ␣
↪    # needs sage.libs.ntl
'3 + e*3^2'
```

print_sep and *print_max_terse_terms* have no effect.

Note that print options affect equality:

```
sage: R == S, R == T, R == U, R == V, S == T, S == U, S == V, T == U, T ==␣
↪V, U == V         # needs sage.libs.ntl
(False, False, False, False, False, False, False, False, False, False)
```

2. **val-unit**: elements are displayed as $p^k u$:

```
sage: R.<a> = Qq(9, 7, print_mode='val-unit'); b = (1+3*a)^9 - 1; b
↪# needs sage.libs.ntl
3^3 * (15 + 64*a) + O(3^7)
sage: ~b
↪# needs sage.libs.ntl
3^-3 * (41 + a) + O(3)
```

print_pos controls whether to use a balanced representation or not.

```
sage: S.<a> = Qq(9, 7, print_mode='val-unit', print_pos=False)              ␣
↪    # needs sage.libs.ntl
sage: b = (1+3*a)^9 - 1; b                                                  ␣
↪    # needs sage.libs.ntl
3^3 * (15 - 17*a) + O(3^7)
sage: ~b                                                                    ␣
↪    # needs sage.libs.ntl
3^-3 * (-40 + a) + O(3)
```

ram_name affects how the prime is printed.

```
sage: # needs sage.libs.ntl
sage: A.<x> = Qp(next_prime(10^6), print_mode='val-unit')[]
sage: T.<a> = Qq(next_prime(10^6)^3, 4, print_mode='val-unit', ram_name='p',
....:            modulus=x^3+385831*x^2+106556*x+321036)
sage: b = ~(next_prime(10^6)^2*(a^2 + a - 4)); b
p^-2 * (503009563508519137754940 + 704413692798200940253892*a
        + 968097057817740999537581*a^2) + O(p^2)
sage: b * (a^2 + a - 4)
p^-2 * 1 + O(p^2)
```

print_max_terse_terms controls how many terms of the polynomial appear in the unit part.

```
sage: U.<a> = Qq(17^4, 6, print_mode='val-unit', print_max_terse_terms=3)   ␣
↪    # needs sage.libs.ntl
sage: b = ~(17*(a^3-a+14)); b                                              ␣
↪    # needs sage.libs.ntl
17^-1 * (22110411 + 11317400*a + 20656972*a^2 + ...) + O(17^5)
sage: b*17*(a^3-a+14)                                                      ␣
↪    # needs sage.libs.ntl
1 + O(17^6)
```

show_prec determines how the precision is printed. It can be either 'none' (or equivalently `False`), 'bigoh' (or equivalently `True`). The default is `False` for the `'floating-point'` type and `True` for all other types.

```
sage: U.<e> = Qq(9, 2, print_mode='val-unit', show_prec=False); repr(-
↪3*(1+2*e)^4)          # needs sage.libs.ntl
'3 * (1 + 3*e)'
```

print_sep, *print_max_ram_terms* and *print_max_unram_terms* have no effect.

Equality again depends on the printing options:

```
sage: R == S, R == T, R == U, S == T, S == U, T == U                      ␣
↪    # needs sage.libs.ntl
(False, False, False, False, False, False)
```

3. **terse**: elements are displayed as a polynomial of degree less than the degree of the extension.

```
sage: R.<a> = Qq(125, print_mode='terse')
↪# needs sage.libs.ntl
sage: (a+5)^177
↪# needs sage.libs.ntl
68210977979428 + 90313850704069*a + 73948093055069*a^2 + O(5^20)
sage: (a/5+1)^177
↪# needs sage.libs.ntl
68210977979428/5^177 + 90313850704069/5^177*a + 73948093055069/5^177*a^2 + O(5^-
↪157)
```

As of version 3.3, if coefficients of the polynomial are non-integral, they are always printed with an explicit power of p in the denominator.

```
sage: 5*a + a^2/25                                                         ␣
↪    # needs sage.libs.ntl
5*a + 1/5^2*a^2 + O(5^18)
```

print_pos controls whether to use a balanced representation or not.

```
sage: (a-5)^6                                                              ␣
↪    # needs sage.libs.ntl
22864 + 95367431627998*a + 8349*a^2 + O(5^20)
sage: S.<a> = Qq(125, print_mode='terse', print_pos=False); b = (a-5)^6; b ␣
↪    # needs sage.libs.ntl
22864 - 12627*a + 8349*a^2 + O(5^20)
```

(continues on next page)

(continued from previous page)

```
sage: (a - 1/5)^6
↪     # needs sage.libs.ntl
-20624/5^6 + 18369/5^5*a + 1353/5^3*a^2 + O(5^14)
```

ram_name affects how the prime is printed.

```
sage: T.<a> = Qq(125, print_mode='terse', ram_name='p'); (a - 1/5)^6
↪     # needs sage.libs.ntl
95367431620001/p^6 + 18369/p^5*a + 1353/p^3*a^2 + O(p^14)
```

print_max_terse_terms controls how many terms of the polynomial are shown.

```
sage: U.<a> = Qq(625, print_mode='terse', print_max_terse_terms=2); (a-1/5)^
↪6   # needs sage.libs.ntl
106251/5^6 + 49994/5^5*a + ... + O(5^14)
```

show_prec determines how the precision is printed. It can be either 'none' (or equivalently `False`), 'bigoh' (or equivalently `True`). The default is `False` for the `'floating-point'` type and `True` for all other types.

```
sage: U.<e> = Qq(9, 2, print_mode='terse', show_prec=False); repr(-
↪3*(1+2*e)^4)              # needs sage.libs.ntl
'3 + 9*e'
```

print_sep, *print_max_ram_terms* and *print_max_unram_terms* have no effect.

Equality again depends on the printing options:

```
sage: R == S, R == T, R == U, S == T, S == U, T == U
↪     # needs sage.libs.ntl
(False, False, False, False, False, False)
```

4. **digits**: This print mode is not available when the residue field is not prime.

   It might make sense to have a dictionary for small fields, but this isn't implemented.

5. **bars**: elements are displayed in a similar fashion to series, but more compactly.

```
sage: R.<a> = Qq(125); (a+5)^6
↪# needs sage.libs.ntl
(4*a^2 + 3*a + 4) + (3*a^2 + 2*a)*5 + (a^2 + a + 1)*5^2 + (3*a + 2)*5^3
 + (3*a^2 + a + 3)*5^4 + (2*a^2 + 3*a + 2)*5^5 + O(5^20)
sage: R.<a> = Qq(125, print_mode='bars', prec=8); repr((a+5)^6)
↪# needs sage.libs.ntl
'...[2, 3, 2]|[3, 1, 3]|[2, 3]|[1, 1, 1]|[0, 2, 3]|[4, 3, 4]'
sage: repr((a-5)^6)
↪# needs sage.libs.ntl
'...[0, 4]|[1, 4]|[2, 0, 2]|[1, 4, 3]|[2, 3, 1]|[4, 4, 3]|[2, 4, 4]|[4, 3, 4]'
```

Note that elements with negative valuation are shown with a decimal point at valuation 0.

```
sage: repr((a+1/5)^6)
↪     # needs sage.libs.ntl
'...[3]|[4, 1, 3]|.|[1, 2, 3]|[3, 3]|[0, 0, 3]|[0, 1]|[0, 1]|[1]'
```

(continues on next page)

(continued from previous page)

```
sage: repr((a+1/5)^2)                                                    ␣
↪    # needs sage.libs.ntl
'...[0, 0, 1]|.|[0, 2]|[1]'
```

If not enough precision is known, `'?'` is used instead.

```
sage: repr((a+R(1/5,relprec=3))^7)                                       ␣
↪    # needs sage.libs.ntl
'...|.|?|?|?|?|[0, 1, 1]|[0, 2]|[1]'
```

Note that it's not possible to read off the precision from the representation in this mode.

```
sage: # needs sage.libs.ntl
sage: b = a + 3; repr(b)
'...[3, 1]'
sage: c = a + R(3, 4); repr(c)
'...[3, 1]'
sage: b.precision_absolute()
8
sage: c.precision_absolute()
4
```

print_pos controls whether the digits can be negative.

```
sage: S.<a> = Qq(125, print_mode='bars', print_pos=False); repr((a-5)^6)   ␣
↪    # needs sage.libs.ntl
'...[1, -1, 1]|[2, 1, -2]|[2, 0, -2]|[-2, -1, 2]|[0, 0, -1]|[-2]|[-1, -2, -
↪1]'
sage: repr((a-1/5)^6)                                                    ␣
↪    # needs sage.libs.ntl
'...[0, 1, 2]|[-1, 1, 1]|.|[-2, -1, -1]|[2, 2, 1]|[0, 0, -2]|[0, -1]|[0, -
↪1]|[1]'
```

print_max_ram_terms controls the maximum number of "digits" shown. Note that this puts a cap on the relative precision, not the absolute precision.

```
sage: T.<a> = Qq(125, print_max_ram_terms=3, print_pos=False); (a-5)^6    ␣
↪    # needs sage.libs.ntl
(-a^2 - 2*a - 1) - 2*5 - a^2*5^2 + ... + O(5^20)
sage: 5*(a-5)^6 + 50                                                     ␣
↪    # needs sage.libs.ntl
(-a^2 - 2*a - 1)*5 - a^2*5^3 + (2*a^2 - a - 2)*5^4 + ... + O(5^21)
```

print_sep controls the separating character (`'|'` by default).

```
sage: U.<a> = Qq(625, print_mode='bars', print_sep=''); b = (a+5)^6;␣
↪repr(b)     # needs sage.libs.ntl
'...[0, 1][4, 0, 2][3, 2, 2, 3][4, 2, 2, 4][0, 3][1, 1, 3][3, 1, 4, 1]'
```

print_max_unram_terms controls how many terms are shown in each "digit":

```
sage: # needs sage.libs.ntl
sage: with local_print_mode(U, {'max_unram_terms': 3}): repr(b)
```

(continues on next page)

(continued from previous page)

```
'...[0, 1][4,..., 0, 2][3,..., 2, 3][4,..., 2, 4][0, 3][1,..., 1, 3][3,...,␣
↪4, 1]'
sage: with local_print_mode(U, {'max_unram_terms': 2}): repr(b)
'...[0, 1][4,..., 2][3,..., 3][4,..., 4][0, 3][1,..., 3][3,..., 1]'
sage: with local_print_mode(U, {'max_unram_terms': 1}): repr(b)
'...[..., 1][..., 2][..., 3][..., 4][..., 3][..., 3][..., 1]'
sage: with local_print_mode(U, {'max_unram_terms':0}): repr(b-75*a)
'...[...][...][...][...][][...][...]'
```

show_prec determines how the precision is printed. It can be either 'none' (or equivalently `False`), 'dots' (or equivalently `True`) or 'bigoh' The default is `False` for the `'floating-point'` type and `True` for all other types.

```
sage: U.<e> = Qq(9, 2, print_mode='bars', show_prec=True); repr(-3*(1+2*e)^
↪4)   # needs sage.libs.ntl
'...[0, 1]|[1]|[]'
```

ram_name and *print_max_terse_terms* have no effect.

Equality depends on printing options:

```
sage: R == S, R == T, R == U, S == T, S == U, T == U                      ␣
↪    # needs sage.libs.ntl
(False, False, False, False, False)
```

EXAMPLES:

Unlike for `Qp`, you can't create `Qq(N)` when `N` is not a prime power.

However, you can use `check=False` to pass in a pair in order to not have to factor. If you do so, you need to use names explicitly rather than the `R.<a>` syntax.

```
sage: p = next_prime(2^123)
sage: k = Qp(p)
sage: R.<x> = k[]                                                        #␣
↪needs sage.libs.ntl
sage: K = Qq([(p, 5)], modulus=x^5+x+4, names='a', ram_name='p',         #␣
↪needs sage.libs.ntl
....:        print_pos=False, check=False)
sage: K.0^5                                                              #␣
↪needs sage.libs.ntl
(-a - 4) + O(p^20)
```

In tests on `sage.math.washington.edu`, the creation of `K` as above took an average of 1.58ms, while:

```
sage: K = Qq(p^5, modulus=x^5+x+4, names='a', ram_name='p',              #␣
↪needs sage.libs.ntl
....:        print_pos=False, check=True)
```

took an average of 24.5ms. Of course, with smaller primes these savings disappear.

`sage.rings.padics.factory.`**`QqCR`**(*q*, *prec=None*, **args*, ***kwds*)

A shortcut function to create capped relative unramified p-adic fields.

Same functionality as *Qq()*. See documentation for *Qq()* for a description of the input parameters.

EXAMPLES:

```
sage: R.<a> = QqCR(25, 40); R                                              #␣
↪needs sage.libs.ntl
5-adic Unramified Extension Field in a defined by x^2 + 4*x + 2
```

sage.rings.padics.factory.**QqFP**(*q*, *prec=None*, **args*, ***kwds*)

A shortcut function to create floating point unramified p-adic fields.

Same functionality as *Qq()*. See documentation for *Qq()* for a description of the input parameters.

EXAMPLES:

```
sage: R.<a> = QqFP(25, 40); R                                              #␣
↪needs sage.libs.flint
5-adic Unramified Extension Field in a defined by x^2 + 4*x + 2
```

sage.rings.padics.factory.**ZpCA**(*p*, *prec=None*, **args*, ***kwds*)

A shortcut function to create capped absolute p-adic rings.

See documentation for `Zp()` for a description of the input parameters.

EXAMPLES:

```
sage: ZpCA(5, 40)
5-adic Ring with capped absolute precision 40
```

sage.rings.padics.factory.**ZpCR**(*p*, *prec=None*, **args*, ***kwds*)

A shortcut function to create capped relative p-adic rings.

Same functionality as `Zp()`. See documentation for `Zp()` for a description of the input parameters.

EXAMPLES:

```
sage: ZpCR(5, 40)
5-adic Ring with capped relative precision 40
```

sage.rings.padics.factory.**ZpER**(*p*, *prec=None*, *halt=None*, *secure=False*, **args*, ***kwds*)

A shortcut function to create relaxed p-adic rings.

INPUT:

- `prec` – an integer (default: `20`), the default precision
- `halt` – an integer (default: twice `prec`), the halting precision
- `secure` – a boolean (default: `False`); if `False`, consider indistinguishable elements at the working precision as equal; otherwise, raise an error.

See documentation for `Zp()` for a description of the other input parameters.

A SHORT INTRODUCTION TO RELAXED p-ADICS:

The model for relaxed p-adics is quite different from any of the other types of p-adics. In addition to storing a finite approximation, one also stores a method for increasing the precision.

Relaxed p-adic rings are created by the constructor *ZpER()*:

```
sage: R = ZpER(5, print_mode="digits"); R                                  #␣
↪needs sage.libs.flint
5-adic Ring handled with relaxed arithmetics
```

The precision is not capped in R:

```
sage: R.precision_cap()                                                    #␣
↪needs sage.libs.flint
+Infinity
```

However, a default precision is fixed. This is the precision at which the elements will be printed:

```
sage: R.default_prec()                                                     #␣
↪needs sage.libs.flint
20
```

A default halting precision is also set. It is the default absolute precision at which the elements will be compared. By default, it is twice the default precision:

```
sage: R.halting_prec()                                                     #␣
↪needs sage.libs.flint
40
```

However, both the default precision and the halting precision can be customized at the creation of the parent as follows:

```
sage: S = ZpER(5, prec=10, halt=100)                                       #␣
↪needs sage.libs.flint
sage: S.default_prec()                                                     #␣
↪needs sage.libs.flint
10
sage: S.halting_prec()                                                     #␣
↪needs sage.libs.flint
100
```

One creates elements as usual:

```
sage: a = R(17/42); a                                                      #␣
↪needs sage.libs.flint
...00244200244200244201

sage: R.random_element()   # random                                        #␣
↪needs sage.libs.flint
...21013213133412431402
```

Here we notice that 20 digits (that is the default precision) are printed. However, the computation model is designed in order to guarantee that more digits of a will be available on demand. This feature is reflected by the fact that, when we ask for the precision of a, the software answers $+\infty$:

```
sage: a.precision_absolute()                                               #␣
↪needs sage.libs.flint
+Infinity
```

Asking for more digits is achieved by the methods `at_precision_absolute()` and `at_precision_relative()`:

```
sage: a.at_precision_absolute(30)                                          #␣
↪needs sage.libs.flint
...?244200244200244200244200244201
```

As a shortcut, one can use the bracket operator:

```
sage: a[:30]                                                                      #␣
↪needs sage.libs.flint
...?244200244200244200244200244201
```

Of course, standard operations are supported:

```
sage: # needs sage.libs.flint
sage: b = R(42/17)
sage: a + b
...03232011214322140002
sage: a - b
...42311334324023403400
sage: a * b
...00000000000000000001
sage: a / b
...12442142113021233401
sage: sqrt(a)
...20042333114021142101
```

We observe again that only 20 digits are printed but, as before, more digits are available on demand:

```
sage: sqrt(a)[:30]                                                                #␣
↪needs sage.libs.flint
...?142443342120042333114021142101
```

Equality tests

Checking equalities between relaxed p-adics is a bit subtle and can sometimes be puzzling at first glance.

When the parent is created with `secure=False` (which is the default), elements are compared at the current precision, or at the default halting precision if it is higher:

```
sage: a == b                                                                      #␣
↪needs sage.libs.flint
False

sage: a == sqrt(a)^2                                                              #␣
↪needs sage.libs.flint
True
sage: a == sqrt(a)^2 + 5^50                                                       #␣
↪needs sage.libs.flint
True
```

In the above example, the halting precision is 40; it is the reason why a congruence modulo 5^50 is considered as an equality. However, if both sides of the equalities have been previously computed with more digits, those digits are taken into account. Hence comparing two elements at different times can produce different results:

```
sage: # needs sage.libs.flint
sage: aa = sqrt(a)^2 + 5^50
sage: a == aa
True
```

(continues on next page)

(continued from previous page)

```
sage: a[:60]
...?244200244200244200244200244200244200244200244200244200244201
sage: aa[:60]
...?244200244300244200244200244200244200244200244200244200244201
sage: a == aa
False
```

This annoying situation, where the output of `a == aa` may change depending on previous computations, cannot occur when the parent is created with `secure=True`. Indeed, in this case, if the equality cannot be decided, an error is raised:

```
sage: # needs sage.libs.flint
sage: S = ZpER(5, secure=True)
sage: u = S.random_element()
sage: uu = u + 5^50
sage: u == uu
Traceback (most recent call last):
...
PrecisionError: unable to decide equality; try to bound precision

sage: u[:60] == uu                                                    #␣
↪needs sage.libs.flint
False
```

Self-referent numbers

A quite interesting feature with relaxed p-adics is the possibility to create (in some cases) self-referent numbers. Here is an example. We first declare a new variable as follows:

```
sage: x = R.unknown(); x                                              #␣
↪needs sage.libs.flint
...?.0
```

We then use the method `set()` to define x by writing down an equation it satisfies:

```
sage: x.set(1 + 5*x^2)                                                #␣
↪needs sage.libs.flint
True
```

The variable x now contains the unique solution of the equation $x = 1 + 5x^2$:

```
sage: x                                                               #␣
↪needs sage.libs.flint
...04222412141121000211
```

This works because the n-th digit of the right hand size of the defining equation only involves the i-th digits of x with $i < n$ (this is due to the factor 5).

As a comparison, the following does not work:

```
sage: # needs sage.libs.flint
sage: y = R.unknown()
```

(continues on next page)

(continued from previous page)

```
sage: y.set(1 + 3*y^2)
True
sage: y
...?.0
sage: y[:20]
Traceback (most recent call last):
...
RecursionError: definition looks circular
```

Self-referent definitions also work with systems of equations:

```
sage: # needs sage.libs.flint
sage: u = R.unknown()
sage: v = R.unknown()
sage: w = R.unknown()
sage: u.set(1 + 2*v + 3*w^2 + 5*u*v*w)
True
sage: v.set(2 + 4*w + sqrt(1 + 5*u + 10*v + 15*w))
True
sage: w.set(3 + 25*(u*v + v*w + u*w))
True
sage: u
...31203130103131131433
sage: v
...33441043031103114240
sage: w
...30212422041102444403
```

sage.rings.padics.factory.**ZpFM**(*p*, *prec=None*, **args*, ***kwds*)

A shortcut function to create fixed modulus p-adic rings.

See documentation for `Zp()` for a description of the input parameters.

EXAMPLES:

```
sage: ZpFM(5, 40)
5-adic Ring of fixed modulus 5^40
```

sage.rings.padics.factory.**ZpFP**(*p*, *prec=None*, **args*, ***kwds*)

A shortcut function to create floating point p-adic rings.

Same functionality as `Zp()`. See documentation for `Zp()` for a description of the input parameters.

EXAMPLES:

```
sage: ZpFP(5, 40)
5-adic Ring with floating precision 40
```

sage.rings.padics.factory.**ZpLC**(*p*, *prec=None*, **args*, ***kwds*)

A shortcut function to create p-adic rings with lattice precision (precision is encoded by a lattice in a large vector space and tracked using automatic differentiation).

See documentation for `Zp()` for a description of the input parameters.

EXAMPLES:

Below is a small demo of the features by this model of precision:

```
sage: R = ZpLC(3, print_mode='terse')
sage: R
3-adic Ring with lattice-cap precision

sage: x = R(1,10)
```

Of course, when we multiply by 3, we gain one digit of absolute precision:

```
sage: 3*x
3 + O(3^11)
```

The lattice precision machinery sees this even if we decompose the computation into several steps:

```
sage: y = x+x
sage: y
2 + O(3^10)
sage: x + y
3 + O(3^11)
```

The same works for the multiplication:

```
sage: z = x^2
sage: z
1 + O(3^10)
sage: x*z
1 + O(3^11)
```

This can be more surprising when we are working with elements given at different precisions:

```
sage: R = ZpLC(2, print_mode='terse')
sage: x = R(1,10)
sage: y = R(1,5)
sage: z = x+y; z
2 + O(2^5)
sage: t = x-y; t
O(2^5)
sage: z+t  # observe that z+t = 2*x
2 + O(2^11)
sage: z-t  # observe that z-t = 2*y
2 + O(2^6)

sage: x = R(28888,15)
sage: y = R(204,10)
sage: z = x/y; z
242 + O(2^9)
sage: z*y  # which is x
28888 + O(2^15)
```

The SOMOS sequence is the sequence defined by the recurrence:

$$u_n = \frac{u_{n-1}u_{n-3} + u_{n-2}^2}{u_{n-4}}$$

It is known for its numerical instability. On the one hand, one can show that if the initial values are invertible in $\mathbf{Z}_p$ and known at precision $O(p^N)$ then all the next terms of the SOMOS sequence will be known at the same precision as well. On the other hand, because of the division, when we unroll the recurrence, we loose a lot of precision. Observe:

```
sage: R = Zp(2, 30, print_mode='terse')
sage: a,b,c,d = R(1,15), R(1,15), R(1,15), R(3,15)
sage: a,b,c,d = b,c,d,(b*d+c*c)/a; print(d)
4 + O(2^15)
sage: a,b,c,d = b,c,d,(b*d+c*c)/a; print(d)
13 + O(2^15)
sage: a,b,c,d = b,c,d,(b*d+c*c)/a; print(d)
55 + O(2^15)
sage: a,b,c,d = b,c,d,(b*d+c*c)/a; print(d)
21975 + O(2^15)
sage: a,b,c,d = b,c,d,(b*d+c*c)/a; print(d)
6639 + O(2^13)
sage: a,b,c,d = b,c,d,(b*d+c*c)/a; print(d)
7186 + O(2^13)
sage: a,b,c,d = b,c,d,(b*d+c*c)/a; print(d)
569 + O(2^13)
sage: a,b,c,d = b,c,d,(b*d+c*c)/a; print(d)
253 + O(2^13)
sage: a,b,c,d = b,c,d,(b*d+c*c)/a; print(d)
4149 + O(2^13)
sage: a,b,c,d = b,c,d,(b*d+c*c)/a; print(d)
2899 + O(2^12)
sage: a,b,c,d = b,c,d,(b*d+c*c)/a; print(d)
3072 + O(2^12)
sage: a,b,c,d = b,c,d,(b*d+c*c)/a; print(d)
349 + O(2^12)
sage: a,b,c,d = b,c,d,(b*d+c*c)/a; print(d)
619 + O(2^12)
sage: a,b,c,d = b,c,d,(b*d+c*c)/a; print(d)
243 + O(2^12)
sage: a,b,c,d = b,c,d,(b*d+c*c)/a; print(d)
3 + O(2^2)
sage: a,b,c,d = b,c,d,(b*d+c*c)/a; print(d)
2 + O(2^2)
```

If instead, we use the lattice precision, everything goes well:

```
sage: R = ZpLC(2, 30, print_mode='terse')
sage: a,b,c,d = R(1,15), R(1,15), R(1,15), R(3,15)
sage: a,b,c,d = b,c,d,(b*d+c*c)/a; print(d)
4 + O(2^15)
sage: a,b,c,d = b,c,d,(b*d+c*c)/a; print(d)
13 + O(2^15)
sage: a,b,c,d = b,c,d,(b*d+c*c)/a; print(d)
55 + O(2^15)
sage: a,b,c,d = b,c,d,(b*d+c*c)/a; print(d)
21975 + O(2^15)
sage: a,b,c,d = b,c,d,(b*d+c*c)/a; print(d)
23023 + O(2^15)
```

(continues on next page)

(continued from previous page)

```
sage: a,b,c,d = b,c,d,(b*d+c*c)/a; print(d)
31762 + O(2^15)
sage: a,b,c,d = b,c,d,(b*d+c*c)/a; print(d)
16953 + O(2^15)
sage: a,b,c,d = b,c,d,(b*d+c*c)/a; print(d)
16637 + O(2^15)

sage: for _ in range(100):
....:     a,b,c,d = b,c,d,(b*d+c*c)/a
sage: a
15519 + O(2^15)
sage: b
32042 + O(2^15)
sage: c
17769 + O(2^15)
sage: d
20949 + O(2^15)
```

ALGORITHM:

The precision is global. It is encoded by a lattice in a huge vector space whose dimension is the number of elements having this parent. Precision is tracked using automatic differentiation techniques (see [CRV2014] and [CRV2018]).

Concretely, this precision datum is an instance of the class `sage.rings.padic.lattice_precision.PrecisionLattice`. It is attached to the parent and is created at the same time as the parent. (It is actually a bit more subtle because two different parents may share the same instance; this happens for instance for a p-adic ring and its field of fractions.)

This precision datum is accessible through the method `precision()`:

```
sage: R = ZpLC(5, print_mode='terse')
sage: prec = R.precision()
sage: prec
Precision lattice on 0 objects
```

This instance knows about all elements of the parent. It is automatically updated when a new element (of this parent) is created:

```
sage: x = R(3513,10)
sage: prec
Precision lattice on 1 object
sage: y = R(176,5)
sage: prec
Precision lattice on 2 objects
sage: z = R.random_element()
sage: prec
Precision lattice on 3 objects
```

The method `tracked_elements()` provides the list of all tracked elements:

```
sage: prec.tracked_elements()
[3513 + O(5^10), 176 + O(5^5), ...]
```

Similarly, when a variable is collected by the garbage collector, the precision lattice is updated. Note however that the update might be delayed. We can force it with the method `del_elements()`:

```
sage: z = 0
sage: prec  # random output, could be 2 objects if the garbage collector is fast
Precision lattice on 3 objects
sage: prec.del_elements()
sage: prec
Precision lattice on 2 objects
```

The method `precision_lattice()` returns (a matrix defining) the lattice that models the precision. Here we have:

```
sage: prec.precision_lattice()                                                    #␣
↪needs sage.geometry.polyhedron
[9765625       0]
[      0    3125]
```

Observe that $5^10 = 9765625$ and $5^5 = 3125$. The above matrix then reflects the precision on x and y.

Now, observe how the precision lattice changes while performing computations:

```
sage: x, y = 3*x+2*y, 2*(x-y)
sage: prec.del_elements()
sage: prec.precision_lattice()                                                    #␣
↪needs sage.geometry.polyhedron
[    3125 48825000]
[       0 48828125]
```

The matrix we get is no longer diagonal, meaning that some digits of precision are diffused among the two new elements x and y. They nevertheless show up when we compute for instance $x + y$:

```
sage: x
1516 + O(5^5)
sage: y
424 + O(5^5)
sage: x+y
17565 + O(5^11)
```

These diffused digits of precision (which are tracked but do not appear on the printing) allow to be always sharp on precision.

NOTE:

Each elementary operation requires significant manipulations on the precision lattice and therefore is costly. Precisely:

- The creation of a new element has a cost $O(n)$ where n is the number of tracked elements.
- The destruction of one element has a cost $O(m^2)$ where m is the distance between the destroyed element and the last one. Fortunately, it seems that m tends to be small in general (the dynamics of the list of tracked elements is rather close to that of a stack).

It is nevertheless still possible to manipulate several hundred variables (e.g. square matrices of size 5 or polynomials of degree 20).

The class `PrecisionLattice` provides several features for introspection, especially concerning timings. See `history()` and `timings()` for details.

See also:

ZpLF()

`sage.rings.padics.factory.ZpLF(p, prec=None, *args, **kwds)`

A shortcut function to create p-adic rings where precision is encoded by a module in a large vector space.

See documentation for `Zp()` for a description of the input parameters.

Note: The precision is tracked using automatic differentiation techniques (see [CRV2018] and [CRV2014]). Floating point p-adic numbers are used for the computation of the differential (which is then not exact).

EXAMPLES:

```
sage: R = ZpLF(5, 40)
sage: R
5-adic Ring with lattice-float precision
```

See also:

ZpLC()

`class sage.rings.padics.factory.Zp_class`

Bases: `UniqueFactory`

A creation function for p-adic rings.

INPUT:

- `p` – integer: the p in $\mathbf{Z}_p$
- `prec` – integer (default: `20`) the precision cap of the ring. In the lattice capped case, `prec` can either be a pair (`relative_cap`, `absolute_cap`) or an integer (understood as relative cap). In the relaxed case, `prec` can be either a pair (`default_prec`, `halting_prec`) or an integer (understood as default precision). Except for the fixed modulus and floating point cases, individual elements keep track of their own precision. See TYPES and PRECISION below.
- `type` – string (default: `'capped-rel'`) Valid types are `'capped-rel'`, `'capped-abs'`, `'fixed-mod'`, `'floating-point'`, `'lattice-cap'`, `'lattice-float'`, `'relaxed'`. See TYPES and PRECISION below.
- `print_mode` – string (default: `None`). Valid modes are `'series'`, `'val-unit'`, `'terse'`, `'digits'`, and `'bars'`. See PRINTING below.
- `names` – string or tuple (defaults to a string representation of p). What to use whenever p is printed.
- `print_pos` – bool (default `None`) Whether to only use positive integers in the representations of elements. See PRINTING below.
- `print_sep` – string (default `None`) The separator character used in the `'bars'` mode. See PRINTING below.
- `print_alphabet` – tuple (default `None`) The encoding into digits for use in the `'digits'` mode. See PRINTING below.
- `print_max_terms` – integer (default `None`) The maximum number of terms shown. See PRINTING below.
- `show_prec` – bool (default `None`) whether to show the precision for elements. See PRINTING below.
- `check` – bool (default `True`) whether to check if p is prime. Non-prime input may cause seg-faults (but can also be useful for base n expansions for example)

- `label` – string (default `None`) used for lattice precision to create parents with different lattices.

OUTPUT:

- The corresponding p-adic ring.

TYPES AND PRECISION:

There are two main types of precision. The first is relative precision; it gives the number of known p-adic digits:

```
sage: R = Zp(5, 20, 'capped-rel', 'series'); a = R(675); a
2*5^2 + 5^4 + O(5^22)
sage: a.precision_relative()
20
```

The second type of precision is absolute precision, which gives the power of p that this element is defined modulo:

```
sage: a.precision_absolute()
22
```

There are several types of p-adic rings, depending on the methods used for tracking precision. Namely, we have:

- capped relative rings (`type='capped-rel'`)
- capped absolute rings (`type='capped-abs'`)
- fixed modulus rings (`type='fixed-mod'`)
- floating point rings (`type='floating-point'`)
- lattice precision rings (`type='lattice-cap'` or `type='lattice-float'`)
- exact fields with relaxed arithmetics (`type='relaxed'`)

In the capped relative case, the relative precision of an element is restricted to be at most a certain value, specified at the creation of the field. Individual elements also store their own precision, so the effect of various arithmetic operations on precision is tracked. When you cast an exact element into a capped relative field, it truncates it to the precision cap of the field.

```
sage: R = Zp(5, 5, 'capped-rel', 'series'); a = R(4006); a
1 + 5 + 2*5^3 + 5^4 + O(5^5)
sage: b = R(4025); b
5^2 + 2*5^3 + 5^4 + 5^5 + O(5^7)
sage: a + b
1 + 5 + 5^2 + 4*5^3 + 2*5^4 + O(5^5)
```

In the capped absolute type, instead of having a cap on the relative precision of an element there is instead a cap on the absolute precision. Elements still store their own precisions, and as with the capped relative case, exact elements are truncated when cast into the ring.

```
sage: R = Zp(5, 5, 'capped-abs', 'series'); a = R(4005); a
5 + 2*5^3 + 5^4 + O(5^5)
sage: b = R(4025); b
5^2 + 2*5^3 + 5^4 + O(5^5)
sage: a * b
5^3 + 2*5^4 + O(5^5)
sage: (a * b) // 5^3
1 + 2*5 + O(5^2)
```

The fixed modulus type is the leanest of the p-adic rings: it is basically just a wrapper around $\mathbf{Z}/p^n\mathbf{Z}$ providing a unified interface with the rest of the p-adics. This is the type you should use if your sole interest is speed. It does not track precision of elements.

```
sage: R = Zp(5,5,'fixed-mod','series'); a = R(4005); a
5 + 2*5^3 + 5^4
sage: a // 5
1 + 2*5^2 + 5^3
```

The floating point case is similar to the fixed modulus type in that elements do not trac their own precision. However, relative precision is truncated with each operation rather than absolute precision.

On the contrary, the lattice type tracks precision using lattices and automatic differentiation. It is rather slow but provides sharp (often optimal) results regarding precision. We refer to the documentation of the function *ZpLC()* for a small demonstration of the capabilities of this precision model.

Finally, the model for relaxed p-adics is quite different from any of the other types. In addition to storing a finite approximation, one also stores a method for increasing the precision. A quite interesting feature with relaxed p-adics is the possibility to create (in some cases) self-referent numbers, that are numbers whose n-th digit is defined by the previous ones. We refer to the documentation of the function `ZpL()` for a small demonstration of the capabilities of this precision model.

PRINTING:

There are many different ways to print p-adic elements. The way elements of a given ring print is controlled by options passed in at the creation of the ring. There are five basic printing modes (`'series'`, `'val-unit'`, `'terse'`, `'digits'` and `'bars'`), as well as various options that either hide some information in the print representation or sometimes make print representations more compact. Note that the printing options affect whether different p-adic fields are considered equal.

1. **series**: elements are displayed as series in p.

   ```
   sage: R = Zp(5, print_mode='series'); a = R(70700); a
   3*5^2 + 3*5^4 + 2*5^5 + 4*5^6 + O(5^22)
   sage: b = R(-70700); b
   2*5^2 + 4*5^3 + 5^4 + 2*5^5 + 4*5^7 + 4*5^8 + 4*5^9 + 4*5^10 + 4*5^11
    + 4*5^12 + 4*5^13 + 4*5^14 + 4*5^15 + 4*5^16 + 4*5^17 + 4*5^18
    + 4*5^19 + 4*5^20 + 4*5^21 + O(5^22)
   ```

   print_pos controls whether negatives can be used in the coefficients of powers of p.

   ```
   sage: S = Zp(5, print_mode='series', print_pos=False); a = S(70700); a
   -2*5^2 + 5^3 - 2*5^4 - 2*5^5 + 5^7 + O(5^22)
   sage: b = S(-70700); b
   2*5^2 - 5^3 + 2*5^4 + 2*5^5 - 5^7 + O(5^22)
   ```

   print_max_terms limits the number of terms that appear.

   ```
   sage: T = Zp(5, print_mode='series', print_max_terms=4); b = R(-70700); b
   2*5^2 + 4*5^3 + 5^4 + 2*5^5 + ... + O(5^22)
   ```

   names affects how the prime is printed.

   ```
   sage: U.<p> = Zp(5); p
   p + O(p^21)
   ```

show_prec determines how the precision is printed. It can be either 'none' (or equivalently `False`), 'bigoh' (or equivalently `True`). The default is `False` for the `'floating-point'` and `'fixed-mod'` types and `True` for all other types.

```
sage: Zp(5, show_prec=False)(6)
1 + 5
```

print_sep and *print_alphabet* have no effect.

Note that print options affect equality:

```
sage: R == S, R == T, R == U, S == T, S == U, T == U
(False, False, False, False, False, False)
```

2. **val-unit**: elements are displayed as $p^k u$:

```
sage: R = Zp(5, print_mode='val-unit'); a = R(70700); a
5^2 * 2828 + O(5^22)
sage: b = R(-707*5); b
5 * 95367431639918 + O(5^21)
```

print_pos controls whether to use a balanced representation or not.

```
sage: S = Zp(5, print_mode='val-unit', print_pos=False); b = S(-70700); b
5^2 * (-2828) + O(5^22)
```

names affects how the prime is printed.

```
sage: T = Zp(5, print_mode='val-unit', names='pi'); a = T(70700); a
pi^2 * 2828 + O(pi^22)
```

show_prec determines how the precision is printed. It can be either 'none' (or equivalently `False`), 'bigoh' (or equivalently `True`). The default is `False` for the `'floating-point'` and `'fixed-mod'` types and `True` for all other types.

```
sage: Zp(5, print_mode='val-unit', show_prec=False)(30)
5 * 6
```

print_max_terms, *print_sep* and *print_alphabet* have no effect.

Equality again depends on the printing options:

```
sage: R == S, R == T, S == T
(False, False, False)
```

3. **terse**: elements are displayed as an integer in base 10:

```
sage: R = Zp(5, print_mode='terse'); a = R(70700); a
70700 + O(5^22)
sage: b = R(-70700); b
2384185790944925 + O(5^22)
```

print_pos controls whether to use a balanced representation or not.

```
sage: S = Zp(5, print_mode='terse', print_pos=False); b = S(-70700); b
-70700 + O(5^22)
```

name affects how the name is printed. Note that this interacts with the choice of shorter string for denominators.

```
sage: T.<unif> = Zp(5, print_mode='terse'); c = T(-707); c
95367431639918 + O(unif^20)
```

show_prec determines how the precision is printed. It can be either 'none' (or equivalently `False`), 'bigoh' (or equivalently `True`). The default is `False` for the `'floating-point'` and `'fixed-mod'` types and `True` for all other types.

```
sage: Zp(5, print_mode='terse', show_prec=False)(30)
30
```

print_max_terms, *print_sep* and *print_alphabet* have no effect.

Equality depends on printing options:

```
sage: R == S, R == T, S == T
(False, False, False)
```

4. **digits**: elements are displayed as a string of base p digits

   Restriction: you can only use the digits printing mode for small primes. Namely, p must be less than the length of the alphabet tuple (default alphabet has length 62).

```
sage: R = Zp(5, print_mode='digits'); a = R(70700); repr(a)
'...4230300'
sage: b = R(-70700); repr(b)
'...4444444444444440214200'
```

   Note that it's not possible to read off the precision from the representation in this mode.

   print_max_terms limits the number of digits that are printed.

```
sage: S = Zp(5, print_max_terms=4); S(-70700)
2*5^2 + 4*5^3 + 5^4 + 2*5^5 + ... + O(5^22)
```

   print_alphabet controls the symbols used to substitute for digits greater than 9. Defaults to ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'):

```
sage: T = Zp(5, print_mode='digits', print_alphabet=('1','2','3','4','5'))
sage: repr(T(-70700))
'...5555555555555551325311'
```

   show_prec determines how the precision is printed. It can be either `'none'` (or equivalently `False`), `'dots'` (or equivalently `True`) or `'bigoh'`. The default is `False` for the `'floating-point'` and `'fixed-mod'` types and `True` for all other types.

```
sage: repr(Zp(5, 2, print_mode='digits', show_prec=True)(6))
'...11'
sage: repr(Zp(5, 2, print_mode='digits', show_prec='bigoh')(6))
'11 + O(5^2)'
```

print_pos, *name* and *print_sep* have no effect.

Equality depends on printing options:

```
sage: R == S, R == T, S == T
(False, False, False)
```

5. **bars**: elements are displayed as a string of base p digits with separators:

```
sage: R = Zp(5, print_mode='bars'); a = R(70700); repr(a)
'...4|2|3|0|3|0|0'
sage: b = R(-70700); repr(b)
'...4|4|4|4|4|4|4|4|4|4|4|4|4|4|4|0|2|1|4|2|0|0'
```

Again, note that it's not possible to read off the precision from the representation in this mode.

print_pos controls whether the digits can be negative.

```
sage: S = Zp(5, print_mode='bars', print_pos=False); b = S(-70700); repr(b)
'...-1|0|2|2|-1|2|0|0'
```

print_max_terms limits the number of digits that are printed.

```
sage: T = Zp(5, print_max_terms=4); T(-70700)
2*5^2 + 4*5^3 + 5^4 + 2*5^5 + ... + O(5^22)
```

print_sep controls the separation character.

```
sage: U = Zp(5, print_mode='bars', print_sep=']['); a = U(70700); repr(a)
'...4][2][3][0][3][0][0'
```

show_prec determines how the precision is printed. It can be either 'none' (or equivalently `False`), 'dots' (or equivalently `True`) or 'bigoh'. The default is `False` for the `'floating-point'` and `'fixed-mod'` types and `True` for all other types.

```
sage: repr(Zp(5, 2, print_mode='bars', show_prec=True)(6))
'...1|1'
sage: repr(Zp(5, 2, print_mode='bars', show_prec=False)(6))
'1|1'
```

name and *print_alphabet* have no effect.

Equality depends on printing options:

```
sage: R == S, R == T, R == U, S == T, S == U, T == U
(False, False, False, False, False, False)
```

EXAMPLES:

We allow non-prime p, but only if `check=False`. Note that some features will not work.

```
sage: K = Zp(15, check=False); a = K(999); a
9 + 6*15 + 4*15^2 + O(15^20)
```

We create rings with various parameters:

```
sage: Zp(7)
7-adic Ring with capped relative precision 20
sage: Zp(9)
Traceback (most recent call last):
...
ValueError: p must be prime
sage: Zp(17, 5)
17-adic Ring with capped relative precision 5
sage: Zp(17, 5)(-1)
16 + 16*17 + 16*17^2 + 16*17^3 + 16*17^4 + O(17^5)
```

It works even with a fairly huge cap:

```
sage: Zp(next_prime(10^50), 100000)
100000000000000000000000000000000000000000000000151-adic Ring
 with capped relative precision 100000
```

We create each type of ring:

```
sage: Zp(7, 20, 'capped-rel')
7-adic Ring with capped relative precision 20
sage: Zp(7, 20, 'fixed-mod')
7-adic Ring of fixed modulus 7^20
sage: Zp(7, 20, 'capped-abs')
7-adic Ring with capped absolute precision 20
```

We create a capped relative ring with each print mode:

```
sage: k = Zp(7, 8, print_mode='series'); k
7-adic Ring with capped relative precision 8
sage: k(7*(19))
5*7 + 2*7^2 + O(7^9)
sage: k(7*(-19))
2*7 + 4*7^2 + 6*7^3 + 6*7^4 + 6*7^5 + 6*7^6 + 6*7^7 + 6*7^8 + O(7^9)
```

```
sage: k = Zp(7, print_mode='val-unit'); k
7-adic Ring with capped relative precision 20
sage: k(7*(19))
7 * 19 + O(7^21)
sage: k(7*(-19))
7 * 79792266297611982 + O(7^21)
```

```
sage: k = Zp(7, print_mode='terse'); k
7-adic Ring with capped relative precision 20
sage: k(7*(19))
133 + O(7^21)
sage: k(7*(-19))
558545864083283874 + O(7^21)
```

Note that p-adic rings are cached (via weak references):

```
sage: a = Zp(7); b = Zp(7)
sage: a is b
True
```

We create some elements in various rings:

```
sage: R = Zp(5); a = R(4); a
4 + O(5^20)
sage: S = Zp(5, 10, type = 'capped-abs'); b = S(2); b
2 + O(5^10)
sage: a + b
1 + 5 + O(5^10)
```

create_key(*p*, *prec=None*, *type='capped-rel'*, *print_mode=None*, *names=None*, *ram_name=None*, *print_pos=None*, *print_sep=None*, *print_alphabet=None*, *print_max_terms=None*, *show_prec=None*, *check=True*, *label=None*)

Creates a key from input parameters for `Zp`.

See the documentation for `Zp()` for more information.

create_object(*version*, *key*)

Creates an object using a given key.

See the documentation for `Zp()` for more information.

sage.rings.padics.factory.**Zq**(*q*, *prec=None*, *type='capped-rel'*, *modulus=None*, *names=None*, *print_mode=None*, *ram_name=None*, *res_name=None*, *print_pos=None*, *print_sep=None*, *print_max_ram_terms=None*, *print_max_unram_terms=None*, *print_max_terse_terms=None*, *show_prec=None*, *check=True*, *implementation='FLINT'*)

Given a prime power $q = p^n$, return the unique unramified extension of $\mathbf{Z}_p$ of degree n.

INPUT:

- `q` – integer, list or tuple: the prime power in $\mathbf{Q}_q$. Or a `Factorization` object, single element list `[(p, n)]` where `p` is a prime and `n` a positive integer, or the pair `(p, n)`.
- `prec` – integer (default: `20`) the precision cap of the field. Individual elements keep track of their own precision. See TYPES and PRECISION below.
- `type` – string (default: `'capped-rel'`) Valid types are `'capped-abs'`, `'capped-rel'`, `'fixed-mod'`, and `'floating-point'`. See TYPES and PRECISION below.
- `modulus` – polynomial (default None) A polynomial defining an unramified extension of $\mathbf{Z}_p$. See MODULUS below.
- `names` – string or tuple (`None` is only allowed when $q = p$). The name of the generator, reducing to a generator of the residue field.
- `print_mode` – string (default: `None`). Valid modes are `'series'`, `'val-unit'`, `'terse'`, and `'bars'`. See PRINTING below.
- `ram_name` – string (defaults to string representation of p if `None`). `ram_name` controls how the prime is printed. See PRINTING below.
- `res_name` – string (defaults to `None`, which corresponds to adding a `'0'` to the end of the name). Controls how elements of the residue field print.

- `print_pos` – bool (default `None`) Whether to only use positive integers in the representations of elements. See PRINTING below.
- `print_sep` – string (default `None`) The separator character used in the `'bars'` mode. See PRINTING below.
- `print_max_ram_terms` – integer (default `None`) The maximum number of powers of p shown. See PRINTING below.
- `print_max_unram_terms` – integer (default `None`) The maximum number of entries shown in a coefficient of p. See PRINTING below.
- `print_max_terse_terms` – integer (default `None`) The maximum number of terms in the polynomial representation of an element (using `'terse'`). See PRINTING below.
- `show_prec` – bool (default `None`) Whether to show the precision for elements. See PRINTING below.
- `check` – bool (default `True`) whether to check inputs.
- `implementation` – string (default `'FLINT'`) which implementation to use. `'NTL'` is the other option.

OUTPUT:

The corresponding unramified p-adic ring.

TYPES AND PRECISION:

There are two types of precision for a p-adic element. The first is relative precision (default), which gives the number of known p-adic digits:

```
sage: R.<a> = Zq(25, 20, 'capped-rel', print_mode='series'); b = 25*a; b        #␣
↪needs sage.libs.ntl
a*5^2 + O(5^22)
sage: b.precision_relative()                                                      #␣
↪needs sage.libs.ntl
20
```

The second type of precision is absolute precision, which gives the power of p that this element is defined modulo:

```
sage: b.precision_absolute()                                                      #␣
↪needs sage.libs.ntl
22
```

There are many types of p-adic rings: capped relative rings (`type='capped-rel'`), capped absolute rings (`type='capped-abs'`), fixed modulus rings (`type='fixed-mod'`), and floating point rings (`type='floating-point'`).

In the capped relative case, the relative precision of an element is restricted to be at most a certain value, specified at the creation of the field. Individual elements also store their own precision, so the effect of various arithmetic operations on precision is tracked. When you cast an exact element into a capped relative field, it truncates it to the precision cap of the field.

```
sage: R.<a> = Zq(9, 5, 'capped-rel', print_mode='series'); b = (1+2*a)^4; b     #␣
↪needs sage.libs.ntl
2 + (2*a + 2)*3 + (2*a + 1)*3^2 + O(3^5)
sage: c = R(3249); c                                                              #␣
↪needs sage.libs.ntl
3^2 + 3^4 + 3^5 + 3^6 + O(3^7)
sage: b + c                                                                       #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.libs.ntl
2 + (2*a + 2)*3 + (2*a + 2)*3^2 + 3^4 + O(3^5)
```

One can invert non-units: the result is in the fraction field.

```
sage: d = ~(3*b+c); d                                                          #␣
↪needs sage.libs.ntl
2*3^-1 + (a + 1) + (a + 1)*3 + a*3^3 + O(3^4)
sage: d.parent()                                                               #␣
↪needs sage.libs.ntl
3-adic Unramified Extension Field in a defined by x^2 + 2*x + 2
```

The capped absolute case is the same as the capped relative case, except that the cap is on the absolute precision rather than the relative precision.

```
sage: # needs sage.libs.flint
sage: R.<a> = Zq(9, 5, 'capped-abs', print_mode='series'); b = 3*(1+2*a)^4; b
2*3 + (2*a + 2)*3^2 + (2*a + 1)*3^3 + O(3^5)
sage: c = R(3249); c                                                           #␣
↪needs sage.libs.ntl
3^2 + 3^4 + O(3^5)
sage: b*c                                                                      #␣
↪needs sage.libs.ntl
2*3^3 + (2*a + 2)*3^4 + O(3^5)
sage: b*c >> 1                                                                 #␣
↪needs sage.libs.ntl
2*3^2 + (2*a + 2)*3^3 + O(3^4)
```

The fixed modulus case is like the capped absolute, except that individual elements don't track their precision.

```
sage: # needs sage.libs.flint
sage: R.<a> = Zq(9, 5, 'fixed-mod', print_mode='series'); b = 3*(1+2*a)^4; b
2*3 + (2*a + 2)*3^2 + (2*a + 1)*3^3
sage: c = R(3249); c                                                           #␣
↪needs sage.libs.ntl
3^2 + 3^4
sage: b*c                                                                      #␣
↪needs sage.libs.ntl
2*3^3 + (2*a + 2)*3^4
sage: b*c >> 1                                                                 #␣
↪needs sage.libs.ntl
2*3^2 + (2*a + 2)*3^3
```

The floating point case is similar to the fixed modulus type in that elements do not trac their own precision. However, relative precision is truncated with each operation rather than absolute precision.

MODULUS:

The modulus needs to define an unramified extension of $\mathbf{Z}_p$: when it is reduced to a polynomial over $\mathbf{F}_p$ it should be irreducible.

The modulus can be given in a number of forms.

1. A **polynomial**.

The base ring can be $\mathbf{Z}$, $\mathbf{Q}$, $\mathbf{Z}_p$, $\mathbf{F}_p$, or anything that can be converted to $\mathbf{Z}_p$.

```
sage: # needs sage.libs.ntl
sage: P.<x> = ZZ[]
sage: R.<a> = Zq(27, modulus = x^3 + 2*x + 1); R.modulus()
(1 + O(3^20))*x^3 + O(3^20)*x^2 + (2 + O(3^20))*x + 1 + O(3^20)
sage: P.<x> = QQ[]
sage: S.<a> = Zq(27, modulus = x^3 + 2/7*x + 1)
sage: P.<x> = Zp(3)[]
sage: T.<a> = Zq(27, modulus = x^3 + 2*x + 1)
sage: P.<x> = Qp(3)[]
sage: U.<a> = Zq(27, modulus = x^3 + 2*x + 1)
sage: P.<x> = GF(3)[]
↪    # needs sage.rings.finite_rings
sage: V.<a> = Zq(27, modulus = x^3 + 2*x + 1)
```

Which form the modulus is given in has no effect on the unramified extension produced:

```
sage: R == S, R == T, T == U, U == V
↪    # needs sage.libs.ntl
(False, True, True, False)
```

unless the modulus is different, or the precision of the modulus differs. In the case of V, the modulus is only given to precision 1, so the resulting field has a precision cap of 1.

```
sage: # needs sage.libs.ntl
sage: V.precision_cap()
1
sage: U.precision_cap()
20
sage: P.<x> = Zp(3)[]
sage: modulus = x^3 + (2 + O(3^7))*x + (1 + O(3^10))
sage: modulus
(1 + O(3^20))*x^3 + (2 + O(3^7))*x + 1 + O(3^10)
sage: W.<a> = Zq(27, modulus = modulus); W.precision_cap()
7
```

2. The modulus can also be given as a **symbolic expression**.

```
sage: x = var('x')
↪# needs sage.symbolic
sage: X.<a> = Zq(27, modulus = x^3 + 2*x + 1); X.modulus()
↪# needs sage.symbolic
(1 + O(3^20))*x^3 + O(3^20)*x^2 + (2 + O(3^20))*x + 1 + O(3^20)
sage: X == R
↪# needs sage.libs.ntl sage.symbolic
True
```

By default, the polynomial chosen is the standard lift of the generator chosen for $\mathbf{F}_q$.

```
sage: GF(125, 'a').modulus()
↪    # needs sage.rings.finite_rings
x^3 + 3*x + 3
```

(continues on next page)

(continued from previous page)

```
sage: Y.<a> = Zq(125); Y.modulus()                                          ␣
↪    # needs sage.libs.ntl
(1 + O(5^20))*x^3 + O(5^20)*x^2 + (3 + O(5^20))*x + 3 + O(5^20)
```

However, you can choose another polynomial if desired (as long as the reduction to $\mathbf{F}_p[x]$ is irreducible).

```
sage: P.<x> = ZZ[]
sage: Z.<a> = Zq(125, modulus = x^3 + 3*x^2 + x + 1); Z.modulus()           ␣
↪    # needs sage.libs.ntl
(1 + O(5^20))*x^3 + (3 + O(5^20))*x^2 + (1 + O(5^20))*x + 1 + O(5^20)
sage: Y == Z                                                                ␣
↪    # needs sage.libs.ntl
False
```

PRINTING:

There are many different ways to print p-adic elements. The way elements of a given field print is controlled by options passed in at the creation of the field. There are four basic printing modes (`'series'`, `'val-unit'`, `'terse'` and `'bars'`; `'digits'` is not available), as well as various options that either hide some information in the print representation or sometimes make print representations more compact. Note that the printing options affect whether different p-adic fields are considered equal.

1. **series**: elements are displayed as series in p.

```
sage: # needs sage.libs.ntl
sage: R.<a> = Zq(9, 20, 'capped-rel', print_mode='series'); (1+2*a)^4
2 + (2*a + 2)*3 + (2*a + 1)*3^2 + O(3^20)
sage: -3*(1+2*a)^4
3 + a*3^2 + 3^3 + (2*a + 2)*3^4 + (2*a + 2)*3^5 + (2*a + 2)*3^6 + (2*a + 2)*3^7
  + (2*a + 2)*3^8 + (2*a + 2)*3^9 + (2*a + 2)*3^10 + (2*a + 2)*3^11 + (2*a +␣
↪2)*3^12
  + (2*a + 2)*3^13 + (2*a + 2)*3^14 + (2*a + 2)*3^15 + (2*a + 2)*3^16
  + (2*a + 2)*3^17 + (2*a + 2)*3^18 + (2*a + 2)*3^19 + (2*a + 2)*3^20 + O(3^21)
sage: b = ~(3*a+18); b
(a + 2)*3^-1 + 1 + 2*3 + (a + 1)*3^2 + 3^3 + 2*3^4 + (a + 1)*3^5 + 3^6 + 2*3^7
  + (a + 1)*3^8 + 3^9 + 2*3^10 + (a + 1)*3^11 + 3^12 + 2*3^13 + (a + 1)*3^14
  + 3^15 + 2*3^16 + (a + 1)*3^17 + 3^18 + O(3^19)
sage: b.parent() is R.fraction_field()
True
```

print_pos controls whether negatives can be used in the coefficients of powers of p.

```
sage: S.<b> = Zq(9, print_mode='series', print_pos=False); (1+2*b)^4        ␣
↪    # needs sage.libs.ntl
-1 - b*3 - 3^2 + (b + 1)*3^3 + O(3^20)
sage: -3*(1+2*b)^4                                                          ␣
↪    # needs sage.libs.ntl
3 + b*3^2 + 3^3 + (-b - 1)*3^4 + O(3^21)
```

ram_name controls how the prime is printed.

```
sage: T.<d> = Zq(9, print_mode='series', ram_name='p'); 3*(1+2*d)^4          # needs sage.libs.ntl
2*p + (2*d + 2)*p^2 + (2*d + 1)*p^3 + O(p^21)
```

print_max_ram_terms limits the number of powers of p that appear.

```
sage: U.<e> = Zq(9, print_mode='series', print_max_ram_terms=4); repr(-3*(1+2*e)^4)          # needs sage.libs.ntl
'3 + e*3^2 + 3^3 + (2*e + 2)*3^4 + ... + O(3^21)'
```

print_max_unram_terms limits the number of terms that appear in a coefficient of a power of p.

```
sage: # needs sage.libs.ntl
sage: V.<f> = Zq(128, prec = 8, print_mode='series'); repr((1+f)^9)
'(f^3 + 1) + (f^5 + f^4 + f^3 + f^2)*2 + (f^6 + f^5 + f^4 + f + 1)*2^2 + (f^5 + f^4 + f^2 + f + 1)*2^3 + (f^6 + f^5 + f^4 + f^3 + f^2 + f + 1)*2^4 + (f^5 + f^4)*2^5 + (f^6 + f^5 + f^4 + f^3 + f + 1)*2^6 + (f + 1)*2^7 + O(2^8)'
sage: V.<f> = Zq(128, prec = 8, print_mode='series', print_max_unram_terms = 3); repr((1+f)^9)
'(f^3 + 1) + (f^5 + f^4 + ... + f^2)*2 + (f^6 + f^5 + ... + 1)*2^2 + (f^5 + f^4 + ... + 1)*2^3 + (f^6 + f^5 + ... + 1)*2^4 + (f^5 + f^4)*2^5 + (f^6 + f^5 + ... + 1)*2^6 + (f + 1)*2^7 + O(2^8)'
sage: V.<f> = Zq(128, prec = 8, print_mode='series', print_max_unram_terms = 2); repr((1+f)^9)
'(f^3 + 1) + (f^5 + ... + f^2)*2 + (f^6 + ... + 1)*2^2 + (f^5 + ... + 1)*2^3 + (f^6 + ... + 1)*2^4 + (f^5 + f^4)*2^5 + (f^6 + ... + 1)*2^6 + (f + 1)*2^7 + O(2^8)'
sage: V.<f> = Zq(128, prec = 8, print_mode='series', print_max_unram_terms = 1); repr((1+f)^9)
'(f^3 + ...) + (f^5 + ...)*2 + (f^6 + ...)*2^2 + (f^5 + ...)*2^3 + (f^6 + ...)*2^4 + (f^5 + ...)*2^5 + (f^6 + ...)*2^6 + (f + ...)*2^7 + O(2^8)'
sage: V.<f> = Zq(128, prec = 8, print_mode='series', print_max_unram_terms = 0); repr((1+f)^9 - 1 - f^3)
'(...)*2 + (...)*2^2 + (...)*2^3 + (...)*2^4 + (...)*2^5 + (...)*2^6 + (...)*2^7 + O(2^8)'
```

show_prec determines how the precision is printed. It can be either `'none'` (or equivalently `False`), `'bigoh'` (or equivalently `True`). The default is `False` for the `'floating-point'` and `'fixed-mod'` types and `True` for all other types.

```
sage: U.<e> = Zq(9, 2, show_prec=False); repr(-3*(1+2*e)^4)          # needs sage.libs.ntl
'3 + e*3^2'
```

print_sep and *print_max_terse_terms* have no effect.

Note that print options affect equality:

```
sage: R == S, R == T, R == U, R == V, S == T, S == U, S == V, T == U, T == V, U == V          # needs sage.libs.ntl
(False, False, False, False, False, False, False, False, False, False)
```

2. **val-unit**: elements are displayed as $p^k u$:

```
sage: R.<a> = Zq(9, 7, print_mode='val-unit'); b = (1+3*a)^9 - 1; b
↪# needs sage.libs.ntl
3^3 * (15 + 64*a) + O(3^7)
sage: ~b
↪# needs sage.libs.ntl
3^-3 * (41 + a) + O(3)
```

print_pos controls whether to use a balanced representation or not.

```
sage: S.<a> = Zq(9, 7, print_mode='val-unit', print_pos=False); b = (1+3*a)^
↪9 - 1; b         # needs sage.libs.ntl
3^3 * (15 - 17*a) + O(3^7)
sage: ~b                                                                          ␣
↪                 # needs sage.libs.ntl
3^-3 * (-40 + a) + O(3)
```

ram_name affects how the prime is printed.

```
sage: # needs sage.libs.ntl
sage: A.<x> = Zp(next_prime(10^6), print_mode='val-unit')[]
sage: T.<a> = Zq(next_prime(10^6)^3, 4, print_mode='val-unit', ram_name='p',
....:          modulus=x^3+385831*x^2+106556*x+321036)
sage: b = next_prime(10^6)^2*(a^2 + a - 4)^4; b
p^2 * (87996187118837557387483 + 246348888344392418464080*a +␣
↪1353538653775332610349*a^2)
 + O(p^6)
sage: b * (a^2 + a - 4)^-4
p^2 * 1 + O(p^6)
```

print_max_terse_terms controls how many terms of the polynomial appear in the unit part.

```
sage: # needs sage.libs.ntl
sage: U.<a> = Zq(17^4, 6, print_mode='val-unit', print_max_terse_terms=3)
sage: b = 17*(a^3-a+14)^6; b
17 * (12131797 + 12076378*a + 10809706*a^2 + ...) + O(17^7)
```

show_prec determines how the precision is printed. It can be either `'none'` (or equivalently `False`), `'bigoh'` (or equivalently `True`). The default is `False` for the `'floating-point'` and `'fixed-mod'` types and `True` for all other types.

```
sage: U.<e> = Zq(9, 2, print_mode='val-unit', show_prec=False); repr(-
↪3*(1+2*e)^4)          # needs sage.libs.ntl
'3 * (1 + 3*e)'
```

print_sep, *print_max_ram_terms* and *print_max_unram_terms* have no effect.

Equality again depends on the printing options:

```
sage: R == S, R == T, R == U, S == T, S == U, T == U                             ␣
↪     # needs sage.libs.ntl
(False, False, False, False, False, False)
```

3. **terse**: elements are displayed as a polynomial of degree less than the degree of the extension.

```
sage: # needs sage.libs.ntl
sage: R.<a> = Zq(125, print_mode='terse')
sage: (a+5)^177
68210977979428 + 90313850704069*a + 73948093055069*a^2 + O(5^20)
sage: (a/5+1)^177
68210977979428/5^177 + 90313850704069/5^177*a + 73948093055069/5^177*a^2 + O(5^-
↪157)
```

Note that in this last computation, you get one fewer p-adic digit than one might expect. This is because R is capped absolute, and thus 5 is cast in with relative precision 19.

As of version 3.3, if coefficients of the polynomial are non-integral, they are always printed with an explicit power of p in the denominator.

```
sage: # needs sage.libs.ntl
sage: 5*a + a^2/25
5*a + 1/5^2*a^2 + O(5^18)
```

print_pos controls whether to use a balanced representation or not.

```
sage: # needs sage.libs.ntl
sage: (a-5)^6
22864 + 95367431627998*a + 8349*a^2 + O(5^20)
sage: S.<a> = Zq(125, print_mode='terse', print_pos=False); b = (a-5)^6; b
22864 - 12627*a + 8349*a^2 + O(5^20)
sage: (a - 1/5)^6
-20624/5^6 + 18369/5^5*a + 1353/5^3*a^2 + O(5^14)
```

ram_name affects how the prime is printed.

```
sage: T.<a> = Zq(125, print_mode='terse', ram_name='p'); (a - 1/5)^6              ␣
↪    # needs sage.libs.ntl
95367431620001/p^6 + 18369/p^5*a + 1353/p^3*a^2 + O(p^14)
```

print_max_terse_terms controls how many terms of the polynomial are shown.

```
sage: U.<a> = Zq(625, print_mode='terse', print_max_terse_terms=2); (a-1/5)^
↪6   # needs sage.libs.ntl
106251/5^6 + 49994/5^5*a + ... + O(5^14)
```

show_prec determines how the precision is printed. It can be either `'none'` (or equivalently `False`), `'bigoh'` (or equivalently `True`). The default is `False` for the `'floating-point'` and `'fixed-mod'` types and `True` for all other types.

```
sage: U.<e> = Zq(9, 2, print_mode='terse', show_prec=False); repr(-
↪3*(1+2*e)^4)             # needs sage.libs.ntl
'3 + 9*e'
```

print_sep, *print_max_ram_terms* and *print_max_unram_terms* have no effect.

Equality again depends on the printing options:

```
sage: R == S, R == T, R == U, S == T, S == U, T == U                              ␣
↪    # needs sage.libs.ntl
(False, False, False, False, False, False)
```

4. **digits**: This print mode is not available when the residue field is not prime. It might make sense to have a dictionary for small fields, but this isn't implemented.

5. **bars**: elements are displayed in a similar fashion to series, but more compactly.

```
sage: # needs sage.libs.ntl
sage: R.<a> = Zq(125); (a+5)^6
(4*a^2 + 3*a + 4) + (3*a^2 + 2*a)*5 + (a^2 + a + 1)*5^2 + (3*a + 2)*5^3
 + (3*a^2 + a + 3)*5^4 + (2*a^2 + 3*a + 2)*5^5 + O(5^20)
sage: R.<a> = Zq(125, print_mode='bars', prec=8); repr((a+5)^6)
'...[2, 3, 2]|[3, 1, 3]|[2, 3]|[1, 1, 1]|[0, 2, 3]|[4, 3, 4]'
sage: repr((a-5)^6)
'...[0, 4]|[1, 4]|[2, 0, 2]|[1, 4, 3]|[2, 3, 1]|[4, 4, 3]|[2, 4, 4]|[4, 3, 4]'
```

Note that it's not possible to read off the precision from the representation in this mode.

```
sage: # needs sage.libs.ntl
sage: b = a + 3; repr(b)
'...[3, 1]'
sage: c = a + R(3, 4); repr(c)
'...[3, 1]'
sage: b.precision_absolute()
8
sage: c.precision_absolute()
4
```

print_pos controls whether the digits can be negative.

```
sage: # needs sage.libs.ntl
sage: S.<a> = Zq(125, print_mode='bars', print_pos=False); repr((a-5)^6)
'...[1, -1, 1]|[2, 1, -2]|[2, 0, -2]|[-2, -1, 2]|[0, 0, -1]|[-2]|[-1, -2, -
↪1]'
sage: repr((a-1/5)^6)
'...[0, 1, 2]|[-1, 1, 1]|.|[-2, -1, -1]|[2, 2, 1]|[0, 0, -2]|[0, -1]|[0, -
↪1]|[1]'
```

print_max_ram_terms controls the maximum number of "digits" shown. Note that this puts a cap on the relative precision, not the absolute precision.

```
sage: # needs sage.libs.ntl
sage: T.<a> = Zq(125, print_max_ram_terms=3, print_pos=False); (a-5)^6
(-a^2 - 2*a - 1) - 2*5 - a^2*5^2 + ... + O(5^20)
sage: 5*(a-5)^6 + 50
(-a^2 - 2*a - 1)*5 - a^2*5^3 + (2*a^2 - a - 2)*5^4 + ... + O(5^21)
sage: (a-1/5)^6
5^-6 - a*5^-5 - a*5^-4 + ... + O(5^14)
```

print_sep controls the separating character (`'|'` by default).

```
sage: U.<a> = Zq(625, print_mode='bars', print_sep=''); b = (a+5)^6;␣
↪repr(b)     # needs sage.libs.ntl
'...[0, 1][4, 0, 2][3, 2, 2, 3][4, 2, 2, 4][0, 3][1, 1, 3][3, 1, 4, 1]'
```

print_max_unram_terms controls how many terms are shown in each `'digit'`:

```
sage: # needs sage.libs.ntl
sage: with local_print_mode(U, {'max_unram_terms': 3}): repr(b)
'...[0, 1][4,..., 0, 2][3,..., 2, 3][4,..., 2, 4][0, 3][1,..., 1, 3][3,...,␣
↪4, 1]'
sage: with local_print_mode(U, {'max_unram_terms': 2}): repr(b)
'...[0, 1][4,..., 2][3,..., 3][4,..., 4][0, 3][1,..., 3][3,..., 1]'
sage: with local_print_mode(U, {'max_unram_terms': 1}): repr(b)
'...[..., 1][..., 2][..., 3][..., 4][..., 3][..., 3][..., 1]'
sage: with local_print_mode(U, {'max_unram_terms':0}): repr(b-75*a)
'...[...][...][...][...][][...][...]'
```

show_prec determines how the precision is printed. It can be either `'none'` (or equivalently `False`), `'dots'` (or equivalently `True`) or `'bigoh'`. The default is `False` for the `'floating-point'` and `'fixed-mod'` types and `True` for all other types.

```
sage: U.<e> = Zq(9, 2, print_mode='bars', show_prec='bigoh'); repr(-
↪3*(1+2*e)^4)          # needs sage.libs.ntl
'[0, 1]|[1]|[] + O(3^3)'
```

ram_name and *print_max_terse_terms* have no effect.

Equality depends on printing options:

```
sage: R == S, R == T, R == U, S == T, S == U, T == U                        ␣
↪    # needs sage.libs.ntl
(False, False, False, False, False, False)
```

EXAMPLES:

Unlike for `Zp()`, you can't create `Zq(N)` when `N` is not a prime power.

However, you can use `check=False` to pass in a pair in order to not have to factor. If you do so, you need to use names explicitly rather than the `R.<a>` syntax.

```
sage: # needs sage.libs.ntl
sage: p = next_prime(2^123)
sage: k = Zp(p)
sage: R.<x> = k[]
sage: K = Zq([(p, 5)], modulus=x^5+x+4, names='a', ram_name='p',
....:          print_pos=False, check=False)
sage: K.0^5
(-a - 4) + O(p^20)
```

In tests on sage.math, the creation of `K` as above took an average of 1.58ms, while:

```
sage: K = Zq(p^5, modulus=x^5+x+4, names='a', ram_name='p',                  #␣
↪needs sage.libs.ntl
....:          print_pos=False, check=True)
```

took an average of 24.5ms. Of course, with smaller primes these savings disappear.

`sage.rings.padics.factory.`**`ZqCA`**`(`*q*, *prec=None*, **args*, ***kwds*`)`

A shortcut function to create capped absolute unramified p-adic rings.

See documentation for *`Zq()`* for a description of the input parameters.

EXAMPLES:

```
sage: R.<a> = ZqCA(25, 40); R                                              #␣
↪needs sage.libs.flint
5-adic Unramified Extension Ring in a defined by x^2 + 4*x + 2
```

sage.rings.padics.factory.**ZqCR**(*q*, *prec=None*, **args*, ***kwds*)

A shortcut function to create capped relative unramified p-adic rings.

Same functionality as *Zq()*. See documentation for *Zq()* for a description of the input parameters.

EXAMPLES:

```
sage: R.<a> = ZqCR(25, 40); R                                              #␣
↪needs sage.libs.ntl
5-adic Unramified Extension Ring in a defined by x^2 + 4*x + 2
```

sage.rings.padics.factory.**ZqFM**(*q*, *prec=None*, **args*, ***kwds*)

A shortcut function to create fixed modulus unramified p-adic rings.

See documentation for *Zq()* for a description of the input parameters.

EXAMPLES:

```
sage: R.<a> = ZqFM(25, 40); R                                              #␣
↪needs sage.libs.flint
5-adic Unramified Extension Ring in a defined by x^2 + 4*x + 2
```

sage.rings.padics.factory.**ZqFP**(*q*, *prec=None*, **args*, ***kwds*)

A shortcut function to create floating point unramified p-adic rings.

Same functionality as *Zq()*. See documentation for *Zq()* for a description of the input parameters.

EXAMPLES:

```
sage: R.<a> = ZqFP(25, 40); R                                              #␣
↪needs sage.libs.flint
5-adic Unramified Extension Ring in a defined by x^2 + 4*x + 2
```

sage.rings.padics.factory.**get_key_base**(*p*, *prec*, *type*, *print_mode*, *names*, *ram_name*, *print_pos*, *print_sep*, *print_alphabet*, *print_max_terms*, *show_prec*, *check*, *valid_types*, *label=None*)

This implements `create_key` for `Zp` and `Qp`: moving it here prevents code duplication.

It fills in unspecified values and checks for contradictions in the input. It also standardizes irrelevant options so that duplicate parents are not created.

EXAMPLES:

```
sage: from sage.rings.padics.factory import get_key_base
sage: get_key_base(11, 5, 'capped-rel', None, None, None, None, ':', None, None,␣
↪False, True, ['capped-rel'])
(11, 5, 'capped-rel', 'series', '11', True, '|', (), -1, 'none', None)
sage: get_key_base(12, 5, 'capped-rel', 'digits', None, None, None, None, None,␣
↪None, True, False, ['capped-rel'])
(12,
 5,
 'capped-rel',
```

(continues on next page)

(continued from previous page)

```
'digits',
'12',
True,
'|',
('0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B'),
-1,
'dots',
None)
```

sage.rings.padics.factory.**is_eisenstein**(*poly*)

Return `True` iff this monic polynomial is Eisenstein.

A polynomial is Eisenstein if it is monic, the constant term has valuation 1 and all other terms have positive valuation.

EXAMPLES:

```
sage: # needs sage.libs.ntl
sage: R = Zp(5)
sage: S.<x> = R[]
sage: from sage.rings.padics.factory import is_eisenstein
sage: f = x^4 - 75*x + 15
sage: is_eisenstein(f)
True
sage: g = x^4 + 75
sage: is_eisenstein(g)
False
sage: h = x^7 + 27*x -15
sage: is_eisenstein(h)
False
```

sage.rings.padics.factory.**is_unramified**(*poly*)

Return `True` iff this monic polynomial is unramified.

A polynomial is unramified if its reduction modulo the maximal ideal is irreducible.

EXAMPLES:

```
sage: # needs sage.libs.ntl
sage: R = Zp(5)
sage: S.<x> = R[]
sage: from sage.rings.padics.factory import is_unramified
sage: f = x^4 + 14*x + 9
sage: is_unramified(f)
True
sage: g = x^6 + 17*x + 6
sage: is_unramified(g)
False
```

sage.rings.padics.factory.**krasner_check**(*poly*, *prec*)

Return `True` iff `poly` determines a unique isomorphism class of extensions at precision `prec`.

Currently just returns `True` (thus allowing extensions that are not defined to high enough precision in order to specify them up to isomorphism). This will change in the future.

EXAMPLES:

```
sage: from sage.rings.padics.factory import krasner_check
sage: krasner_check(1,2)  # this is a stupid example.
True
```

class sage.rings.padics.factory.**pAdicExtension_class**

Bases: UniqueFactory

A class for creating extensions of p-adic rings and fields.

EXAMPLES:

```
sage: R = Zp(5,3)
sage: S.<x> = ZZ[]
sage: W.<w> = pAdicExtension(R, x^4 - 15); W                                  #␣
↪needs sage.libs.ntl
5-adic Eisenstein Extension Ring in w defined by x^4 - 15
sage: W.precision_cap()                                                       #␣
↪needs sage.libs.ntl
12
```

create_key_and_extra_args(*base*, *modulus*, *prec=None*, *print_mode=None*, *names=None*, *var_name=None*, *res_name=None*, *unram_name=None*, *ram_name=None*, *print_pos=None*, *print_sep=None*, *print_alphabet=None*, *print_max_ram_terms=None*, *print_max_unram_terms=None*, *print_max_terse_terms=None*, *show_prec=None*, *check=True*, *unram=False*, *implementation='FLINT'*)

Creates a key from input parameters for `pAdicExtension`.

See the documentation for `Qq()` for more information.

create_object(*version*, *key*, *approx_modulus=None*, *shift_seed=None*)

Creates an object using a given key.

See the documentation for `pAdicExtension` for more information.

sage.rings.padics.factory.**split**(*poly*, *prec*)

Given a polynomial `poly` and a desired precision `prec`, computes `upoly` and epoly so that the extension defined by `poly` is isomorphic to the extension defined by first taking an extension by the unramified polynomial `upoly`, and then an extension by the Eisenstein polynomial `epoly`.

We need better p-adic factoring in Sage before this function can be implemented.

EXAMPLES:

```
sage: k = Qp(13)
sage: x = polygen(k)                                                          #␣
↪needs sage.libs.ntl
sage: f = x^2 + 1                                                             #␣
↪needs sage.libs.ntl
sage: sage.rings.padics.factory.split(f, 10)                                  #␣
↪needs sage.libs.ntl sage.rings.real_double
Traceback (most recent call last):
...
NotImplementedError: Extensions by general polynomials not yet supported.
Please use an unramified or Eisenstein polynomial.
```

sage.rings.padics.factory.**truncate_to_prec**(*poly*, *R*, *absprec*)

Truncates the unused precision off of a polynomial.

EXAMPLES:

```
sage: R = Zp(5)
sage: S.<x> = R[]                                                     #␣
↪needs sage.libs.ntl
sage: from sage.rings.padics.factory import truncate_to_prec
sage: f = x^4 + (3+O(5^6))*x^3 + O(5^4)                               #␣
↪needs sage.libs.ntl
sage: truncate_to_prec(f, R, 5)                                       #␣
↪needs sage.libs.ntl
(1 + O(5^5))*x^4 + (3 + O(5^5))*x^3 + O(5^5)*x^2 + O(5^5)*x + O(5^4)
```

CHAPTER

THREE

LOCAL GENERIC

Superclass for p-adic and power series rings.

AUTHORS:

- David Roe

class sage.rings.padics.local_generic.**LocalGeneric**(*base*, *prec*, *names*, *element_class*, *category=None*)

Bases: CommutativeRing

Initialize self.

EXAMPLES:

```
sage: R = Zp(5)  # indirect doctest
sage: R.precision_cap()
20
```

In github issue #14084, the category framework has been implemented for p-adic rings:

```
sage: TestSuite(R).run()                                          # needs␣
↪sage.geometry.polyhedron
sage: K = Qp(7)
sage: TestSuite(K).run()                                          # needs␣
↪sage.geometry.polyhedron
```

absolute_degree()

Return the degree of this extension over the prime p-adic field/ring.

EXAMPLES:

```
sage: K.<a> = Qq(3^5)                                           #␣
↪needs sage.libs.ntl
sage: K.absolute_degree()                                       #␣
↪needs sage.libs.ntl
5

sage: R.<x> = QQ[]
sage: L.<pi> = Qp(3).extension(x^2 - 3)                         #␣
↪needs sage.libs.ntl
sage: L.absolute_degree()                                       #␣
↪needs sage.libs.ntl
2
```

absolute_e()

Return the absolute ramification index of this ring/field.

EXAMPLES:

```
sage: K.<a> = Qq(3^5)                                                   #␣
↪needs sage.libs.ntl
sage: K.absolute_e()                                                    #␣
↪needs sage.libs.ntl
1

sage: R.<x> = QQ[]
sage: L.<pi> = Qp(3).extension(x^2 - 3)                                 #␣
↪needs sage.libs.ntl
sage: L.absolute_e()                                                    #␣
↪needs sage.libs.ntl
2
```

absolute_f()

Return the degree of the residue field of this ring/field over its prime subfield.

EXAMPLES:

```
sage: K.<a> = Qq(3^5)                                                   #␣
↪needs sage.libs.ntl
sage: K.absolute_f()                                                    #␣
↪needs sage.libs.ntl
5

sage: R.<x> = QQ[]
sage: L.<pi> = Qp(3).extension(x^2 - 3)                                 #␣
↪needs sage.libs.ntl
sage: L.absolute_f()                                                    #␣
↪needs sage.libs.ntl
1
```

absolute_inertia_degree()

Return the degree of the residue field of this ring/field over its prime subfield.

EXAMPLES:

```
sage: K.<a> = Qq(3^5)                                                   #␣
↪needs sage.libs.ntl
sage: K.absolute_inertia_degree()                                       #␣
↪needs sage.libs.ntl
5

sage: R.<x> = QQ[]
sage: L.<pi> = Qp(3).extension(x^2 - 3)                                 #␣
↪needs sage.libs.ntl
sage: L.absolute_inertia_degree()                                       #␣
↪needs sage.libs.ntl
1
```

absolute_ramification_index()

Return the absolute ramification index of this ring/field.

EXAMPLES:

```
sage: K.<a> = Qq(3^5)                                                        #␣
↪needs sage.libs.ntl
sage: K.absolute_ramification_index()                                        #␣
↪needs sage.libs.ntl
1

sage: R.<x> = QQ[]
sage: L.<pi> = Qp(3).extension(x^2 - 3)                                      #␣
↪needs sage.libs.ntl
sage: L.absolute_ramification_index()                                        #␣
↪needs sage.libs.ntl
2
```

change(***kwds*)

Return a new ring with changed attributes.

INPUT:

The following arguments are applied to every ring in the tower:

- `type` – string, the precision type
- **p – the prime of the ground ring. Defining polynomials**
  will be converted to the new base rings.
- `print_mode` – string
- `print_pos` – bool
- `print_sep` – string
- `print_alphabet` – dict
- `show_prec` – bool
- `check` – bool
- `label` – string (only for lattice precision)

The following arguments are only applied to the top ring in the tower:

- `var_name` – string
- `res_name` – string
- `unram_name` – string
- `ram_name` – string
- `names` – string
- `modulus` – polynomial

The following arguments have special behavior:

- `prec` – integer. If the precision is increased on an extension ring, the precision on the base is increased as necessary (respecting ramification). If the precision is decreased, the precision of the base is unchanged.

- `field` – bool. If `True`, switch to a tower of fields via the fraction field. If False, switch to a tower of rings of integers.
- `q` – prime power. Replace the initial unramified extension of $\mathbf{Q}_p$ or $\mathbf{Z}_p$ with an unramified extension of residue cardinality q. If the initial extension is ramified, add in an unramified extension.
- `base` – ring or field. Use a specific base ring instead of recursively calling *change()* down the tower.

See the *constructors* for more details on the meaning of these arguments.

EXAMPLES:

We can use this method to change the precision:

```
sage: Zp(5).change(prec=40)
5-adic Ring with capped relative precision 40
```

or the precision type:

```
sage: Zp(5).change(type="capped-abs")
5-adic Ring with capped absolute precision 20
```

or even the prime:

```
sage: ZpCA(3).change(p=17)
17-adic Ring with capped absolute precision 20
```

You can switch between the ring of integers and its fraction field:

```
sage: ZpCA(3).change(field=True)
3-adic Field with capped relative precision 20
```

You can also change print modes:

```
sage: R = Zp(5).change(prec=5, print_mode='digits')
sage: repr(~R(17))
'...13403'
```

Changing print mode to 'digits' works for Eisenstein extensions:

```
sage: # needs sage.libs.ntl
sage: S.<x> = ZZ[]
sage: W.<w> = Zp(3).extension(x^4 + 9*x^2 + 3*x - 3)
sage: W.print_mode()
'series'
sage: W.change(print_mode='digits').print_mode()
'digits'
```

You can change extensions:

```
sage: # needs sage.libs.flint
sage: K.<a> = QqFP(125, prec=4)
sage: K.change(q=64)
2-adic Unramified Extension Field in a defined by x^6 + x^4 + x^3 + x + 1
sage: R.<x> = QQ[]
sage: K.change(modulus = x^2 - x + 2, print_pos=False)
5-adic Unramified Extension Field in a defined by x^2 - x + 2
```

and variable names:

```
sage: K.change(names='b')                                              #␣
↪needs sage.libs.flint
5-adic Unramified Extension Field in b defined by x^3 + 3*x + 3
```

and precision:

```
sage: # needs sage.libs.flint
sage: Kup = K.change(prec=8); Kup
5-adic Unramified Extension Field in a defined by x^3 + 3*x + 3
sage: Kup.precision_cap()
8
sage: Kup.base_ring()
5-adic Field with floating precision 8
```

If you decrease the precision, the precision of the base stays the same:

```
sage: # needs sage.libs.flint
sage: Kdown = K.change(prec=2); Kdown
5-adic Unramified Extension Field in a defined by x^3 + 3*x + 3
sage: Kdown.precision_cap()
2
sage: Kdown.base_ring()
5-adic Field with floating precision 4
```

Changing the prime works for extensions:

```
sage: # needs sage.libs.ntl
sage: x = polygen(ZZ)
sage: R.<a> = Zp(5).extension(x^2 + 2)
sage: S = R.change(p=7)
sage: S.defining_polynomial(exact=True)
x^2 + 2
sage: A.<y> = Zp(5)[]
sage: R.<a> = Zp(5).extension(y^2 + 2)
sage: S = R.change(p=7)
sage: S.defining_polynomial(exact=True)
y^2 + 2
```

```
sage: # needs sage.libs.ntl
sage: R.<a> = Zq(5^3)
sage: S = R.change(prec=50)
sage: S.defining_polynomial(exact=True)
x^3 + 3*x + 3
```

Changing label for lattice precision (the precision lattice is not copied):

```
sage: R = ZpLC(37, (8,11))
sage: S = R.change(label = "change"); S
37-adic Ring with lattice-cap precision (label: change)
sage: S.change(label = "new")
37-adic Ring with lattice-cap precision (label: new)
```

defining_polynomial(*var='x'*, *exact=False*)

Return the defining polynomial of this local ring

INPUT:

- `var` – string (default: `'x'`), the name of the variable
- `exact` – a boolean (default: `False`), whether to return the underlying exact defining polynomial rather than the one with coefficients in the base ring.

OUTPUT:

The defining polynomial of this ring as an extension over its ground ring

EXAMPLES:

```
sage: R = Zp(3, 3, 'fixed-mod')

sage: R.defining_polynomial().parent()
Univariate Polynomial Ring in x over 3-adic Ring of fixed modulus 3^3
sage: R.defining_polynomial('foo')
foo

sage: R.defining_polynomial(exact=True).parent()
Univariate Polynomial Ring in x over Integer Ring
```

degree()

Return the degree of this extension.

Raise an error if the base ring/field is itself an extension.

EXAMPLES:

```
sage: K.<a> = Qq(3^5)                                                   #␣
↪needs sage.libs.ntl
sage: K.degree()                                                        #␣
↪needs sage.libs.ntl
5

sage: R.<x> = QQ[]
sage: L.<pi> = Qp(3).extension(x^2 - 3)                                 #␣
↪needs sage.libs.ntl
sage: L.degree()                                                        #␣
↪needs sage.libs.ntl
2
```

e()

Return the ramification index of this extension.

Raise an error if the base ring/field is itself an extension.

EXAMPLES:

```
sage: K.<a> = Qq(3^5)                                                   #␣
↪needs sage.libs.ntl
sage: K.e()                                                             #␣
↪needs sage.libs.ntl
1
```

(continues on next page)

(continued from previous page)

```
sage: R.<x> = QQ[]
sage: L.<pi> = Qp(3).extension(x^2 - 3)                                       #␣
↪needs sage.libs.ntl
sage: L.e()                                                                     #␣
↪needs sage.libs.ntl
2
```

ext(**args*, ***kwds*)

Construct an extension of `self`. See `extension()` for more details.

EXAMPLES:

```
sage: A = Zp(7,10)
sage: S.<x> = A[]                                                               #␣
↪needs sage.libs.ntl
sage: B.<t> = A.ext(x^2 + 7)                                                    #␣
↪needs sage.libs.ntl
sage: B.uniformiser()                                                           #␣
↪needs sage.libs.ntl
t + O(t^21)
```

f()

Return the degree of the residual extension.

Raise an error if the base ring/field is itself an extension.

EXAMPLES:

```
sage: K.<a> = Qq(3^5)                                                           #␣
↪needs sage.libs.ntl
sage: K.f()                                                                     #␣
↪needs sage.libs.ntl
5

sage: R.<x> = QQ[]
sage: L.<pi> = Qp(3).extension(x^2 - 3)                                       #␣
↪needs sage.libs.ntl
sage: L.f()                                                                     #␣
↪needs sage.libs.ntl
1
```

ground_ring()

Return `self`.

Will be overridden by extensions.

INPUT:

- `self` – a local ring

OUTPUT:

The ground ring of `self`, i.e., itself.

EXAMPLES:

```
sage: R = Zp(3, 5, 'fixed-mod')
sage: S = Zp(3, 4, 'fixed-mod')
sage: R.ground_ring() is R
True
sage: S.ground_ring() is R
False
```

ground_ring_of_tower()

Return `self`.

Will be overridden by extensions.

INPUT:

- `self` – a p-adic ring

OUTPUT:

The ground ring of the tower for `self`, i.e., itself.

EXAMPLES:

```
sage: R = Zp(5)
sage: R.ground_ring_of_tower()
5-adic Ring with capped relative precision 20
```

inertia_degree()

Return the degree of the residual extension.

Raise an error if the base ring/field is itself an extension.

EXAMPLES:

```
sage: K.<a> = Qq(3^5)                                                   #␣
↪needs sage.libs.ntl
sage: K.inertia_degree()                                                #␣
↪needs sage.libs.ntl
5

sage: R.<x> = QQ[]
sage: L.<pi> = Qp(3).extension(x^2 - 3)                                 #␣
↪needs sage.libs.ntl
sage: L.inertia_degree()                                                #␣
↪needs sage.libs.ntl
1
```

inertia_subring()

Return the inertia subring, i.e. `self`.

INPUT:

- `self` – a local ring

OUTPUT:

- the inertia subring of self, i.e., itself

EXAMPLES:

```
sage: R = Zp(5)
sage: R.inertia_subring()
5-adic Ring with capped relative precision 20
```

is_capped_absolute()

Return whether this p-adic ring bounds precision in a capped absolute fashion.

The absolute precision of an element is the power of p modulo which that element is defined. In a capped absolute ring, the absolute precision of elements are bounded by a constant depending on the ring.

EXAMPLES:

```
sage: R = ZpCA(5, 15)
sage: R.is_capped_absolute()
True
sage: R(5^7)
5^7 + O(5^15)
sage: S = Zp(5, 15)
sage: S.is_capped_absolute()
False
sage: S(5^7)
5^7 + O(5^22)
```

is_capped_relative()

Return whether this p-adic ring bounds precision in a capped relative fashion.

The relative precision of an element is the power of p modulo which the unit part of that element is defined. In a capped relative ring, the relative precision of elements are bounded by a constant depending on the ring.

EXAMPLES:

```
sage: R = ZpCA(5, 15)
sage: R.is_capped_relative()
False
sage: R(5^7)
5^7 + O(5^15)
sage: S = Zp(5, 15)
sage: S.is_capped_relative()
True
sage: S(5^7)
5^7 + O(5^22)
```

is_exact()

Return whether this p-adic ring is exact, i.e. `False`.

EXAMPLES:

```
sage: R = Zp(5, 3, 'fixed-mod'); R.is_exact()
False
```

is_fixed_mod()

Return whether this p-adic ring bounds precision in a fixed modulus fashion.

The absolute precision of an element is the power of p modulo which that element is defined. In a fixed modulus ring, the absolute precision of every element is defined to be the precision cap of the parent. This means that some operations, such as division by p, don't return a well defined answer.

EXAMPLES:

```
sage: R = ZpFM(5,15)
sage: R.is_fixed_mod()
True
sage: R(5^7,absprec=9)
5^7
sage: S = ZpCA(5, 15)
sage: S.is_fixed_mod()
False
sage: S(5^7,absprec=9)
5^7 + O(5^9)
```

is_floating_point()

Return whether this p-adic ring bounds precision in a floating point fashion.

The relative precision of an element is the power of p modulo which the unit part of that element is defined. In a floating point ring, elements do not store precision, but arithmetic operations truncate to a relative precision depending on the ring.

EXAMPLES:

```
sage: R = ZpCR(5, 15)
sage: R.is_floating_point()
False
sage: R(5^7)
5^7 + O(5^22)
sage: S = ZpFP(5, 15)
sage: S.is_floating_point()
True
sage: S(5^7)
5^7
```

is_lattice_prec()

Return whether this p-adic ring bounds precision using a lattice model.

In lattice precision, relationships between elements are stored in a precision object of the parent, which allows for optimal precision tracking at the cost of increased memory usage and runtime.

EXAMPLES:

```
sage: R = ZpCR(5, 15)
sage: R.is_lattice_prec()
False
sage: x = R(25, 8)
sage: x - x
O(5^8)
sage: S = ZpLC(5, 15)
doctest:...: FutureWarning: This class/method/function is marked as␣
↪experimental. It, its functionality or its interface might change without a␣
↪formal deprecation.
See https://github.com/sagemath/sage/issues/23505 for details.
sage: S.is_lattice_prec()
True
```

(continues on next page)

(continued from previous page)

```
sage: x = S(25, 8)
sage: x - x
O(5^30)
```

is_relaxed()

Return whether this p-adic ring bounds precision in a relaxed fashion.

In a relaxed ring, elements have mechanisms for computing themselves to greater precision.

EXAMPLES:

```
sage: R = Zp(5)
sage: R.is_relaxed()
False
```

maximal_unramified_subextension()

Return the maximal unramified subextension.

INPUT:

- `self` – a local ring

OUTPUT:

- the maximal unramified subextension of `self`

EXAMPLES:

```
sage: R = Zp(5)
sage: R.maximal_unramified_subextension()
5-adic Ring with capped relative precision 20
```

precision_cap()

Return the precision cap for this ring.

EXAMPLES:

```
sage: R = Zp(3, 10,'fixed-mod'); R.precision_cap()
10
sage: R = Zp(3, 10,'capped-rel'); R.precision_cap()
10
sage: R = Zp(3, 10,'capped-abs'); R.precision_cap()
10
```

Note: This will have different meanings depending on the type of local ring. For fixed modulus rings, all elements are considered modulo `self.prime()^self.precision_cap()`. For rings with an absolute cap (i.e. the class `pAdicRingCappedAbsolute`), each element has a precision that is tracked and is bounded above by `self.precision_cap()`. Rings with relative caps (e.g. the class `pAdicRingCappedRelative`) are the same except that the precision is the precision of the unit part of each element.

ramification_index()

Return the ramification index of this extension.

Raise an error if the base ring/field is itself an extension.

EXAMPLES:

```
sage: K.<a> = Qq(3^5)                                                   #␣
↪needs sage.libs.ntl
sage: K.ramification_index()                                            #␣
↪needs sage.libs.ntl
1

sage: R.<x> = QQ[]
sage: L.<pi> = Qp(3).extension(x^2 - 3)                                 #␣
↪needs sage.libs.ntl
sage: L.ramification_index()                                            #␣
↪needs sage.libs.ntl
2
```

relative_degree()

Return the degree of this extension over its base field/ring.

EXAMPLES:

```
sage: K.<a> = Qq(3^5)                                                   #␣
↪needs sage.libs.ntl
sage: K.relative_degree()                                               #␣
↪needs sage.libs.ntl
5

sage: R.<x> = QQ[]
sage: L.<pi> = Qp(3).extension(x^2 - 3)                                 #␣
↪needs sage.libs.ntl
sage: L.relative_degree()                                               #␣
↪needs sage.libs.ntl
2
```

relative_e()

Return the ramification index of this extension over its base ring/field.

EXAMPLES:

```
sage: K.<a> = Qq(3^5)                                                   #␣
↪needs sage.libs.ntl
sage: K.relative_e()                                                    #␣
↪needs sage.libs.ntl
1

sage: R.<x> = QQ[]
sage: L.<pi> = Qp(3).extension(x^2 - 3)                                 #␣
↪needs sage.libs.ntl
sage: L.relative_e()                                                    #␣
↪needs sage.libs.ntl
2
```

relative_f()

Return the degree of the residual extension over its base ring/field.

EXAMPLES:

```
sage: K.<a> = Qq(3^5)                                                    #␣
↪needs sage.libs.ntl
sage: K.relative_f()                                                     #␣
↪needs sage.libs.ntl
5

sage: R.<x> = QQ[]
sage: L.<pi> = Qp(3).extension(x^2 - 3)                                  #␣
↪needs sage.libs.ntl
sage: L.relative_f()                                                     #␣
↪needs sage.libs.ntl
1
```

relative_inertia_degree()

Return the degree of the residual extension over its base ring/field.

EXAMPLES:

```
sage: K.<a> = Qq(3^5)                                                    #␣
↪needs sage.libs.ntl
sage: K.relative_inertia_degree()                                        #␣
↪needs sage.libs.ntl
5

sage: R.<x> = QQ[]
sage: L.<pi> = Qp(3).extension(x^2 - 3)                                  #␣
↪needs sage.libs.ntl
sage: L.relative_inertia_degree()                                        #␣
↪needs sage.libs.ntl
1
```

relative_ramification_index()

Return the ramification index of this extension over its base ring/field.

EXAMPLES:

```
sage: K.<a> = Qq(3^5)                                                    #␣
↪needs sage.libs.ntl
sage: K.relative_ramification_index()                                    #␣
↪needs sage.libs.ntl
1

sage: R.<x> = QQ[]
sage: L.<pi> = Qp(3).extension(x^2 - 3)                                  #␣
↪needs sage.libs.ntl
sage: L.relative_ramification_index()                                    #␣
↪needs sage.libs.ntl
2
```

residue_characteristic()

Return the characteristic of `self`'s residue field.

INPUT:

- `self` – a p-adic ring.

OUTPUT:

The characteristic of the residue field.

EXAMPLES:

```
sage: R = Zp(3, 5, 'capped-rel'); R.residue_characteristic()
3
```

uniformiser()

Return a uniformiser for `self`, ie a generator for the unique maximal ideal.

EXAMPLES:

```
sage: R = Zp(5)
sage: R.uniformiser()
5 + O(5^21)
sage: A = Zp(7,10)
sage: S.<x> = A[]                                                      #␣
↪needs sage.libs.ntl
sage: B.<t> = A.ext(x^2+7)                                             #␣
↪needs sage.libs.ntl
sage: B.uniformiser()                                                  #␣
↪needs sage.libs.ntl
t + O(t^21)
```

uniformiser_pow(n)

Return the $n`thpoweroftheuniformiserof``self`$ (as an element of `self`).

EXAMPLES:

```
sage: R = Zp(5)
sage: R.uniformiser_pow(5)
5^5 + O(5^25)
```

CHAPTER

FOUR

P-ADIC GENERIC

A generic superclass for all p-adic parents.

AUTHORS:

- David Roe
- Genya Zaytman: documentation
- David Harvey: doctests
- Julian Rueth (2013-03-16): test methods for basic arithmetic

class sage.rings.padics.padic_generic.**ResidueLiftingMap**

Bases: Morphism

Lifting map to a p-adic ring or field from its residue field or ring.

These maps must be created using the `_create_()` method in order to support categories correctly.

EXAMPLES:

```
sage: from sage.rings.padics.padic_generic import ResidueLiftingMap
sage: R.<a> = Zq(125); k = R.residue_field()                                #␣
↪needs sage.libs.ntl
sage: f = ResidueLiftingMap._create_(k, R); f                               #␣
↪needs sage.libs.ntl
Lifting morphism:
  From: Finite Field in a0 of size 5^3
  To:   5-adic Unramified Extension Ring in a defined by x^3 + 3*x + 3
```

class sage.rings.padics.padic_generic.**ResidueReductionMap**

Bases: Morphism

Reduction map from a p-adic ring or field to its residue field or ring.

These maps must be created using the `_create_()` method in order to support categories correctly.

EXAMPLES:

```
sage: from sage.rings.padics.padic_generic import ResidueReductionMap
sage: R.<a> = Zq(125); k = R.residue_field()                                #␣
↪needs sage.libs.ntl
sage: f = ResidueReductionMap._create_(R, k); f                             #␣
↪needs sage.libs.ntl
Reduction morphism:
  From: 5-adic Unramified Extension Ring in a defined by x^3 + 3*x + 3
  To:   Finite Field in a0 of size 5^3
```

is_injective()

The reduction map is far from injective.

EXAMPLES:

```
sage: GF(5).convert_map_from(ZpCA(5)).is_injective()                    #␣
↪needs sage.rings.finite_rings
False
```

is_surjective()

The reduction map is surjective.

EXAMPLES:

```
sage: GF(7).convert_map_from(Qp(7)).is_surjective()                     #␣
↪needs sage.rings.finite_rings
True
```

section()

Return the section from the residue ring or field back to the p-adic ring or field.

EXAMPLES:

```
sage: GF(3).convert_map_from(Zp(3)).section()                           #␣
↪needs sage.rings.finite_rings
Lifting morphism:
  From: Finite Field of size 3
  To:   3-adic Ring with capped relative precision 20
```

sage.rings.padics.padic_generic.**local_print_mode**(*obj*, *print_options*, *pos=None*, *ram_name=None*)

Context manager for safely temporarily changing the print_mode of a p-adic ring/field.

EXAMPLES:

```
sage: R = Zp(5)
sage: R(45)
4*5 + 5^2 + O(5^21)
sage: with local_print_mode(R, 'val-unit'):
....:     print(R(45))
5 * 9 + O(5^21)
```

Note: For more documentation see `sage.structure.parent_gens.localvars`.

class sage.rings.padics.padic_generic.**pAdicGeneric**(*base*, *p*, *prec*, *print_mode*, *names*, *element_class*, *category=None*)

Bases: `PrincipalIdealDomain`, `LocalGeneric`

Initialize `self`.

INPUT:

- `base` – base ring
- `p` – prime
- `print_mode` – dictionary of print options

- `names` – how to print the uniformizer
- `element_class` – the class for elements of this ring

EXAMPLES:

```
sage: R = Zp(17)  # indirect doctest
```

`characteristic()`

Return the characteristic of `self`, which is always 0.

EXAMPLES:

```
sage: R = Zp(3, 10,'fixed-mod'); R.characteristic()
0
```

`extension`(*modulus*, *prec=None*, *names=None*, *print_mode=None*, *implementation='FLINT'*, ***kwds*)

Create an extension of this p-adic ring.

EXAMPLES:

```
sage: # needs sage.libs.ntl
sage: k = Qp(5)
sage: R.<x> = k[]
sage: l.<w> = k.extension(x^2 - 5); l
5-adic Eisenstein Extension Field in w defined by x^2 - 5
sage: F = list(Qp(19)['x'](cyclotomic_polynomial(5)).factor())[0][0]
sage: L = Qp(19).extension(F, names='a'); L
19-adic Unramified Extension Field in a defined by x^2 +␣
↪8751674996211859573806383*x + 1
```

`fraction_field`(*print_mode=None*)

Return the fraction field of this ring or field.

For $\mathbf{Z}_p$, this is the p-adic field with the same options, and for extensions, it is just the extension of the fraction field of the base determined by the same polynomial.

The fraction field of a capped absolute ring is capped relative, and that of a fixed modulus ring is floating point.

INPUT:

- `print_mode` – (optional) a dictionary containing print options; defaults to the same options as this ring

OUTPUT:

- the fraction field of this ring

EXAMPLES:

```
sage: R = Zp(5, print_mode='digits', show_prec=False)
sage: K = R.fraction_field(); K(1/3)
31313131313131313132
sage: L = R.fraction_field({'max_ram_terms':4}); L(1/3)
doctest:warning
...
DeprecationWarning: Use the change method if you want to change print options␣
↪in fraction_field()
```

(continues on next page)

(continued from previous page)

```
See https://github.com/sagemath/sage/issues/23227 for details.
3132
sage: U.<a> = Zq(17^4, 6, print_mode='val-unit', print_max_terse_terms=3)    #␣
↪needs sage.libs.ntl
sage: U.fraction_field()                                                     #␣
↪needs sage.libs.ntl
17-adic Unramified Extension Field in a defined by x^4 + 7*x^2 + 10*x + 3
sage: U.fraction_field({"pos":False}) == U.fraction_field()                  #␣
↪needs sage.libs.ntl
False
```

frobenius_endomorphism(*n=1*)

Return the n-th power of the absolute arithmetic Frobeninus endomorphism on this field.

INPUT:

- n – an integer (default: 1)

EXAMPLES:

```
sage: K.<a> = Qq(3^5)                                                        #␣
↪needs sage.libs.ntl
sage: Frob = K.frobenius_endomorphism(); Frob                                #␣
↪needs sage.libs.ntl
Frobenius endomorphism on 3-adic Unramified Extension
... lifting a |--> a^3 on the residue field
sage: Frob(a) == a.frobenius()                                               #␣
↪needs sage.libs.ntl
True
```

We can specify a power:

```
sage: K.frobenius_endomorphism(2)                                            #␣
↪needs sage.libs.ntl
Frobenius endomorphism on 3-adic Unramified Extension
... lifting a |--> a^(3^2) on the residue field
```

The result is simplified if possible:

```
sage: K.frobenius_endomorphism(6)                                            #␣
↪needs sage.libs.ntl
Frobenius endomorphism on 3-adic Unramified Extension
... lifting a |--> a^3 on the residue field
sage: K.frobenius_endomorphism(5)                                            #␣
↪needs sage.libs.ntl
Identity endomorphism of 3-adic Unramified Extension ...
```

Comparisons work:

```
sage: K.frobenius_endomorphism(6) == Frob                                    #␣
↪needs sage.libs.ntl
True
```

gens()

Return a list of generators.

EXAMPLES:

```
sage: R = Zp(5); R.gens()
[5 + O(5^21)]
sage: Zq(25,names='a').gens()                                              #␣
↪needs sage.libs.ntl
[a + O(5^20)]
sage: S.<x> = ZZ[]; f = x^5 + 25*x -5; W.<w> = R.ext(f); W.gens()         #␣
↪needs sage.libs.ntl
[w + O(w^101)]
```

integer_ring(*print_mode=None*)

Return the ring of integers of this ring or field.

For $\mathbf{Q}_p$, this is the p-adic ring with the same options, and for extensions, it is just the extension of the ring of integers of the base determined by the same polynomial.

INPUT:

- `print_mode` – (optional) a dictionary containing print options; defaults to the same options as this ring

OUTPUT:

- the ring of elements of this field with nonnegative valuation

EXAMPLES:

```
sage: K = Qp(5, print_mode='digits', show_prec=False)
sage: R = K.integer_ring(); R(1/3)
31313131313131313132
sage: S = K.integer_ring({'max_ram_terms':4}); S(1/3)
doctest:warning
...
DeprecationWarning: Use the change method if you want to change print options␣
↪in integer_ring()
See https://github.com/sagemath/sage/issues/23227 for details.
3132
sage: U.<a> = Qq(17^4, 6, print_mode='val-unit', print_max_terse_terms=3)   #␣
↪needs sage.libs.ntl
sage: U.integer_ring()                                                     #␣
↪needs sage.libs.ntl
17-adic Unramified Extension Ring in a defined by x^4 + 7*x^2 + 10*x + 3
sage: U.fraction_field({"print_mode":"terse"}) == U.fraction_field()       #␣
↪needs sage.libs.ntl
doctest:warning
...
DeprecationWarning: Use the change method if you want to change print options␣
↪in fraction_field()
See https://github.com/sagemath/sage/issues/23227 for details.
False
```

ngens()

Return the number of generators of `self`.

We conventionally define this as 1: for base rings, we take a uniformizer as the generator; for extension rings, we take a root of the minimal polynomial defining the extension.

EXAMPLES:

```
sage: Zp(5).ngens()
1
sage: Zq(25,names='a').ngens()                                              #␣
↪needs sage.libs.ntl
1
```

prime()

Return the prime, ie the characteristic of the residue field.

OUTPUT:

The characteristic of the residue field.

EXAMPLES:

```
sage: R = Zp(3,5,'fixed-mod')
sage: R.prime()
3
```

primitive_root_of_unity(*n=None*, *order=False*)

Return a generator of the group of `n`-th roots of unity in this ring.

INPUT:

- `n` – an integer or `None` (default: `None`)
- `order` – a boolean (default: `False`)

OUTPUT:

A generator of the group of `n`-th roots of unity. If `n` is `None`, a generator of the full group of roots of unity is returned.

If `order` is `True`, the order of the above group is returned as well.

EXAMPLES:

```
sage: R = Zp(5, 10)
sage: zeta = R.primitive_root_of_unity(); zeta
2 + 5 + 2*5^2 + 5^3 + 3*5^4 + 4*5^5 + 2*5^6 + 3*5^7 + 3*5^9 + O(5^10)
sage: zeta == R.teichmuller(2)
True
```

Now we consider an example with non trivial `p`-th roots of unity:

```
sage: # needs sage.libs.ntl
sage: W = Zp(3, 2)
sage: S.<x> = W[]
sage: R.<pi> = W.extension((x+1)^6 + (x+1)^3 + 1)
sage: zeta, order = R.primitive_root_of_unity(order=True)
sage: zeta
2 + 2*pi + 2*pi^3 + 2*pi^7 + 2*pi^8 + 2*pi^9 + pi^11 + O(pi^12)
sage: order
18
sage: zeta.multiplicative_order()
18
```

(continues on next page)

(continued from previous page)

```
sage: zeta, order = R.primitive_root_of_unity(24, order=True)
sage: zeta
2 + pi^3 + 2*pi^7 + 2*pi^8 + 2*pi^10 + 2*pi^11 + O(pi^12)
sage: order   # equal to gcd(18,24)
6
sage: zeta.multiplicative_order()
6
```

print_mode()

Return the current print mode as a string.

EXAMPLES:

```
sage: R = Qp(7,5, 'capped-rel')
sage: R.print_mode()
'series'
```

residue_characteristic()

Return the prime, i.e., the characteristic of the residue field.

OUTPUT:

The characteristic of the residue field.

EXAMPLES:

```
sage: R = Zp(3,5,'fixed-mod')
sage: R.residue_characteristic()
3
```

residue_class_field()

Return the residue class field.

EXAMPLES:

```
sage: R = Zp(3,5,'fixed-mod')
sage: k = R.residue_class_field()
sage: k
Finite Field of size 3
```

residue_field()

Return the residue class field.

EXAMPLES:

```
sage: R = Zp(3,5,'fixed-mod')
sage: k = R.residue_field()
sage: k
Finite Field of size 3
```

residue_ring(n)

Return the quotient of the ring of integers by the `n`-th power of the maximal ideal.

EXAMPLES:

```
sage: R = Zp(11)
sage: R.residue_ring(3)
Ring of integers modulo 1331
```

residue_system()

Return a list of elements representing all the residue classes.

EXAMPLES:

```
sage: R = Zp(3, 5,'fixed-mod')
sage: R.residue_system()
[0, 1, 2]
```

roots_of_unity(*n=None*)

Return all the n-th roots of unity in this ring.

INPUT:

- `n` – an integer or `None` (default: `None`); if `None`, the full group of roots of unity is returned

EXAMPLES:

```
sage: R = Zp(5, 10)
sage: roots = R.roots_of_unity(); roots
[1 + O(5^10),
 2 + 5 + 2*5^2 + 5^3 + 3*5^4 + 4*5^5 + 2*5^6 + 3*5^7 + 3*5^9 + O(5^10),
 4 + 4*5 + 4*5^2 + 4*5^3 + 4*5^4 + 4*5^5 + 4*5^6 + 4*5^7 + 4*5^8 + 4*5^9 + O(5^
↪10),
 3 + 3*5 + 2*5^2 + 3*5^3 + 5^4 + 2*5^6 + 5^7 + 4*5^8 + 5^9 + O(5^10)]

sage: R.roots_of_unity(10)
[1 + O(5^10),
 4 + 4*5 + 4*5^2 + 4*5^3 + 4*5^4 + 4*5^5 + 4*5^6 + 4*5^7 + 4*5^8 + 4*5^9 + O(5^
↪10)]
```

In this case, the roots of unity are the Teichmüller representatives:

```
sage: R.teichmuller_system()
[1 + O(5^10),
 2 + 5 + 2*5^2 + 5^3 + 3*5^4 + 4*5^5 + 2*5^6 + 3*5^7 + 3*5^9 + O(5^10),
 3 + 3*5 + 2*5^2 + 3*5^3 + 5^4 + 2*5^6 + 5^7 + 4*5^8 + 5^9 + O(5^10),
 4 + 4*5 + 4*5^2 + 4*5^3 + 4*5^4 + 4*5^5 + 4*5^6 + 4*5^7 + 4*5^8 + 4*5^9 + O(5^
↪10)]
```

In general, there might be more roots of unity (it happens when the ring has non trivial p-th roots of unity):

```
sage: # needs sage.libs.ntl
sage: W.<a> = Zq(3^2, 2)
sage: S.<x> = W[]
sage: R.<pi> = W.extension((x+1)^2 + (x+1) + 1)
sage: roots = R.roots_of_unity(); roots
[1 + O(pi^4),
 a + 2*a*pi + 2*a*pi^2 + a*pi^3 + O(pi^4),
 ...
 1 + pi + O(pi^4),
```

(continues on next page)

(continued from previous page)

```
 a + a*pi^2 + 2*a*pi^3 + O(pi^4),
 ...
 1 + 2*pi + pi^2 + O(pi^4),
 a + a*pi + a*pi^2 + O(pi^4),
 ...]
sage: len(roots)
24
```

We check that the logarithm of each root of unity vanishes:

```
sage: # needs sage.libs.ntl
sage: for root in roots:
....:     if root.log() != 0:
....:         raise ValueError
```

some_elements()

Return a list of elements in this ring.

This is typically used for running generic tests (see TestSuite).

EXAMPLES:

```
sage: Zp(2,4).some_elements()
[0, 1 + O(2^4), 2 + O(2^5), 1 + 2^2 + 2^3 + O(2^4), 2 + 2^2 + 2^3 + 2^4 + O(2^
↪5)]
```

teichmuller(*x*, *prec=None*)

Return the Teichmüller representative of x.

- x – something that can be cast into self

OUTPUT:

- the Teichmüller lift of x

EXAMPLES:

```
sage: R = Zp(5, 10, 'capped-rel', 'series')
sage: R.teichmuller(2)
2 + 5 + 2*5^2 + 5^3 + 3*5^4 + 4*5^5 + 2*5^6 + 3*5^7 + 3*5^9 + O(5^10)
sage: R = Qp(5, 10,'capped-rel','series')
sage: R.teichmuller(2)
2 + 5 + 2*5^2 + 5^3 + 3*5^4 + 4*5^5 + 2*5^6 + 3*5^7 + 3*5^9 + O(5^10)
sage: R = Zp(5, 10, 'capped-abs', 'series')
sage: R.teichmuller(2)
2 + 5 + 2*5^2 + 5^3 + 3*5^4 + 4*5^5 + 2*5^6 + 3*5^7 + 3*5^9 + O(5^10)
sage: R = Zp(5, 10, 'fixed-mod', 'series')
sage: R.teichmuller(2)
2 + 5 + 2*5^2 + 5^3 + 3*5^4 + 4*5^5 + 2*5^6 + 3*5^7 + 3*5^9

sage: # needs sage.libs.ntl
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
```

(continues on next page)

(continued from previous page)

```
sage: y = W.teichmuller(3); y
3 + 3*w^5 + w^7 + 2*w^9 + 2*w^10 + 4*w^11 + w^12 + 2*w^13 + 3*w^15
 + 2*w^16 + 3*w^17 + w^18 + 3*w^19 + 3*w^20 + 2*w^21 + 2*w^22
 + 3*w^23 + 4*w^24 + O(w^25)
sage: y^5 == y
True
sage: g = x^3 + 3*x + 3
sage: A.<a> = R.ext(g)
sage: b = A.teichmuller(1 + 2*a - a^2); b
(4*a^2 + 2*a + 1) + 2*a*5 + (3*a^2 + 1)*5^2 + (a + 4)*5^3
 + (a^2 + a + 1)*5^4 + O(5^5)
sage: b^125 == b
True
```

We check that github issue #23736 is resolved:

```
sage: # needs sage.libs.ntl
sage: R.teichmuller(GF(5)(2))
2 + 5 + 2*5^2 + 5^3 + 3*5^4 + O(5^5)
```

AUTHORS:

- Initial version: David Roe
- Quadratic time version: Kiran Kedlaya <kedlaya@math.mit.edu> (2007-03-27)

teichmuller_system()

Return a set of Teichmüller representatives for the invertible elements of $\mathbf{Z}/p\mathbf{Z}$.

OUTPUT:

A list of Teichmüller representatives for the invertible elements of $\mathbf{Z}/p\mathbf{Z}$.

EXAMPLES:

```
sage: R = Zp(3, 5,'fixed-mod', 'terse')
sage: R.teichmuller_system()
[1, 242]
```

Check that github issue #20457 is fixed:

```
sage: F.<a> = Qq(5^2,6)                                                  #␣
↪needs sage.libs.ntl
sage: F.teichmuller_system()[3]                                          #␣
↪needs sage.libs.ntl
(2*a + 2) + (4*a + 1)*5 + 4*5^2 + (2*a + 1)*5^3 + (4*a + 1)*5^4 + (2*a + 3)*5^5␣
↪+ O(5^6)
```

Note: Should this return 0 as well?

uniformizer_pow(n)

Return p^n, as an element of `self`.

If `n` is infinity, returns 0.

EXAMPLES:

```
sage: R = Zp(3, 5, 'fixed-mod')
sage: R.uniformizer_pow(3)
3^3
sage: R.uniformizer_pow(infinity)
0
```

valuation()

Return the p-adic valuation on this ring.

OUTPUT:

A valuation that is normalized such that the rational prime p has valuation 1.

EXAMPLES:

```
sage: # needs sage.libs.ntl
sage: K = Qp(3)
sage: R.<a> = K[]
sage: L.<a> = K.extension(a^3 - 3)
sage: v = L.valuation(); v
3-adic valuation
sage: v(3)
1
sage: L(3).valuation()
3
```

The normalization is chosen such that the valuation restricts to the valuation on the base ring:

```
sage: v(3) == K.valuation()(3)                                          #␣
↪needs sage.libs.ntl
True
sage: v.restriction(K) == K.valuation()                                 #␣
↪needs sage.libs.ntl
True
```

See also:

NumberField_generic.valuation(), Order.valuation()

CHAPTER

FIVE

P-ADIC GENERIC NODES

This file contains a bunch of intermediate classes for the p-adic parents, allowing a function to be implemented at the right level of generality.

AUTHORS:

- David Roe

class sage.rings.padics.generic_nodes.**CappedAbsoluteGeneric**(*base*, *prec*, *names*, *element_class*, *category=None*)

Bases: *LocalGeneric*

is_capped_absolute()

Return whether this p-adic ring bounds precision in a capped absolute fashion.

The absolute precision of an element is the power of p modulo which that element is defined. In a capped absolute ring, the absolute precision of elements are bounded by a constant depending on the ring.

EXAMPLES:

```
sage: R = ZpCA(5, 15)
sage: R.is_capped_absolute()
True
sage: R(5^7)
5^7 + O(5^15)
sage: S = Zp(5, 15)
sage: S.is_capped_absolute()
False
sage: S(5^7)
5^7 + O(5^22)
```

class sage.rings.padics.generic_nodes.**CappedRelativeFieldGeneric**(*base*, *prec*, *names*, *element_class*, *category=None*)

Bases: *CappedRelativeGeneric*

class sage.rings.padics.generic_nodes.**CappedRelativeGeneric**(*base*, *prec*, *names*, *element_class*, *category=None*)

Bases: *LocalGeneric*

is_capped_relative()

Return whether this p-adic ring bounds precision in a capped relative fashion.

The relative precision of an element is the power of p modulo which the unit part of that element is defined. In a capped relative ring, the relative precision of elements are bounded by a constant depending on the ring.

EXAMPLES:

```
sage: R = ZpCA(5, 15)
sage: R.is_capped_relative()
False
sage: R(5^7)
5^7 + O(5^15)
sage: S = Zp(5, 15)
sage: S.is_capped_relative()
True
sage: S(5^7)
5^7 + O(5^22)
```

class sage.rings.padics.generic_nodes.**CappedRelativeRingGeneric**(*base*, *prec*, *names*, *element_class*, *category=None*)

Bases: *CappedRelativeGeneric*

class sage.rings.padics.generic_nodes.**FixedModGeneric**(*base*, *prec*, *names*, *element_class*, *category=None*)

Bases: *LocalGeneric*

is_fixed_mod()

Return whether this p-adic ring bounds precision in a fixed modulus fashion.

The absolute precision of an element is the power of p modulo which that element is defined. In a fixed modulus ring, the absolute precision of every element is defined to be the precision cap of the parent. This means that some operations, such as division by p, don't return a well defined answer.

EXAMPLES:

```
sage: R = ZpFM(5,15)
sage: R.is_fixed_mod()
True
sage: R(5^7,absprec=9)
5^7
sage: S = ZpCA(5, 15)
sage: S.is_fixed_mod()
False
sage: S(5^7,absprec=9)
5^7 + O(5^9)
```

class sage.rings.padics.generic_nodes.**FloatingPointFieldGeneric**(*base*, *prec*, *names*, *element_class*, *category=None*)

Bases: *FloatingPointGeneric*

class sage.rings.padics.generic_nodes.**FloatingPointGeneric**(*base*, *prec*, *names*, *element_class*, *category=None*)

Bases: *LocalGeneric*

is_floating_point()

Return whether this p-adic ring uses a floating point precision model.

Elements in the floating point model are stored by giving a valuation and a unit part. Arithmetic is done where the unit part is truncated modulo a fixed power of the uniformizer, stored in the precision cap of the parent.

EXAMPLES:

```
sage: R = ZpFP(5,15)
sage: R.is_floating_point()
True
sage: R(5^7,absprec=9)
5^7
sage: S = ZpCR(5,15)
sage: S.is_floating_point()
False
sage: S(5^7,absprec=9)
5^7 + O(5^9)
```

class sage.rings.padics.generic_nodes.**FloatingPointRingGeneric**(*base*, *prec*, *names*, *element_class*, *category=None*)

Bases: *FloatingPointGeneric*

class sage.rings.padics.generic_nodes.**pAdicCappedAbsoluteRingGeneric**(*base*, *p*, *prec*, *print_mode*, *names*, *element_class*, *category=None*)

Bases: *pAdicRingGeneric*, *CappedAbsoluteGeneric*

class sage.rings.padics.generic_nodes.**pAdicCappedRelativeFieldGeneric**(*base*, *p*, *prec*, *print_mode*, *names*, *element_class*, *category=None*)

Bases: *pAdicFieldGeneric*, *CappedRelativeFieldGeneric*

class sage.rings.padics.generic_nodes.**pAdicCappedRelativeRingGeneric**(*base*, *p*, *prec*, *print_mode*, *names*, *element_class*, *category=None*)

Bases: *pAdicRingGeneric*, *CappedRelativeRingGeneric*

class sage.rings.padics.generic_nodes.**pAdicFieldBaseGeneric**(*p*, *prec*, *print_mode*, *names*, *element_class*)

Bases: *pAdicBaseGeneric*, *pAdicFieldGeneric*

composite(*subfield1*, *subfield2*)

Return the composite of two subfields of `self`, i.e., the largest subfield containing both

INPUT:

- `self` – a p-adic field
- `subfield1` – a subfield
- `subfield2` – a subfield

OUTPUT:

the composite of `subfield1` and `subfield2`

EXAMPLES:

```
sage: K = Qp(17); K.composite(K, K) is K
True
```

construction(*forbid_frac_field=False*)

Return the functorial construction of `self`, namely, completion of the rational numbers with respect a given prime.

Also preserves other information that makes this field unique (e.g., precision, rounding, print mode).

INPUT:

- `forbid_frac_field` – require a completion functor rather than a fraction field functor. This is used in the *sage.rings.padics.local_generic.LocalGeneric.change()* method.

EXAMPLES:

```
sage: K = Qp(17, 8, print_mode='val-unit', print_sep='&')
sage: c, L = K.construction(); L
17-adic Ring with capped relative precision 8
sage: c
FractionField
sage: c(L)
17-adic Field with capped relative precision 8
sage: K == c(L)
True
```

We can get a completion functor by forbidding the fraction field:

```
sage: c, L = K.construction(forbid_frac_field=True); L
Rational Field
sage: c
Completion[17, prec=8]
sage: c(L)
17-adic Field with capped relative precision 8
sage: K == c(L)
True
```

subfield(*list*)

Return the subfield generated by the elements in `list`

INPUT:

- `self` – a p-adic field
- `list` – a list of elements of `self`

OUTPUT:

the subfield of `self` generated by the elements of `list`

EXAMPLES:

```
sage: K = Qp(17); K.subfield([K(17), K(1827)]) is K
True
```

subfields_of_degree(*n*)

Return the number of subfields of `self` of degree n

INPUT:

- `self` – a p-adic field
- `n` – an integer

OUTPUT:

integer – the number of subfields of degree n over `self.base_ring()`

EXAMPLES:

```
sage: K = Qp(17)
sage: K.subfields_of_degree(1)
1
```

class sage.rings.padics.generic_nodes.**pAdicFieldGeneric**(*base*, *p*, *prec*, *print_mode*, *names*, *element_class*, *category=None*)

Bases: *pAdicGeneric*, pAdicField

class sage.rings.padics.generic_nodes.**pAdicFixedModRingGeneric**(*base*, *p*, *prec*, *print_mode*, *names*, *element_class*, *category=None*)

Bases: *pAdicRingGeneric*, *FixedModGeneric*

class sage.rings.padics.generic_nodes.**pAdicFloatingPointFieldGeneric**(*base*, *p*, *prec*, *print_mode*, *names*, *element_class*, *category=None*)

Bases: *pAdicFieldGeneric*, *FloatingPointFieldGeneric*

class sage.rings.padics.generic_nodes.**pAdicFloatingPointRingGeneric**(*base*, *p*, *prec*, *print_mode*, *names*, *element_class*, *category=None*)

Bases: *pAdicRingGeneric*, *FloatingPointRingGeneric*

class sage.rings.padics.generic_nodes.**pAdicLatticeGeneric**(*p*, *prec*, *print_mode*, *names*, *label=None*)

Bases: *pAdicGeneric*

An implementation of the p-adic rationals with lattice precision.

INPUT:

- p – the underlying prime number
- prec – the precision
- subtype – either "cap" or "float", specifying the precision model used for tracking precision
- label – a string or None (default: None)

convert_multiple(**elts*)

Convert a list of elements to this parent.

NOTE:

This function tries to be sharp on precision as much as possible. In particular, if the precision of the input elements are handled by a lattice, diffused digits of precision are preserved during the conversion.

EXAMPLES:

```
sage: R = ZpLC(2)
sage: x = R(1, 10); y = R(1, 5)
sage: x,y = x+y, x-y
```

Remark that the pair (x, y) has diffused digits of precision:

```
sage: x
2 + O(2^5)
sage: y
O(2^5)
```

(continues on next page)

(continued from previous page)

```
sage: x + y
2 + O(2^11)

sage: R.precision().diffused_digits([x,y])                          # needs sage.geometry.polyhedron
6
```

As a consequence, if we convert x and y separately, we lose some precision:

```
sage: R2 = ZpLC(2, label='copy')
sage: x2 = R2(x); y2 = R2(y)
sage: x2
2 + O(2^5)
sage: y2
O(2^5)
sage: x2 + y2
2 + O(2^5)

sage: R2.precision().diffused_digits([x2,y2])                       # needs sage.geometry.polyhedron
0
```

On the other hand, this issue disappears when we use multiple conversion:

```
sage: x2,y2 = R2.convert_multiple(x,y)                              # needs sage.geometry.polyhedron
sage: x2 + y2                                                       # needs sage.rings.padics
2 + O(2^11)

sage: R2.precision().diffused_digits([x2,y2])                       # needs sage.geometry.polyhedron
6
```

is_lattice_prec()

Return whether this p-adic ring bounds precision using a lattice model.

In lattice precision, relationships between elements are stored in a precision object of the parent, which allows for optimal precision tracking at the cost of increased memory usage and runtime.

EXAMPLES:

```
sage: R = ZpCR(5, 15)
sage: R.is_lattice_prec()
False
sage: x = R(25, 8)
sage: x - x
O(5^8)
sage: S = ZpLC(5, 15)
sage: S.is_lattice_prec()
True
sage: x = S(25, 8)
sage: x - x
O(5^30)
```

label()

Return the label of this parent.

NOTE:

Labels can be used to distinguish between parents with the same defining data.

They are useful in the lattice precision framework in order to limit the size of the lattice modeling the precision (which is roughly the number of elements having this parent).

Elements of a parent with some label do not coerce to a parent with a different label. However conversions are allowed.

EXAMPLES:

```
sage: R = ZpLC(5)
sage: R.label()  # no label by default

sage: R = ZpLC(5, label='mylabel')
sage: R.label()
'mylabel'
```

Labels are typically useful to isolate computations. For example, assume that we first want to do some calculations with matrices:

```
sage: R = ZpLC(5, label='matrices')
sage: M = random_matrix(R, 4, 4)                                              #␣
↪needs sage.geometry.polyhedron
sage: d = M.determinant()                                                     #␣
↪needs sage.geometry.polyhedron
```

Now, if we want to do another unrelated computation, we can use a different label:

```
sage: R = ZpLC(5, label='polynomials')
sage: S.<x> = PolynomialRing(R)
sage: P = (x-1)*(x-2)*(x-3)*(x-4)*(x-5)
```

Without labels, the software would have modeled the precision on the matrices and on the polynomials using the same lattice (manipulating a lattice of higher dimension can have a significant impact on performance).

precision()

Return the lattice precision object attached to this parent.

EXAMPLES:

```
sage: R = ZpLC(5, label='precision')
sage: R.precision()
Precision lattice on 0 objects (label: precision)

sage: x = R(1, 10); y = R(1, 5)
sage: R.precision()
Precision lattice on 2 objects (label: precision)
```

See also:

`sage.rings.padics.lattice_precision.PrecisionLattice`

precision_cap()

Return the relative precision cap for this ring if it is finite. Otherwise return the absolute precision cap.

EXAMPLES:

```
sage: R = ZpLC(3)
sage: R.precision_cap()
20
sage: R.precision_cap_relative()
20

sage: R = ZpLC(3, prec=(infinity,20))
sage: R.precision_cap()
20
sage: R.precision_cap_relative()
+Infinity
sage: R.precision_cap_absolute()
20
```

See also:

precision_cap_relative(), *precision_cap_absolute()*

precision_cap_absolute()

Return the absolute precision cap for this ring.

EXAMPLES:

```
sage: R = ZpLC(3)
sage: R.precision_cap_absolute()
40

sage: R = ZpLC(3, prec=(infinity,20))
sage: R.precision_cap_absolute()
20
```

See also:

precision_cap(), *precision_cap_relative()*

precision_cap_relative()

Return the relative precision cap for this ring.

EXAMPLES:

```
sage: R = ZpLC(3)
sage: R.precision_cap_relative()
20

sage: R = ZpLC(3, prec=(infinity,20))
sage: R.precision_cap_relative()
+Infinity
```

See also:

precision_cap(), *precision_cap_absolute()*

class sage.rings.padics.generic_nodes.**pAdicRelaxedGeneric**(*base*, *p*, *prec*, *print_mode*, *names*, *element_class*, *category=None*)

Bases: *pAdicGeneric*

Generic class for relaxed p-adics.

INPUT:

- p – the underlying prime number
- prec – the default precision

an_element(*unbounded=False*)

Return an element in this ring.

EXAMPLES:

```
sage: R = ZpER(7, prec=5)                                               #␣
↪needs sage.libs.flint
sage: R.an_element()                                                    #␣
↪needs sage.libs.flint
7 + O(7^5)
sage: R.an_element(unbounded=True)                                      #␣
↪needs sage.libs.flint
7 + ...
```

default_prec()

Return the default precision of this relaxed p-adic ring.

The default precision is mostly used for printing: it is the number of digits which are printed for unbounded elements (that is elements having infinite absolute precision).

EXAMPLES:

```
sage: # needs sage.libs.flint
sage: R = ZpER(5, print_mode="digits")
sage: R.default_prec()
20
sage: R(1/17)
...34024323104201213403
sage: S = ZpER(5, prec=10, print_mode="digits")
sage: S.default_prec()
10
sage: S(1/17)
...4201213403
```

halting_prec()

Return the default halting precision of this relaxed p-adic ring.

The halting precision is the precision at which elements of this parent are compared (unless more digits have been previously computed). By default, it is twice the default precision.

EXAMPLES:

```
sage: R = ZpER(5, print_mode="digits")                                  #␣
↪needs sage.libs.flint
sage: R.halting_prec()                                                  #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.libs.flint
40
```

is_relaxed()

Return whether this p-adic ring is relaxed.

EXAMPLES:

```
sage: R = Zp(5)
sage: R.is_relaxed()
False
sage: S = ZpER(5)                                                    #␣
↪needs sage.libs.flint
sage: S.is_relaxed()                                                 #␣
↪needs sage.libs.flint
True
```

is_secure()

Return `False` if this p-adic relaxed ring is not secure (i.e., if indistinguishable elements at the working precision are considered as equal); `True` otherwise (in which case, an error is raised when equality cannot be decided).

EXAMPLES:

```
sage: # needs sage.libs.flint
sage: R = ZpER(5)
sage: R.is_secure()
False
sage: x = R(20/21)
sage: y = x + 5^50
sage: x == y
True

sage: # needs sage.libs.flint
sage: S = ZpER(5, secure=True)
sage: S.is_secure()
True
sage: x = S(20/21)
sage: y = x + 5^50
sage: x == y
Traceback (most recent call last):
...
PrecisionError: unable to decide equality; try to bound precision
```

precision_cap()

Return the precision cap of this p-adic ring, which is infinite in the case of relaxed rings.

EXAMPLES:

```
sage: R = ZpER(5)                                                    #␣
↪needs sage.libs.flint
sage: R.precision_cap()                                              #␣
↪needs sage.libs.flint
+Infinity
```

random_element(*integral=False*, *prec=None*)

Return a random element in this ring.

INPUT:

- `integral` – a boolean (default: `False`); if `True`, return a random element in the ring of integers of this ring
- `prec` – an integer or `None` (default: `None`); if given, bound the precision of the output to `prec`

EXAMPLES:

```
sage: R = ZpER(5, prec=10)                                                   # needs sage.libs.flint
```

By default, this method returns a unbounded element:

```
sage: a = R.random_element()                                                 # needs sage.libs.flint
sage: a  # random                                                            # needs sage.libs.flint
4 + 3*5 + 3*5^2 + 5^3 + 3*5^4 + 2*5^5 + 2*5^6 + 5^7 + 5^9 + ...
sage: a.precision_absolute()                                                 # needs sage.libs.flint
+Infinity
```

The precision can be bounded by passing in a precision:

```
sage: b = R.random_element(prec=15)                                          # needs sage.libs.flint
sage: b  # random                                                            # needs sage.libs.flint
2 + 3*5^2 + 5^3 + 3*5^4 + 5^5 + 3*5^6 + 3*5^8 + 3*5^9 + 4*5^10
 + 5^11 + 4*5^12 + 5^13 + 2*5^14 + O(5^15)
sage: b.precision_absolute()                                                 # needs sage.libs.flint
15
```

some_elements(*unbounded=False*)

Return a list of elements in this ring.

This is typically used for running generic tests (see `TestSuite`).

EXAMPLES:

```
sage: R = ZpER(7, prec=5)                                                    # needs sage.libs.flint
sage: R.some_elements()                                                      # needs sage.libs.flint
[O(7^5),
 1 + O(7^5),
 7 + O(7^5),
 7 + O(7^5),
 1 + 5*7 + 3*7^2 + 6*7^3 + O(7^5),
 7 + 6*7^2 + 6*7^3 + 6*7^4 + O(7^5)]
```

(continues on next page)

(continued from previous page)

```
sage: R.some_elements(unbounded=True)                                      #␣
↪needs sage.libs.flint
[0,
 1 + ...,
 7 + ...,
 7 + ...,
 1 + 5*7 + 3*7^2 + 6*7^3 + ...,
 7 + 6*7^2 + 6*7^3 + 6*7^4 + ...]
```

teichmuller(x)

Return the Teichmuller representative of x.

EXAMPLES:

```
sage: R = ZpER(5, print_mode="digits")                                     #␣
↪needs sage.libs.flint
sage: R.teichmuller(2)                                                     #␣
↪needs sage.libs.flint
...40423140223032431212
```

teichmuller_system()

Return a set of teichmuller representatives for the invertible elements of $\mathbf{Z}/p\mathbf{Z}$.

EXAMPLES:

```
sage: R = ZpER(7, print_mode="digits")                                     #␣
↪needs sage.libs.flint
sage: R.teichmuller_system()                                               #␣
↪needs sage.libs.flint
[...00000000000000000001,
 ...16412125443426203642,
 ...16412125443426203643,
 ...50254541223240463024,
 ...50254541223240463025,
 ...66666666666666666666]
```

unknown(*start_val=0*, *digits=None*)

Return a self-referent number in this ring.

INPUT:

- `start_val` – an integer (default: 0); a lower bound on the valuation of the returned element
- `digits` – an element, a list or `None` (default: `None`); the first digit or the list of the digits of the returned element

NOTE:

Self-referent numbers are numbers whose digits are defined in terms of the previous ones. This method is used to declare a self-referent number (and optionally, to set its first digits). The definition of the number itself will be given afterwords using to method `sage.rings.padics.relaxed_template.RelaxedElement_unknown.set()` of the element.

EXAMPLES:

sage: R = ZpER(5, prec=10) # needs sage.libs.flint

We declare a self-referent number:

```
sage: a = R.unknown()                                                    #␣
↪needs sage.libs.flint
```

So far, we do not know anything on a (except that it has nonnegative valuation):

```
sage: a                                                                  #␣
↪needs sage.libs.flint
O(5^0)
```

We can now use the method `sage.rings.padics.relaxed_template.RelaxedElement_unknown.set()` to define a. Below, for example, we say that the digits of a have to agree with the digits of $1 + 5a$. Note that the factor 5 shifts the digits; the n-th digit of a is then defined by the previous ones:

```
sage: a.set(1 + 5*a)                                                     #␣
↪needs sage.libs.flint
True
```

After this, a contains the solution of the equation $a = 1 + 5a$, that is $a = -1/4$:

```
sage: a                                                                  #␣
↪needs sage.libs.flint
1 + 5 + 5^2 + 5^3 + 5^4 + 5^5 + 5^6 + 5^7 + 5^8 + 5^9 + ...
```

Here is another example with an equation of degree 2:

```
sage: # needs sage.libs.flint
sage: b = R.unknown()
sage: b.set(1 - 5*b^2)
True
sage: b
1 + 4*5 + 5^2 + 3*5^4 + 4*5^6 + 4*5^8 + 2*5^9 + ...
sage: (sqrt(R(21)) - 1) / 10
1 + 4*5 + 5^2 + 3*5^4 + 4*5^6 + 4*5^8 + 2*5^9 + ...
```

Cross self-referent definitions are also allowed:

```
sage: # needs sage.libs.flint
sage: u = R.unknown()
sage: v = R.unknown()
sage: w = R.unknown()
sage: u.set(1 + 2*v + 3*w^2 + 5*u*v*w)
True
sage: v.set(2 + 4*w + sqrt(1 + 5*u + 10*v + 15*w))
True
sage: w.set(3 + 25*(u*v + v*w + u*w))
True
sage: u
3 + 3*5 + 4*5^2 + 5^3 + 3*5^4 + 5^5 + 5^6 + 3*5^7 + 5^8 + 3*5^9 + ...
sage: v
4*5 + 2*5^2 + 4*5^3 + 5^4 + 5^5 + 3*5^6 + 5^8 + 5^9 + ...
sage: w
3 + 4*5^2 + 4*5^3 + 4*5^4 + 4*5^5 + 2*5^6 + 5^8 + 5^9 + ...
```

class sage.rings.padics.generic_nodes.**pAdicRingBaseGeneric**(*p*, *prec*, *print_mode*, *names*, *element_class*)

Bases: *pAdicBaseGeneric*, *pAdicRingGeneric*

construction(*forbid_frac_field=False*)

Return the functorial construction of `self`, namely, completion of the rational numbers with respect to a given prime.

Also preserves other information that makes this field unique (e.g., precision, rounding, print mode).

INPUT:

- `forbid_frac_field` – ignored, for compatibility with other p-adic types.

EXAMPLES:

```
sage: K = Zp(17, 8, print_mode='val-unit', print_sep='&')
sage: c, L = K.construction(); L
Integer Ring
sage: c(L)
17-adic Ring with capped relative precision 8
sage: K == c(L)
True
```

random_element(*algorithm='default'*)

Return a random element of `self`, optionally using the `algorithm` argument to decide how it generates the element. Algorithms currently implemented:

- `'default'`: Choose a_i, $i \geq 0$, randomly between 0 and $p - 1$ until a nonzero choice is made. Then continue choosing a_i randomly between 0 and $p - 1$ until we reach precision_cap, and return $\sum a_i p^i$.

EXAMPLES:

```
sage: Zp(5,6).random_element().parent() is Zp(5,6)
True
sage: ZpCA(5,6).random_element().parent() is ZpCA(5,6)
True
sage: ZpFM(5,6).random_element().parent() is ZpFM(5,6)
True
```

class sage.rings.padics.generic_nodes.**pAdicRingGeneric**(*base*, *p*, *prec*, *print_mode*, *names*, *element_class*, *category=None*)

Bases: *pAdicGeneric*, `pAdicRing`

is_field(*proof=True*)

Return whether this ring is actually a field, ie `False`.

EXAMPLES:

```
sage: Zp(5).is_field()
False
```

krull_dimension()

Return the Krull dimension of self, i.e. 1

INPUT:

- self – a p-adic ring

OUTPUT:

- the Krull dimension of self. Since self is a p-adic ring, this is 1.

EXAMPLES:

```
sage: Zp(5).krull_dimension()
1
```

CHAPTER

SIX

P-ADIC BASE GENERIC

A superclass for implementations of $\mathbf{Z}_p$ and $\mathbf{Q}_p$.

AUTHORS:

- David Roe

class sage.rings.padics.padic_base_generic.**pAdicBaseGeneric**(*p*, *prec*, *print_mode*, *names*, *element_class*)

Bases: *pAdicGeneric*

Initialization

absolute_discriminant()

Returns the absolute discriminant of this p-adic ring

EXAMPLES:

```
sage: Zp(5).absolute_discriminant()
1
```

discriminant(*K=None*)

Returns the discriminant of this p-adic ring over K

INPUT:

- self – a p-adic ring
- K – a sub-ring of self or None (default: None)

OUTPUT:

- integer – the discriminant of this ring over K (or the absolute discriminant if K is None)

EXAMPLES:

```
sage: Zp(5).discriminant()
1
```

exact_field()

Returns the rational field.

For compatibility with extensions of p-adics.

EXAMPLES:

```
sage: Zp(5).exact_field()
Rational Field
```

exact_ring()

Returns the integer ring.

EXAMPLES:

```
sage: Zp(5).exact_ring()
Integer Ring
```

gen(*n=0*)

Returns the nth generator of this extension. For base rings/fields, we consider the generator to be the prime.

EXAMPLES:

```
sage: R = Zp(5); R.gen()
5 + O(5^21)
```

has_pth_root()

Returns whether or not $\mathbf{Z}_p$ has a primitive p^{th} root of unity.

EXAMPLES:

```
sage: Zp(2).has_pth_root()
True
sage: Zp(17).has_pth_root()
False
```

has_root_of_unity(*n*)

Returns whether or not $\mathbf{Z}_p$ has a primitive n^{th} root of unity.

INPUT:

- `self` – a p-adic ring
- `n` – an integer

OUTPUT:

- `boolean` – whether `self` has primitive n^{th} root of unity

EXAMPLES:

```
sage: R=Zp(37)
sage: R.has_root_of_unity(12)
True
sage: R.has_root_of_unity(11)
False
```

is_abelian()

Returns whether the Galois group is abelian, i.e. `True`. #should this be automorphism group?

EXAMPLES:

```
sage: R = Zp(3, 10,'fixed-mod'); R.is_abelian()
True
```

is_isomorphic(*ring*)

Returns whether `self` and `ring` are isomorphic, i.e. whether `ring` is an implementation of $\mathbf{Z}_p$ for the same prime as `self`.

INPUT:

- `self` – a p-adic ring
- `ring` – a ring

OUTPUT:

- `boolean` – whether `ring` is an implementation of ZZ_p` for the same prime as `self`.

EXAMPLES:

```
sage: R = Zp(5, 15, print_mode='digits'); S = Zp(5, 44, print_max_terms=4); R.
↪is_isomorphic(S)
True
```

is_normal()

Returns whether or not this is a normal extension, i.e. `True`.

EXAMPLES:

```
sage: R = Zp(3, 10,'fixed-mod'); R.is_normal()
True
```

modulus(*exact=False*)

Returns the polynomial defining this extension.

For compatibility with extension fields; we define the modulus to be x-1.

INPUT:

- `exact` – boolean (default `False`), whether to return a polynomial with integer entries.

EXAMPLES:

```
sage: Zp(5).modulus(exact=True)
x
```

plot(*max_points=2500*, ***args*)

Create a visualization of this p-adic ring as a fractal similar to a generalization of the Sierpi'nski triangle.

The resulting image attempts to capture the algebraic and topological characteristics of $\mathbf{Z}_p$.

INPUT:

- `max_points` – the maximum number or points to plot, which controls the depth of recursion (default 2500)
- `**args` – color, size, etc. that are passed to the underlying point graphics objects

REFERENCES:

- Cuoco, A. ‘’Visualizing the p-adic Integers”, The American Mathematical Monthly, Vol. 98, No. 4 (Apr., 1991), pp. 355-364

EXAMPLES:

```
sage: Zp(3).plot()                                                          #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
sage: Zp(5).plot(max_points=625)                                            #␣
↪needs sage.plot
```

(continues on next page)

(continued from previous page)

```
Graphics object consisting of 1 graphics primitive
sage: Zp(23).plot(rgbcolor=(1,0,0))                                          #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

uniformizer()

Returns a uniformizer for this ring.

EXAMPLES:

```
sage: R = Zp(3,5,'fixed-mod', 'series')
sage: R.uniformizer()
3
```

uniformizer_pow(*n*)

Returns the `nth` power of the uniformizer of `self` (as an element of `self`).

EXAMPLES:

```
sage: R = Zp(5)
sage: R.uniformizer_pow(5)
5^5 + O(5^25)
sage: R.uniformizer_pow(infinity)
0
```

zeta(*n=None*)

Returns a generator of the group of roots of unity.

INPUT:

- `self` – a p-adic ring
- `n` – an integer or `None` (default: `None`)

OUTPUT:

- `element` – a generator of the n^{th} roots of unity, or a generator of the full group of roots of unity if `n` is `None`

EXAMPLES:

```
sage: R = Zp(37,5)
sage: R.zeta(12)
8 + 24*37 + 37^2 + 29*37^3 + 23*37^4 + O(37^5)
```

zeta_order()

Returns the order of the group of roots of unity.

EXAMPLES:

```
sage: R = Zp(37); R.zeta_order()
36
sage: Zp(2).zeta_order()
2
```

CHAPTER

SEVEN

P-ADIC EXTENSION GENERIC

A common superclass for all extensions of Qp and Zp.

AUTHORS:

- David Roe

class sage.rings.padics.padic_extension_generic.**DefPolyConversion**

Bases: Morphism

Conversion map between p-adic rings/fields with the same defining polynomial.

INPUT:

- R – a p-adic extension ring or field.
- S – a p-adic extension ring or field with the same defining polynomial.

EXAMPLES:

```
sage: R.<a> = Zq(125, print_mode='terse')
sage: S = R.change(prec = 15, type='floating-point')
sage: a - 1
95367431640624 + a + O(5^20)
sage: S(a - 1)
30517578124 + a + O(5^15)
```

```
sage: R.<a> = Zq(125, print_mode='terse')
sage: S = R.change(prec = 15, type='floating-point')
sage: f = S.convert_map_from(R)
sage: TestSuite(f).run()
```

class sage.rings.padics.padic_extension_generic.**MapFreeModuleToOneStep**

Bases: *pAdicModuleIsomorphism*

The isomorphism from the underlying module of a one-step p-adic extension to the extension.

EXAMPLES:

```
sage: K.<a> = Qq(125)
sage: V, fr, to = K.free_module()
sage: TestSuite(fr).run(skip=['_test_nonzero_equal'])  # skipped since Qq(125) doesn
↪'t have dimension()
```

class sage.rings.padics.padic_extension_generic.**MapFreeModuleToTwoStep**

Bases: *pAdicModuleIsomorphism*

The isomorphism from the underlying module of a two-step p-adic extension to the extension.

EXAMPLES:

```
sage: K.<a> = Qq(125)
sage: R.<x> = ZZ[]
sage: L.<b> = K.extension(x^2 - 5*x + 5)
sage: V, fr, to = L.free_module(base=Qp(5))
sage: TestSuite(fr).run(skip=['_test_nonzero_equal'])  # skipped since L doesn't
↪have dimension()
```

class sage.rings.padics.padic_extension_generic.**MapOneStepToFreeModule**

Bases: *pAdicModuleIsomorphism*

The isomorphism from a one-step p-adic extension to its underlying free module

EXAMPLES:

```
sage: K.<a> = Qq(125)
sage: V, fr, to = K.free_module()
sage: TestSuite(to).run()
```

class sage.rings.padics.padic_extension_generic.**MapTwoStepToFreeModule**

Bases: *pAdicModuleIsomorphism*

The isomorphism from a two-step p-adic extension to its underlying free module

EXAMPLES:

```
sage: K.<a> = Qq(125)
sage: R.<x> = ZZ[]
sage: L.<b> = K.extension(x^2 - 5*x + 5)
sage: V, fr, to = L.free_module(base=Qp(5))
sage: TestSuite(to).run()
```

class sage.rings.padics.padic_extension_generic.**pAdicExtensionGeneric**(*poly*, *prec*, *print_mode*, *names*, *element_class*)

Bases: *pAdicGeneric*

Initialization

EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)  # indirect doctest
```

construction(*forbid_frac_field=False*)

Returns the functorial construction of this ring, namely, the algebraic extension of the base ring defined by the given polynomial.

Also preserves other information that makes this ring unique (e.g. precision, rounding, print mode).

INPUT:

- forbid_frac_field – require a completion functor rather than a fraction field functor. This is used in the *sage.rings.padics.local_generic.LocalGeneric.change()* method.

EXAMPLES:

```
sage: R.<a> = Zq(25, 8, print_mode='val-unit')
sage: c, R0 = R.construction(); R0
5-adic Ring with capped relative precision 8
sage: c(R0)
5-adic Unramified Extension Ring in a defined by x^2 + 4*x + 2
sage: c(R0) == R
True
```

For a field, by default we return a fraction field functor.

```
sage: K.<a> = Qq(25, 8)
sage: c, R = K.construction(); R
5-adic Unramified Extension Ring in a defined by x^2 + 4*x + 2
sage: c
FractionField
```

If you prefer an extension functor, you can use the forbit_frac_field keyword:

```
sage: c, R = K.construction(forbid_frac_field=True); R
5-adic Field with capped relative precision 8
sage: c
AlgebraicExtensionFunctor
sage: c(R) is K
True
```

defining_polynomial(*var=None*, *exact=False*)

Returns the polynomial defining this extension.

INPUT:

- var – string (default: 'x'), the name of the variable
- **exact – boolean (default False), whether to return the underlying exact**
  defining polynomial rather than the one with coefficients in the base ring.

EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: W.defining_polynomial()
(1 + O(5^5))*x^5 + O(5^6)*x^4 + (3*5^2 + O(5^6))*x^3 + (2*5 + 4*5^2 + 4*5^3 +
↪4*5^4 + 4*5^5 + O(5^6))*x^2 + (5^3 + O(5^6))*x + 4*5 + 4*5^2 + 4*5^3 + 4*5^4
↪+ 4*5^5 + O(5^6)
sage: W.defining_polynomial(exact=True)
x^5 + 75*x^3 - 15*x^2 + 125*x - 5

sage: W.defining_polynomial(var='y', exact=True)
y^5 + 75*y^3 - 15*y^2 + 125*y - 5
```

See also:

modulus() exact_field()

exact_field()

Return a number field with the same defining polynomial.

Note that this method always returns a field, even for a p-adic ring.

EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: W.exact_field()
Number Field in w with defining polynomial x^5 + 75*x^3 - 15*x^2 + 125*x - 5
```

See also:

defining_polynomial() modulus()

exact_ring()

Return the order with the same defining polynomial.

Will raise a `ValueError` if the coefficients of the defining polynomial are not integral.

EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: W.exact_ring()
Order in Number Field in w with defining polynomial x^5 + 75*x^3 - 15*x^2 +␣
↪125*x - 5

sage: T = Zp(5,5)
sage: U.<z> = T[]
sage: g = 2*z^4 + 1
sage: V.<v> = T.ext(g)
sage: V.exact_ring()
Traceback (most recent call last):
...
ValueError: each generator must be integral
```

free_module(*base=None*, *basis=None*, *map=True*)

Return a free module V over a specified base ring together with maps to and from V.

INPUT:

- `base` – a subring R so that this ring/field is isomorphic to a finite-rank free R-module V
- `basis` – a basis for this ring/field over the base
- `map` – boolean (default `True`), whether to return R-linear maps to and from V

OUTPUT:

- A finite-rank free R-module V

- An R-module isomorphism from V to this ring/field (only included if `map` is `True`)
- An R-module isomorphism from this ring/field to V (only included if `map` is `True`)

EXAMPLES:

```
sage: R.<x> = ZZ[]
sage: K.<a> = Qq(125)
sage: L.<pi> = K.extension(x^2-5)
sage: V, from_V, to_V = K.free_module()
sage: W, from_W, to_W = L.free_module()
sage: W0, from_W0, to_W0 = L.free_module(base=Qp(5))
sage: to_V(a + O(5^7))
(O(5^7), 1 + O(5^7), O(5^7))
sage: to_W(a)
(a + O(5^20), O(5^20))
sage: to_W0(a + O(5^7))
(O(5^7), 1 + O(5^7), O(5^7), O(5^7), O(5^7), O(5^7))
sage: to_W(pi)
(O(5^21), 1 + O(5^20))
sage: to_W0(pi + O(pi^11))
(O(5^6), O(5^6), O(5^6), 1 + O(5^5), O(5^5), O(5^5))

sage: X, from_X, to_X = K.free_module(K)
sage: to_X(a)
(a + O(5^20))
```

ground_ring()

Returns the ring of which this ring is an extension.

EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: W.ground_ring()
5-adic Ring with capped relative precision 5
```

ground_ring_of_tower()

Returns the p-adic base ring of which this is ultimately an extension.

Currently this function is identical to ground_ring(), since relative extensions have not yet been implemented.

EXAMPLES:

```
sage: Qq(27,30,names='a').ground_ring_of_tower()
3-adic Field with capped relative precision 30
```

modulus(*exact=False*)

Returns the polynomial defining this extension.

INPUT:

- **`exact` – boolean (default `False`), whether to return the underlying exact**
  defining polynomial rather than the one with coefficients in the base ring.

EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: W.modulus()
(1 + O(5^5))*x^5 + O(5^6)*x^4 + (3*5^2 + O(5^6))*x^3 + (2*5 + 4*5^2 + 4*5^3 +
↪4*5^4 + 4*5^5 + O(5^6))*x^2 + (5^3 + O(5^6))*x + 4*5 + 4*5^2 + 4*5^3 + 4*5^4
↪+ 4*5^5 + O(5^6)
sage: W.modulus(exact=True)
x^5 + 75*x^3 - 15*x^2 + 125*x - 5
```

See also:

defining_polynomial() exact_field()

polynomial_ring()

Returns the polynomial ring of which this is a quotient.

EXAMPLES:

```
sage: Qq(27,30,names='a').polynomial_ring()
Univariate Polynomial Ring in x over 3-adic Field with capped relative
↪precision 30
```

random_element()

Return a random element of `self`.

This is done by picking a random element of the ground ring self.degree() times, then treating those elements as coefficients of a polynomial in self.gen().

EXAMPLES:

```
sage: R.<a> = Zq(125, 5)
sage: R.random_element().parent() is R
True
sage: R = Zp(5,3); S.<x> = ZZ[]; f = x^5 + 25*x^2 - 5; W.<w> = R.ext(f)
sage: W.random_element().parent() is W
True
```

class sage.rings.padics.padic_extension_generic.**pAdicModuleIsomorphism**

Bases: `Map`

A base class for various isomorphisms between p-adic rings/fields and free modules

EXAMPLES:

```
sage: K.<a> = Qq(125)
sage: V, fr, to = K.free_module()
sage: from sage.rings.padics.padic_extension_generic import pAdicModuleIsomorphism
sage: isinstance(fr, pAdicModuleIsomorphism)
True
```

is_injective()

EXAMPLES:

```
sage: K.<a> = Qq(125)
sage: V, fr, to = K.free_module()
sage: fr.is_injective()
True
```

is_surjective()

EXAMPLES:

```
sage: K.<a> = Qq(125)
sage: V, fr, to = K.free_module()
sage: fr.is_surjective()
True
```

CHAPTER

EIGHT

EISENSTEIN EXTENSION GENERIC

This file implements the shared functionality for Eisenstein extensions.

AUTHORS:

- David Roe

class sage.rings.padics.eisenstein_extension_generic.**EisensteinExtensionGeneric**(*poly*, *prec*, *print_mode*, *names*, *element_class*)

Bases: *pAdicExtensionGeneric*

Initializes self.

EXAMPLES:

```
sage: A = Zp(7,10)
sage: S.<x> = A[]                                                    # needs sage.libs.ntl
sage: B.<t> = A.ext(x^2+7)  # indirect doctest                       # needs sage.libs.ntl sage.rings.padics
```

absolute_e()

Return the absolute ramification index of this ring or field

EXAMPLES:

```
sage: K.<a> = Qq(3^5)                                                # needs sage.libs.ntl
sage: K.absolute_e()                                                 # needs sage.libs.ntl
1

sage: x = polygen(ZZ, 'x')
sage: L.<pi> = Qp(3).extension(x^2 - 3)                              # needs sage.libs.ntl
sage: L.absolute_e()                                                 # needs sage.libs.ntl
2
```

gen(*n=0*)

Return a generator for self as an extension of its ground ring.

EXAMPLES:

```
sage: A = Zp(7,10)
sage: S.<x> = A[]                                                      #␣
↪needs sage.libs.ntl
sage: B.<t> = A.ext(x^2 + 7)                                           #␣
↪needs sage.libs.ntl
sage: B.gen()                                                          #␣
↪needs sage.libs.ntl
t + O(t^21)
```

inertia_subring()

Return the inertia subring.

Since an Eisenstein extension is totally ramified, this is just the ground field.

EXAMPLES:

```
sage: A = Zp(7,10)
sage: S.<x> = A[]                                                      #␣
↪needs sage.libs.ntl
sage: B.<t> = A.ext(x^2 + 7)                                           #␣
↪needs sage.libs.ntl
sage: B.inertia_subring()                                              #␣
↪needs sage.libs.ntl
7-adic Ring with capped relative precision 10
```

residue_class_field()

Return the residue class field.

INPUT:

- `self` – a p-adic ring

OUTPUT:

the residue field

EXAMPLES:

```
sage: A = Zp(7,10)
sage: S.<x> = A[]                                                      #␣
↪needs sage.libs.ntl
sage: B.<t> = A.ext(x^2 + 7)                                           #␣
↪needs sage.libs.ntl
sage: B.residue_class_field()                                          #␣
↪needs sage.libs.ntl
Finite Field of size 7
```

residue_ring(n)

Return the quotient of the ring of integers by the n-th power of its maximal ideal.

EXAMPLES:

```
sage: S.<x> = ZZ[]
sage: W.<w> = Zp(5).extension(x^2 - 5)                                 #␣
↪needs sage.libs.ntl
sage: W.residue_ring(1)                                                #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.libs.ntl
Ring of integers modulo 5
```

The following requires implementing more general Artinian rings:

```
sage: W.residue_ring(2)                                                  #␣
↪needs sage.libs.ntl
Traceback (most recent call last):
...
NotImplementedError
```

uniformizer()

Return the uniformizer of `self`, i.e., a generator for the unique maximal ideal.

EXAMPLES:

```
sage: A = Zp(7,10)
sage: S.<x> = A[]                                                        #␣
↪needs sage.libs.ntl
sage: B.<t> = A.ext(x^2 + 7)                                             #␣
↪needs sage.libs.ntl
sage: B.uniformizer()                                                    #␣
↪needs sage.libs.ntl
t + O(t^21)
```

uniformizer_pow(n)

Return the n-th power of the uniformizer of `self` (as an element of `self`).

EXAMPLES:

```
sage: A = Zp(7,10)
sage: S.<x> = A[]                                                        #␣
↪needs sage.libs.ntl
sage: B.<t> = A.ext(x^2 + 7)                                             #␣
↪needs sage.libs.ntl
sage: B.uniformizer_pow(5)                                               #␣
↪needs sage.libs.ntl
t^5 + O(t^25)
```

CHAPTER

NINE

UNRAMIFIED EXTENSION GENERIC

This file implements the shared functionality for unramified extensions.

AUTHORS:

- David Roe

class sage.rings.padics.unramified_extension_generic.**UnramifiedExtensionGeneric**(*poly*, *prec*, *print_mode*, *names*, *element_class*)

Bases: *pAdicExtensionGeneric*

An unramified extension of $\mathbf{Q}_p$ or $\mathbf{Z}_p$.

absolute_f()

Return the degree of the residue field of this ring/field over its prime subfield.

EXAMPLES:

```
sage: K.<a> = Qq(3^5)                                                    #␣
↪needs sage.libs.ntl
sage: K.absolute_f()                                                     #␣
↪needs sage.libs.ntl
5

sage: x = polygen(ZZ, 'x')
sage: L.<pi> = Qp(3).extension(x^2 - 3)                                  #␣
↪needs sage.libs.ntl
sage: L.absolute_f()                                                     #␣
↪needs sage.libs.ntl
1
```

discriminant(*K=None*)

Return the discriminant of `self` over the subring K.

INPUT:

- `K` – a subring/subfield (defaults to the base ring).

EXAMPLES:

```
sage: R.<a> = Zq(125)                                                    #␣
↪needs sage.libs.ntl
sage: R.discriminant()                                                   #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.libs.ntl
Traceback (most recent call last):
...
NotImplementedError
```

gen(*n=0*)

Return a generator for this unramified extension.

This is an element that satisfies the polynomial defining this extension. Such an element will reduce to a generator of the corresponding residue field extension.

EXAMPLES:

```
sage: R.<a> = Zq(125); R.gen()                                          #␣
↪needs sage.libs.ntl
a + O(5^20)
```

has_pth_root()

Return whether or not $\mathbf{Z}_p$ has a primitive p-th root of unity.

Since adjoining a p-th root of unity yields a totally ramified extension, `self` will contain one if and only if the ground ring does.

INPUT:

- `self` – a p-adic ring

OUTPUT:

boolean – whether `self` has primitive p-th root of unity.

EXAMPLES:

```
sage: R.<a> = Zq(1024); R.has_pth_root()                                #␣
↪needs sage.libs.ntl
True
sage: R.<a> = Zq(17^5); R.has_pth_root()                                #␣
↪needs sage.libs.ntl
False
```

has_root_of_unity(*n*)

Return whether or not $\mathbf{Z}_p$ has a primitive n-th root of unity.

INPUT:

- `self` – a p-adic ring
- `n` – an integer

OUTPUT:

- boolean

EXAMPLES:

```
sage: # needs sage.libs.ntl
sage: R.<a> = Zq(37^8)
sage: R.has_root_of_unity(144)
True
```

(continues on next page)

(continued from previous page)

```
sage: R.has_root_of_unity(89)
True
sage: R.has_root_of_unity(11)
False
```

is_galois(*K=None*)

Return `True` if this extension is Galois.

Every unramified extension is Galois.

INPUT:

- `K` – a subring/subfield (defaults to the base ring).

EXAMPLES:

```
sage: R.<a> = Zq(125); R.is_galois()                                    #␣
↪needs sage.libs.ntl
True
```

residue_class_field()

Returns the residue class field.

EXAMPLES:

```
sage: R.<a> = Zq(125); R.residue_class_field()                          #␣
↪needs sage.libs.ntl
Finite Field in a0 of size 5^3
```

residue_ring(n)

Return the quotient of the ring of integers by the n-th power of its maximal ideal.

EXAMPLES:

```
sage: R.<a> = Zq(125)                                                   #␣
↪needs sage.libs.ntl
sage: R.residue_ring(1)                                                 #␣
↪needs sage.libs.ntl
Finite Field in a0 of size 5^3
```

The following requires implementing more general Artinian rings:

```
sage: R.residue_ring(2)                                                 #␣
↪needs sage.libs.ntl
Traceback (most recent call last):
...
NotImplementedError
```

uniformizer()

Return a uniformizer for this extension.

Since this extension is unramified, a uniformizer for the ground ring will also be a uniformizer for this extension.

EXAMPLES:

```
sage: R.<a> = ZqCR(125)                                                     #␣
↪needs sage.libs.ntl
sage: R.uniformizer()                                                       #␣
↪needs sage.libs.ntl
5 + O(5^21)
```

uniformizer_pow(n)

Return the n-th power of the uniformizer of `self` (as an element of `self`).

EXAMPLES:

```
sage: R.<a> = ZqCR(125)                                                     #␣
↪needs sage.libs.ntl
sage: R.uniformizer_pow(5)                                                  #␣
↪needs sage.libs.ntl
5^5 + O(5^25)
```

CHAPTER

TEN

P-ADIC BASE LEAVES

Implementations of $\mathbf{Z}_p$ and $\mathbf{Q}_p$

AUTHORS:

- David Roe
- Genya Zaytman: documentation
- David Harvey: doctests
- William Stein: doctest updates

EXAMPLES:

p-adic rings and fields are examples of inexact structures, as the reals are. That means that elements cannot generally be stored exactly: to do so would take an infinite amount of storage. Instead, we store an approximation to the elements with varying precision.

There are two types of precision for a p-adic element. The first is relative precision, which gives the number of known p-adic digits:

```
sage: R = Qp(5, 20, 'capped-rel', 'series'); a = R(675); a
2*5^2 + 5^4 + O(5^22)
sage: a.precision_relative()
20
```

The second type of precision is absolute precision, which gives the power of p that this element is stored modulo:

```
sage: a.precision_absolute()
22
```

The number of times that p divides the element is called the valuation, and can be accessed with the methods `valuation()` and `ordp()`:

sage: a.valuation() 2

The following relationship holds:

`self.valuation() + self.precision_relative() == self.precision_absolute().`

sage: a.valuation() + a.precision_relative() == a.precision_absolute() True

In the capped relative case, the relative precision of an element is restricted to be at most a certain value, specified at the creation of the field. Individual elements also store their own precision, so the effect of various arithmetic operations on precision is tracked. When you cast an exact element into a capped relative field, it truncates it to the precision cap of the field.:

```
sage: R = Qp(5, 5); a = R(4006); a
1 + 5 + 2*5^3 + 5^4 + O(5^5)
sage: b = R(17/3); b
4 + 2*5 + 3*5^2 + 5^3 + 3*5^4 + O(5^5)
sage: c = R(4025); c
5^2 + 2*5^3 + 5^4 + 5^5 + O(5^7)
sage: a + b
4*5 + 3*5^2 + 3*5^3 + 4*5^4 + O(5^5)
sage: a + b + c
4*5 + 4*5^2 + 5^4 + O(5^5)
```

```
sage: R = Zp(5, 5, 'capped-rel', 'series'); a = R(4006); a
1 + 5 + 2*5^3 + 5^4 + O(5^5)
sage: b = R(17/3); b
4 + 2*5 + 3*5^2 + 5^3 + 3*5^4 + O(5^5)
sage: c = R(4025); c
5^2 + 2*5^3 + 5^4 + 5^5 + O(5^7)
sage: a + b
4*5 + 3*5^2 + 3*5^3 + 4*5^4 + O(5^5)
sage: a + b + c
4*5 + 4*5^2 + 5^4 + O(5^5)
```

In the capped absolute type, instead of having a cap on the relative precision of an element there is instead a cap on the absolute precision. Elements still store their own precisions, and as with the capped relative case, exact elements are truncated when cast into the ring.:

```
sage: R = ZpCA(5, 5); a = R(4005); a
5 + 2*5^3 + 5^4 + O(5^5)
sage: b = R(4025); b
5^2 + 2*5^3 + 5^4 + O(5^5)
sage: a * b
5^3 + 2*5^4 + O(5^5)
sage: (a * b) // 5^3
1 + 2*5 + O(5^2)
sage: type((a * b) // 5^3)
<class 'sage.rings.padics.padic_capped_absolute_element.pAdicCappedAbsoluteElement'>
sage: (a * b) / 5^3
1 + 2*5 + O(5^2)
sage: type((a * b) / 5^3)
<class 'sage.rings.padics.padic_capped_relative_element.pAdicCappedRelativeElement'>
```

The fixed modulus type is the leanest of the p-adic rings: it is basically just a wrapper around $\mathbf{Z}/p^n\mathbf{Z}$ providing a unified interface with the rest of the p-adics. This is the type you should use if your primary interest is in speed (though it's not all that much faster than other p-adic types). It does not track precision of elements.:

```
sage: R = ZpFM(5, 5); a = R(4005); a
5 + 2*5^3 + 5^4
sage: a // 5
1 + 2*5^2 + 5^3
```

p-adic rings and fields should be created using the creation functions `Zp()` and `Qp()` as above. This will ensure that there is only one instance of $\mathbf{Z}_p$ and $\mathbf{Q}_p$ of a given type, p, print mode and precision. It also saves typing very long class names.:

```
sage: Qp(17,10)
17-adic Field with capped relative precision 10
sage: R = Qp(7, prec = 20, print_mode = 'val-unit'); S = Qp(7, prec = 20, print_mode =
↪'val-unit'); R is S
True
sage: Qp(2)
2-adic Field with capped relative precision 20
```

Once one has a p-adic ring or field, one can cast elements into it in the standard way. Integers, ints, longs, Rationals, other p-adic types, pari p-adics and elements of $\mathbf{Z}/p^n\mathbf{Z}$ can all be cast into a p-adic field.:

```
sage: R = Qp(5, 5, 'capped-rel','series'); a = R(16); a
1 + 3*5 + O(5^5)
sage: b = R(23/15); b
5^-1 + 3 + 3*5 + 5^2 + 3*5^3 + O(5^4)
sage: S = Zp(5, 5, 'fixed-mod','val-unit'); c = S(Mod(75,125)); c
5^2 * 3
sage: R(c)
3*5^2 + O(5^5)
```

In the previous example, since fixed-mod elements don't keep track of their precision, we assume that it has the full precision of the ring. This is why you have to cast manually here.

While you can cast explicitly as above, the chains of automatic coercion are more restricted. As always in Sage, the following arrows are transitive and the diagram is commutative.:

```
int -> long -> Integer -> Zp capped-rel -> Zp capped_abs -> IntegerMod
Integer -> Zp fixed-mod -> IntegerMod
Integer -> Zp capped-abs -> Qp capped-rel
```

In addition, there are arrows within each type. For capped relative and capped absolute rings and fields, these arrows go from lower precision cap to higher precision cap. This works since elements track their own precision: choosing the parent with higher precision cap means that precision is less likely to be truncated unnecessarily. For fixed modulus parents, the arrow goes from higher precision cap to lower. The fact that elements do not track precision necessitates this choice in order to not produce incorrect results.

class sage.rings.padics.padic_base_leaves.**pAdicFieldCappedRelative**(*p*, *prec*, *print_mode*, *names*)

Bases: *pAdicFieldBaseGeneric*, *pAdicCappedRelativeFieldGeneric*

An implementation of p-adic fields with capped relative precision.

EXAMPLES:

```
sage: K = Qp(17, 1000000)  # indirect doctest
sage: K = Qp(101)  # indirect doctest
```

random_element(*algorithm='default'*)

Return a random element of `self`, optionally using the `algorithm` argument to decide how it generates the element. Algorithms currently implemented:

- `'default'`: Choose an integer k using the standard distribution on the integers. Then choose an integer a uniformly in the range $0 \leq a < p^N$ where N is the precision cap of `self`. Return `self(p^k * a, absprec = k + self.precision_cap())`.

EXAMPLES:

```
sage: Qp(17,6).random_element().parent() is Qp(17,6)
True
```

class sage.rings.padics.padic_base_leaves.**pAdicFieldFloatingPoint**(*p*, *prec*, *print_mode*, *names*)

Bases: *pAdicFieldBaseGeneric*, *pAdicFloatingPointFieldGeneric*

An implementation of the p-adic rationals with floating point precision.

class sage.rings.padics.padic_base_leaves.**pAdicFieldLattice**(*p*, *prec*, *subtype*, *print_mode*, *names*, *label=None*)

Bases: *pAdicLatticeGeneric*, *pAdicFieldBaseGeneric*

An implementation of the p-adic numbers with lattice precision.

INPUT:

- p – prime
- prec – precision cap, given as a pair (relative_cap, absolute_cap)
- subtype – either 'cap' or 'float'
- print_mode – dictionary with print options
- names – how to print the prime
- label – the label of this ring

See also:

label()

EXAMPLES:

```
sage: R = QpLC(next_prime(10^60))  # indirect doctest
doctest:...: FutureWarning: This class/method/function is marked as experimental.
It, its functionality or its interface might change without a formal deprecation.
See https://github.com/sagemath/sage/issues/23505 for details.
sage: type(R)
<class 'sage.rings.padics.padic_base_leaves.pAdicFieldLattice_with_category'>

sage: R = QpLC(2,label='init')  # indirect doctest
sage: R
2-adic Field with lattice-cap precision (label: init)
```

random_element(*prec=None*, *integral=False*)

Return a random element of this ring.

INPUT:

- prec – an integer or None (the default): the absolute precision of the generated random element
- integral – a boolean (default: False); if True, return an element in the ring of integers

EXAMPLES:

```
sage: K = QpLC(2)
sage: K.random_element()   # not tested, known bug (see :trac:`32126`)
2^-8 + 2^-7 + 2^-6 + 2^-5 + 2^-3 + 1 + 2^2 + 2^3 + 2^5 + O(2^12)
sage: K.random_element(integral=True)    # random
2^3 + 2^4 + 2^5 + 2^6 + 2^7 + 2^10 + 2^11 + 2^14 + 2^15 + 2^16
```

(continues on next page)

(continued from previous page)

```
 + 2^17 + 2^18 + 2^19 + O(2^20)

sage: K.random_element(prec=10)    # random
2^(-3) + 1 + 2 + 2^4 + 2^8 + O(2^10)
```

If the given precision is higher than the internal cap of the parent, then the cap is used:

```
sage: K.precision_cap_relative()
20
sage: K.random_element(prec=100)    # random
2^5 + 2^8 + 2^11 + 2^12 + 2^14 + 2^18 + 2^20 + 2^24 + O(2^25)
```

class sage.rings.padics.padic_base_leaves.**pAdicFieldRelaxed**(*p*, *prec*, *print_mode*, *names*)

Bases: *pAdicRelaxedGeneric*, *pAdicFieldBaseGeneric*

An implementation of relaxed arithmetics over $\mathbf{Q}_p$.

INPUT:

- p – prime
- prec – default precision
- print_mode – dictionary with print options
- names – how to print the prime

EXAMPLES:

```
sage: R = QpER(5)  # indirect doctest                                       #␣
↪needs sage.libs.flint
sage: type(R)                                                               #␣
↪needs sage.libs.flint
<class 'sage.rings.padics.padic_base_leaves.pAdicFieldRelaxed_with_category'>
```

class sage.rings.padics.padic_base_leaves.**pAdicRingCappedAbsolute**(*p*, *prec*, *print_mode*, *names*)

Bases: *pAdicRingBaseGeneric*, *pAdicCappedAbsoluteRingGeneric*

An implementation of the p-adic integers with capped absolute precision.

class sage.rings.padics.padic_base_leaves.**pAdicRingCappedRelative**(*p*, *prec*, *print_mode*, *names*)

Bases: *pAdicRingBaseGeneric*, *pAdicCappedRelativeRingGeneric*

An implementation of the p-adic integers with capped relative precision.

class sage.rings.padics.padic_base_leaves.**pAdicRingFixedMod**(*p*, *prec*, *print_mode*, *names*)

Bases: *pAdicRingBaseGeneric*, *pAdicFixedModRingGeneric*

An implementation of the p-adic integers using fixed modulus.

class sage.rings.padics.padic_base_leaves.**pAdicRingFloatingPoint**(*p*, *prec*, *print_mode*, *names*)

Bases: *pAdicRingBaseGeneric*, *pAdicFloatingPointRingGeneric*

An implementation of the p-adic integers with floating point precision.

class sage.rings.padics.padic_base_leaves.**pAdicRingLattice**(*p*, *prec*, *subtype*, *print_mode*, *names*, *label=None*)

Bases: *pAdicLatticeGeneric*, *pAdicRingBaseGeneric*

An implementation of the p-adic integers with lattice precision.

INPUT:

- p – prime
- prec – precision cap, given as a pair (relative_cap, absolute_cap)
- subtype – either 'cap' or 'float'
- print_mode – dictionary with print options
- names – how to print the prime
- label – the label of this ring

See also:

label()

EXAMPLES:

```
sage: R = ZpLC(next_prime(10^60))  # indirect doctest
doctest:...: FutureWarning: This class/method/function is marked as experimental.
It, its functionality or its interface might change without a formal deprecation.
See https://github.com/sagemath/sage/issues/23505 for details.
sage: type(R)
<class 'sage.rings.padics.padic_base_leaves.pAdicRingLattice_with_category'>

sage: R = ZpLC(2, label='init')  # indirect doctest
sage: R
2-adic Ring with lattice-cap precision (label: init)
```

random_element(*prec=None*)

Return a random element of this ring.

INPUT:

- prec – an integer or None (the default): the absolute precision of the generated random element

EXAMPLES:

```
sage: R = ZpLC(2)
sage: R.random_element()    # random
2^3 + 2^4 + 2^5 + 2^6 + 2^7 + 2^10 + 2^11 + 2^14 + 2^15 + 2^16
 + 2^17 + 2^18 + 2^19 + 2^21 + O(2^23)

sage: R.random_element(prec=10)    # random
1 + 2^3 + 2^4 + 2^7 + O(2^10)
```

class sage.rings.padics.padic_base_leaves.**pAdicRingRelaxed**(*p*, *prec*, *print_mode*, *names*)

Bases: *pAdicRelaxedGeneric*, *pAdicRingBaseGeneric*

An implementation of relaxed arithmetics over $\mathbf{Z}_p$.

INPUT:

- p – prime
- prec – default precision
- print_mode – dictionary with print options
- names – how to print the prime

EXAMPLES:

```
sage: R = ZpER(5)  # indirect doctest                                    # needs sage.libs.flint
sage: type(R)                                                            # needs sage.libs.flint
<class 'sage.rings.padics.padic_base_leaves.pAdicRingRelaxed_with_category'>
```

CHAPTER

ELEVEN

P-ADIC EXTENSION LEAVES

The final classes for extensions of $\mathbf{Z}_p$ and $\mathbf{Q}_p$ (i.e., classes that are not just designed to be inherited from).

AUTHORS:

- David Roe

class sage.rings.padics.padic_extension_leaves.**EisensteinExtensionFieldCappedRelative**(*exact_modulus*, *poly*, *prec*, *print_mode*, *shift_seed*, *names*, *implementation='NTL'*)

Bases: *EisensteinExtensionGeneric*, *pAdicCappedRelativeFieldGeneric*

class sage.rings.padics.padic_extension_leaves.**EisensteinExtensionRingCappedAbsolute**(*exact_modulus*, *poly*, *prec*, *print_mode*, *shift_seed*, *names*, *implementation*)

Bases: *EisensteinExtensionGeneric*, *pAdicCappedAbsoluteRingGeneric*

class sage.rings.padics.padic_extension_leaves.**EisensteinExtensionRingCappedRelative**(*exact_modulus*, *poly*, *prec*, *print_mode*, *shift_seed*, *names*, *implementation='NTL'*)

Bases: *EisensteinExtensionGeneric*, *pAdicCappedRelativeRingGeneric*

class sage.rings.padics.padic_extension_leaves.**EisensteinExtensionRingFixedMod**(*exact_modulus*, *poly*, *prec*, *print_mode*, *shift_seed*, *names*, *implementation='NTL'*)

Bases: *EisensteinExtensionGeneric*, *pAdicFixedModRingGeneric*

fraction_field()

Eisenstein extensions with fixed modulus do not support fraction fields.

EXAMPLES:

```
sage: S.<x> = ZZ[]
sage: R.<a> = ZpFM(5).extension(x^2 - 5)                                    # needs sage.libs.ntl
sage: R.fraction_field()                                                    # needs sage.libs.ntl
Traceback (most recent call last):
...
TypeError: This implementation of the p-adic ring
does not support fields of fractions.
```

class sage.rings.padics.padic_extension_leaves.**UnramifiedExtensionFieldCappedRelative**(*exact_modulus*, *poly*, *prec*, *print_mode*, *shift_seed*, *names*, *implementation='FLINT'*)

Bases: *UnramifiedExtensionGeneric*, *pAdicCappedRelativeFieldGeneric*

class sage.rings.padics.padic_extension_leaves.**UnramifiedExtensionFieldFloatingPoint**(*exact_modulus*, *poly*, *prec*, *print_mode*, *shift_seed*, *names*, *implementation='FLINT'*)

Bases: *UnramifiedExtensionGeneric*, *pAdicFloatingPointFieldGeneric*

class sage.rings.padics.padic_extension_leaves.**UnramifiedExtensionRingCappedAbsolute**(*exact_modulus*, *poly*, *prec*, *print_mode*, *shift_seed*, *names*, *implementation='FLINT'*)

Bases: *UnramifiedExtensionGeneric*, *pAdicCappedAbsoluteRingGeneric*

class sage.rings.padics.padic_extension_leaves.**UnramifiedExtensionRingCappedRelative**(*exact_modulus*, *poly*, *prec*, *print_mode*, *shift_seed*, *names*, *implementation='FLINT'*)

Bases: *UnramifiedExtensionGeneric*, *pAdicCappedRelativeRingGeneric*

class sage.rings.padics.padic_extension_leaves.**UnramifiedExtensionRingFixedMod**(*exact_modulus*, *poly*, *prec*, *print_mode*, *shift_seed*, *names*, *implementation='FLINT'*)

Bases: *UnramifiedExtensionGeneric*, *pAdicFixedModRingGeneric*

class sage.rings.padics.padic_extension_leaves.**UnramifiedExtensionRingFloatingPoint**(*exact_modulus*, *poly*, *prec*, *print_mode*, *shift_seed*, *names*, *implementation='FLINT'*)

Bases: *UnramifiedExtensionGeneric*, *pAdicFloatingPointRingGeneric*

CHAPTER

TWELVE

LOCAL GENERIC ELEMENT

This file contains a common superclass for p-adic elements and power series elements.

AUTHORS:

- David Roe: initial version
- Julian Rueth (2012-10-15, 2014-06-25, 2017-08-04): added inverse_of_unit(); improved add_bigoh(); added _test_expansion()

class sage.rings.padics.local_generic_element.**LocalGenericElement**

Bases: CommutativeRingElement

add_bigoh(*absprec*)

Return a copy of this element with absolute precision decreased to absprec.

INPUT:

- absprec – an integer or positive infinity

EXAMPLES:

```
sage: K = QpCR(3,4)
sage: o = K(1); o
1 + O(3^4)
sage: o.add_bigoh(2)
1 + O(3^2)
sage: o.add_bigoh(-5)
O(3^-5)
```

One cannot use add_bigoh to lift to a higher precision; this can be accomplished with lift_to_precision():

```
sage: o.add_bigoh(5)
1 + O(3^4)
```

Negative values of absprec return an element in the fraction field of the element's parent:

```
sage: R = ZpCA(3,4)
sage: R(3).add_bigoh(-5)
O(3^-5)
```

For fixed-mod elements this method truncates the element:

```
sage: R = ZpFM(3,4)
sage: R(3).add_bigoh(1)
0
```

If absprec exceeds the precision of the element, then this method has no effect:

```
sage: R(3).add_bigoh(5)
3
```

A negative value for absprec returns an element in the fraction field:

```
sage: R(3).add_bigoh(-1).parent()
3-adic Field with floating precision 4
```

euclidean_degree()

Return the degree of this element as an element of an Euclidean domain.

EXAMPLES:

For a field, this is always zero except for the zero element:

```
sage: K = Qp(2)
sage: K.one().euclidean_degree()
0
sage: K.gen().euclidean_degree()
0
sage: K.zero().euclidean_degree()
Traceback (most recent call last):
...
ValueError: euclidean degree not defined for the zero element
```

For a ring which is not a field, this is the valuation of the element:

```
sage: R = Zp(2)
sage: R.one().euclidean_degree()
0
sage: R.gen().euclidean_degree()
1
sage: R.zero().euclidean_degree()
Traceback (most recent call last):
...
ValueError: euclidean degree not defined for the zero element
```

inverse_of_unit()

Returns the inverse of self if self is a unit.

OUTPUT:

- an element in the same ring as self

EXAMPLES:

```
sage: R = ZpCA(3,5)
sage: a = R(2); a
2 + O(3^5)
```

(continues on next page)

(continued from previous page)

```
sage: b = a.inverse_of_unit(); b
2 + 3 + 3^2 + 3^3 + 3^4 + O(3^5)
```

A ZeroDivisionError is raised if an element has no inverse in the ring:

```
sage: R(3).inverse_of_unit()
Traceback (most recent call last):
...
ZeroDivisionError: inverse of 3 + O(3^5) does not exist
```

Unlike the usual inverse of an element, the result is in the same ring as self and not just in its fraction field:

```
sage: c = ~a; c
2 + 3 + 3^2 + 3^3 + 3^4 + O(3^5)
sage: a.parent()
3-adic Ring with capped absolute precision 5
sage: b.parent()
3-adic Ring with capped absolute precision 5
sage: c.parent()
3-adic Field with capped relative precision 5
```

For fields this does of course not make any difference:

```
sage: R = QpCR(3,5)
sage: a = R(2)
sage: b = a.inverse_of_unit()
sage: c = ~a
sage: a.parent()
3-adic Field with capped relative precision 5
sage: b.parent()
3-adic Field with capped relative precision 5
sage: c.parent()
3-adic Field with capped relative precision 5
```

is_integral()

Returns whether self is an integral element.

INPUT:

- self – a local ring element

OUTPUT:

- boolean – whether self is an integral element.

EXAMPLES:

```
sage: R = Qp(3,20)
sage: a = R(7/3); a.is_integral()
False
sage: b = R(7/5); b.is_integral()
True
```

is_padic_unit()

Returns whether self is a p-adic unit. That is, whether it has zero valuation.

INPUT:

- `self` – a local ring element

OUTPUT:

- boolean – whether `self` is a unit

EXAMPLES:

```
sage: R = Zp(3,20,'capped-rel'); K = Qp(3,20,'capped-rel')
sage: R(0).is_padic_unit()
False
sage: R(1).is_padic_unit()
True
sage: R(2).is_padic_unit()
True
sage: R(3).is_padic_unit()
False
sage: Qp(5,5)(5).is_padic_unit()
False
```

is_unit()

Returns whether self is a unit

INPUT:

- `self` – a local ring element

OUTPUT:

- boolean – whether `self` is a unit

Note: For fields all nonzero elements are units. For DVR's, only those elements of valuation 0 are. An older implementation ignored the case of fields, and returned always the negation of self.valuation()==0. This behavior is now supported with self.is_padic_unit().

EXAMPLES:

```
sage: R = Zp(3,20,'capped-rel'); K = Qp(3,20,'capped-rel')
sage: R(0).is_unit()
False
sage: R(1).is_unit()
True
sage: R(2).is_unit()
True
sage: R(3).is_unit()
False
sage: Qp(5,5)(5).is_unit()  # Note that 5 is invertible in `QQ_5`, even if it␣
↪has positive valuation!
True
sage: Qp(5,5)(5).is_padic_unit()
False
```

normalized_valuation()

Returns the normalized valuation of this local ring element, i.e., the valuation divided by the absolute ramification index.

INPUT:

`self` – a local ring element.

OUTPUT:

rational – the normalized valuation of `self`.

EXAMPLES:

```
sage: Q7 = Qp(7)
sage: R.<x> = Q7[]                                                          #␣
↪needs sage.libs.ntl
sage: F.<z> = Q7.ext(x^3+7*x+7)                                             #␣
↪needs sage.libs.ntl
sage: z.normalized_valuation()                                              #␣
↪needs sage.libs.ntl
1/3
```

quo_rem(*other*, *integral=False*)

Return the quotient with remainder of the division of this element by `other`.

INPUT:

- `other` – an element in the same ring
- `integral` – if True, use integral-style remainders even when the parent is a field. Namely, the remainder will have no terms in its p-adic expansion above the valuation of `other`.

EXAMPLES:

```
sage: R = Zp(3, 5)
sage: R(12).quo_rem(R(2))
(2*3 + O(3^6), 0)
sage: R(2).quo_rem(R(12))
(O(3^4), 2 + O(3^5))

sage: K = Qp(3, 5)
sage: K(12).quo_rem(K(2))
(2*3 + O(3^6), 0)
sage: K(2).quo_rem(K(12))
(2*3^-1 + 1 + 3 + 3^2 + 3^3 + O(3^4), 0)
```

You can get the same behavior for fields as for rings by using integral=True:

```
sage: K(12).quo_rem(K(2), integral=True)
(2*3 + O(3^6), 0)
sage: K(2).quo_rem(K(12), integral=True)
(O(3^4), 2 + O(3^5))
```

slice(*i*, *j*, *k=1*, *lift_mode='simple'*)

Returns the sum of the $pi^{i+l\cdot k}$ terms of the series expansion of this element, where pi is the uniformizer, for $i+l\cdot k$ between `i` and `j-1` inclusive, and nonnegative integers l. Behaves analogously to the slice function for lists.

INPUT:

- `i` – an integer; if set to `None`, the sum will start with the first non-zero term of the series.

- `j` – an integer; if set to `None` or ∞, this method behaves as if it was set to the absolute precision of this element.
- `k` – (default: 1) a positive integer

EXAMPLES:

```
sage: R = Zp(5, 6, 'capped-rel')
sage: a = R(1/2); a
3 + 2*5 + 2*5^2 + 2*5^3 + 2*5^4 + 2*5^5 + O(5^6)
sage: a.slice(2, 4)
2*5^2 + 2*5^3 + O(5^4)
sage: a.slice(1, 6, 2)
2*5 + 2*5^3 + 2*5^5 + O(5^6)
```

The step size `k` has to be positive:

```
sage: a.slice(0, 3, 0)
Traceback (most recent call last):
...
ValueError: slice step must be positive
sage: a.slice(0, 3, -1)
Traceback (most recent call last):
...
ValueError: slice step must be positive
```

If `i` exceeds `j`, then the result will be zero, with the precision given by `j`:

```
sage: a.slice(5, 4)
O(5^4)
sage: a.slice(6, 5)
O(5^5)
```

However, the precision cannot exceed the precision of the element:

```
sage: a.slice(101,100)
O(5^6)
sage: a.slice(0,5,2)
3 + 2*5^2 + 2*5^4 + O(5^5)
sage: a.slice(0,6,2)
3 + 2*5^2 + 2*5^4 + O(5^6)
sage: a.slice(0,7,2)
3 + 2*5^2 + 2*5^4 + O(5^6)
```

If start is left blank, it is set to the valuation:

```
sage: K = Qp(5, 6)
sage: x = K(1/25 + 5); x
5^-2 + 5 + O(5^4)
sage: x.slice(None, 3)
5^-2 + 5 + O(5^3)
sage: x[:3]
doctest:warning
...
DeprecationWarning: __getitem__ is changing to match the behavior of number␣
```

(continues on next page)

(continued from previous page)

```
↪fields. Please use expansion instead.
See https://github.com/sagemath/sage/issues/14825 for details.
5^-2 + 5 + O(5^3)
```

sqrt(*extend=True*, *all=False*, *algorithm=None*)

Return the square root of this element.

INPUT:

- `self` – a p-adic element.
- `extend` – a boolean (default: `True`); if `True`, return a square root in an extension if necessary; if `False` and no root exists in the given ring or field, raise a ValueError.
- `all` – a boolean (default: `False`); if `True`, return a list of all square roots.
- `algorithm` – `"pari"`, `"sage"` or `None` (default: `None`); Sage provides an implementation for any extension of Q_p whereas only square roots over Q_p is implemented in Pari; the default is `"pari"` if the ground field is Q_p, `"sage"` otherwise.

OUTPUT:

The square root or the list of all square roots of this element.

Note: The square root is chosen (resp. the square roots are ordered) in a deterministic way, which is compatible with change of precision.

EXAMPLES:

```
sage: R = Zp(3, 20)
sage: sqrt(R(0))
0

sage: sqrt(R(1))
1 + O(3^20)

sage: R(2).sqrt(extend=False)
Traceback (most recent call last):
...
ValueError: element is not a square

sage: s = sqrt(R(4)); -s
2 + O(3^20)

sage: s = sqrt(R(9)); s
3 + O(3^21)
```

Over the 2-adics, the precision of the square root is less than the input:

```
sage: R2 = Zp(2, 20)
sage: sqrt(R2(1))
1 + O(2^19)
sage: sqrt(R2(4))
2 + O(2^20)
```

(continues on next page)

(continued from previous page)

```
sage: R.<t> = Zq(2^10, 10)                                                    #␣
↪needs sage.libs.ntl
sage: u = 1 + 8*t                                                             #␣
↪needs sage.libs.ntl
sage: sqrt(u)                                                                 #␣
↪needs sage.libs.ntl
1 + t*2^2 + t^2*2^3 + t^2*2^4 + (t^4 + t^3 + t^2)*2^5 + (t^4 + t^2)*2^6
  + (t^5 + t^2)*2^7 + (t^6 + t^5 + t^4 + t^2)*2^8 + O(2^9)

sage: R.<a> = Zp(2).extension(x^3 - 2)
sage: u = R(1 + a^4 + a^5 + a^7 + a^8, 10); u
1 + a^4 + a^5 + a^7 + a^8 + O(a^10)
sage: v = sqrt(u); v                                                          #␣
↪needs sage.libs.ntl
1 + a^2 + a^4 + a^6 + O(a^7)
```

However, observe that the precision increases to its original value when we recompute the square of the square root:

```
sage: v^2                                                                     #␣
↪needs sage.libs.ntl
1 + a^4 + a^5 + a^7 + a^8 + O(a^10)
```

If the input does not have enough precision in order to determine if the given element has a square root in the ground field, an error is raised:

```
sage: R(1, 6).sqrt()
Traceback (most recent call last):
...
PrecisionError: not enough precision to be sure that this element has a square␣
↪root

sage: R(1, 7).sqrt()
1 + O(a^4)

sage: R(1+a^6, 7).sqrt(extend=False)
Traceback (most recent call last):
...
ValueError: element is not a square
```

In particular, an error is raised when we try to compute the square root of an inexact

CHAPTER

THIRTEEN

P-ADIC GENERIC ELEMENT

Elements of p-adic Rings and Fields

AUTHORS:

- David Roe
- Genya Zaytman: documentation
- David Harvey: doctests
- Julian Rueth: fixes for exp() and log(), implemented gcd, xgcd

sage.rings.padics.padic_generic_element.**dwork_mahler_coeffs**(*R*, *bd=20*)

Compute Dwork's formula for Mahler coefficients of p-adic Gamma.

This is called internally when one computes Gamma for a p-adic integer. Normally there is no need to call it directly.

INPUT:

- R – p-adic ring in which to compute
- bd – integer. Number of terms in the expansion to use

OUTPUT:

A list of p-adic integers.

EXAMPLES:

```
sage: from sage.rings.padics.padic_generic_element import dwork_mahler_coeffs,␣
↪evaluate_dwork_mahler
sage: R = Zp(3)
sage: v = dwork_mahler_coeffs(R)
sage: x = R(1/7)
sage: evaluate_dwork_mahler(v, x, 3, 20, 1)
2 + 2*3 + 3^2 + 3^3 + 3^4 + 3^5 + 2*3^6 + 2*3^7 + 2*3^8 + 2*3^9 + 2*3^11
 + 2*3^12 + 3^13 + 3^14 + 2*3^16 + 3^17 + 3^19 + O(3^20)
sage: x.dwork_expansion(a=1)  # Same result
2 + 2*3 + 3^2 + 3^3 + 3^4 + 3^5 + 2*3^6 + 2*3^7 + 2*3^8 + 2*3^9 + 2*3^11
 + 2*3^12 + 3^13 + 3^14 + 2*3^16 + 3^17 + 3^19 + O(3^20)
```

sage.rings.padics.padic_generic_element.**evaluate_dwork_mahler**(*v*, *x*, *p*, *bd*, *a*)

Evaluate Dwork's Mahler series for p-adic Gamma.

EXAMPLES:

```
sage: from sage.rings.padics.padic_generic_element import dwork_mahler_coeffs,␣
↪evaluate_dwork_mahler
sage: R = Zp(3)
sage: v = dwork_mahler_coeffs(R)
sage: x = R(1/7)
sage: evaluate_dwork_mahler(v, x, 3, 20, 1)
2 + 2*3 + 3^2 + 3^3 + 3^4 + 3^5 + 2*3^6 + 2*3^7 + 2*3^8 + 2*3^9 + 2*3^11
 + 2*3^12 + 3^13 + 3^14 + 2*3^16 + 3^17 + 3^19 + O(3^20)
sage: x.dwork_expansion(a=1)  # Same result
2 + 2*3 + 3^2 + 3^3 + 3^4 + 3^5 + 2*3^6 + 2*3^7 + 2*3^8 + 2*3^9 + 2*3^11
 + 2*3^12 + 3^13 + 3^14 + 2*3^16 + 3^17 + 3^19 + O(3^20)
```

sage.rings.padics.padic_generic_element.**gauss_table**(*p*, *f*, *prec*, *use_longs*)

Compute a table of Gauss sums using the Gross-Koblitz formula.

This is used in the computation of L-functions of hypergeometric motives. The Gross-Koblitz formula is used as in $sage.rings.padics.misc.gauss_sum$, but further unpacked for efficiency.

INPUT:

- p - prime
- f, prec - positive integers
- **use_longs - boolean; if True, computations are done in C long long**
  integers rather than Sage p-adics, and the results are returned as a Python array rather than a list.

OUTPUT:

A list of length $q - 1 = p^f - 1$. The entries are p-adic units created with absolute precision prec.

EXAMPLES:

```
sage: from sage.rings.padics.padic_generic_element import gauss_table
sage: gauss_table(2,2,4,False)
[1 + 2 + 2^2 + 2^3, 1 + 2 + 2^2 + 2^3, 1 + 2 + 2^2 + 2^3]
sage: gauss_table(3,2,4,False)[3]
2 + 3 + 2*3^2
```

class sage.rings.padics.padic_generic_element.**pAdicGenericElement**

Bases: *LocalGenericElement*

abs(*prec=None*)

Return the p-adic absolute value of self.

This is normalized so that the absolute value of p is $1/p$.

INPUT:

- prec – Integer. The precision of the real field in which the answer is returned. If None, returns a rational for absolutely unramified fields, or a real with 53 bits of precision for ramified fields.

EXAMPLES:

```
sage: a = Qp(5)(15); a.abs()
1/5
sage: a.abs(53)                                                        #␣
↪needs sage.rings.real_mpfr
0.200000000000000
```

(continues on next page)

(continued from previous page)

```
sage: Qp(7)(0).abs()
0
sage: Qp(7)(0).abs(prec=20)                                          #␣
↪needs sage.rings.real_mpfr
0.00000
```

An unramified extension:

```
sage: # needs sage.libs.ntl
sage: R = Zp(5,5)
sage: P.<x> = PolynomialRing(R)
sage: Z25.<u> = R.ext(x^2 - 3)
sage: u.abs()
1
sage: (u^24-1).abs()
1/5
```

A ramified extension:

```
sage: # needs sage.libs.ntl
sage: W.<w> = R.ext(x^5 + 75*x^3 - 15*x^2 + 125*x - 5)
sage: w.abs()
0.724779663677696
sage: W(0).abs()
0.000000000000000
```

additive_order(*prec=None*)

Return the additive order of this element truncated at precision `prec`.

INPUT:

- `prec` – an integer or `None` (default: `None`)

OUTPUT:

The additive order of this element

EXAMPLES:

```
sage: R = Zp(7, 4, 'capped-rel', 'series'); a = R(7^3); a.additive_order(3)
1
sage: a.additive_order(4)
+Infinity
sage: R = Zp(7, 4, 'fixed-mod', 'series'); a = R(7^5); a.additive_order(6)
1
```

algdep(*n*)

Returns a polynomial of degree at most n which is approximately satisfied by this number. Note that the returned polynomial need not be irreducible, and indeed usually won't be if this number is a good approximation to an algebraic number of degree less than n.

ALGORITHM: Uses the PARI C-library pari:algdep command.

INPUT:

- `self` – a p-adic element

- n – an integer

OUTPUT:

polynomial – degree n polynomial approximately satisfied by `self`

EXAMPLES:

```
sage: K = Qp(3,20,'capped-rel','series'); R = Zp(3,20,'capped-rel','series')
sage: a = K(7/19); a
1 + 2*3 + 3^2 + 3^3 + 2*3^4 + 2*3^5 + 3^8 + 2*3^9 + 3^11 + 3^12
  + 2*3^15 + 2*3^16 + 3^17 + 2*3^19 + O(3^20)
sage: a.algdep(1)
19*x - 7
sage: K2 = Qp(7,20,'capped-rel')
sage: b = K2.zeta(); b.algdep(2)
x^2 - x + 1
sage: K2 = Qp(11,20,'capped-rel')
sage: b = K2.zeta(); b.algdep(4)
x^4 - x^3 + x^2 - x + 1
sage: a = R(7/19); a
1 + 2*3 + 3^2 + 3^3 + 2*3^4 + 2*3^5 + 3^8 + 2*3^9 + 3^11 + 3^12
  + 2*3^15 + 2*3^16 + 3^17 + 2*3^19 + O(3^20)
sage: a.algdep(1)
19*x - 7
sage: R2 = Zp(7,20,'capped-rel')
sage: b = R2.zeta(); b.algdep(2)
x^2 - x + 1
sage: R2 = Zp(11,20,'capped-rel')
sage: b = R2.zeta(); b.algdep(4)
x^4 - x^3 + x^2 - x + 1
```

algebraic_dependency(n)

Returns a polynomial of degree at most n which is approximately satisfied by this number. Note that the returned polynomial need not be irreducible, and indeed usually won't be if this number is a good approximation to an algebraic number of degree less than n.

ALGORITHM: Uses the PARI C-library pari:algdep command.

INPUT:

- `self` – a p-adic element
- n – an integer

OUTPUT:

polynomial – degree n polynomial approximately satisfied by `self`

EXAMPLES:

```
sage: K = Qp(3,20,'capped-rel','series'); R = Zp(3,20,'capped-rel','series')
sage: a = K(7/19); a
1 + 2*3 + 3^2 + 3^3 + 2*3^4 + 2*3^5 + 3^8 + 2*3^9 + 3^11 + 3^12
  + 2*3^15 + 2*3^16 + 3^17 + 2*3^19 + O(3^20)
sage: a.algebraic_dependency(1)
19*x - 7
sage: K2 = Qp(7,20,'capped-rel')
```

(continues on next page)

(continued from previous page)

```
sage: b = K2.zeta(); b.algebraic_dependency(2)
x^2 - x + 1
sage: K2 = Qp(11,20,'capped-rel')
sage: b = K2.zeta(); b.algebraic_dependency(4)
x^4 - x^3 + x^2 - x + 1
sage: a = R(7/19); a
1 + 2*3 + 3^2 + 3^3 + 2*3^4 + 2*3^5 + 3^8 + 2*3^9 + 3^11 + 3^12
  + 2*3^15 + 2*3^16 + 3^17 + 2*3^19 + O(3^20)
sage: a.algebraic_dependency(1)
19*x - 7
sage: R2 = Zp(7,20,'capped-rel')
sage: b = R2.zeta(); b.algebraic_dependency(2)
x^2 - x + 1
sage: R2 = Zp(11,20,'capped-rel')
sage: b = R2.zeta(); b.algebraic_dependency(4)
x^4 - x^3 + x^2 - x + 1
```

artin_hasse_exp(*prec=None*, *algorithm=None*)

Return the Artin-Hasse exponential of this element.

INPUT:

- `prec` – an integer or `None` (default: `None`) the desired precision on the result; if `None`, the precision is derived from the precision on the input
- `algorithm` – `'direct'`, `'series'`, `'newton'` or `None` (default)

  The direct algorithm computes the Artin-Hasse exponential of x, namely `AH(x)` as

  $$AH(x) = \exp(x + \frac{x^p}{p} + \frac{x^{p^2}}{p^2} + \dots$$

  It runs roughly as fast as the computation of the exponential (since the computation of the argument is not that costly).

  The series algorithm computes the series defining the Artin-Hasse exponential and evaluates it.

  The `'newton'` algorithm solves the equation

  $$\log(AH(x)) = x + \frac{x^p}{p} + \frac{x^{p^2}}{p^2} + \dots$$

  using a Newton scheme. It runs roughly as fast as the computation of the logarithm.

  By default, we use the `'direct'` algorithm if a fast algorithm for computing the exponential is available. If not, we use the `'newton'` algorithm if a fast algorithm for computing the logarithm is available. Otherwise we switch to the `'series'` algorithm.

OUTPUT:

The Artin-Hasse exponential of this element.

See Wikipedia article Artin-Hasse_exponential for more information.

EXAMPLES:

```
sage: x = Zp(5)(45/7)
sage: y = x.artin_hasse_exp(); y
```

(continues on next page)

(continued from previous page)

```
1 + 2*5 + 4*5^2 + 3*5^3 + 5^7 + 2*5^8 + 3*5^10 + 2*5^11 + 2*5^12 +
2*5^13 + 5^14 + 3*5^17 + 2*5^18 + 2*5^19 + O(5^20)

sage: y * (-x).artin_hasse_exp()
1 + O(5^20)
```

The function respects your precision:

```
sage: x = Zp(3,30)(45/7)
sage: x.artin_hasse_exp()
1 + 2*3^2 + 3^4 + 2*3^5 + 3^6 + 2*3^7 + 2*3^8 + 3^9 + 2*3^10 + 3^11 +
3^13 + 2*3^15 + 2*3^16 + 2*3^17 + 3^19 + 3^20 + 2*3^21 + 3^23 + 3^24 +
3^26 + 3^27 + 2*3^28 + O(3^30)
```

Unless you tell it not to:

```
sage: x = Zp(3,30)(45/7)
sage: x.artin_hasse_exp()
1 + 2*3^2 + 3^4 + 2*3^5 + 3^6 + 2*3^7 + 2*3^8 + 3^9 + 2*3^10 + 3^11 +
3^13 + 2*3^15 + 2*3^16 + 2*3^17 + 3^19 + 3^20 + 2*3^21 + 3^23 + 3^24 +
3^26 + 3^27 + 2*3^28 + O(3^30)
sage: x.artin_hasse_exp(10)
1 + 2*3^2 + 3^4 + 2*3^5 + 3^6 + 2*3^7 + 2*3^8 + 3^9 + O(3^10)
```

For precision 1 the function just returns 1 since the exponential is always a 1-unit:

```
sage: x = Zp(3).random_element()
sage: while x.dist(0) >= 1:
....:     x = Zp(3).random_element()
sage: x.artin_hasse_exp(1)
1 + O(3)
```

AUTHORS:

- Mitchell Owen, Sebastian Pancrantz (2012-02): initial version.
- Xavier Caruso (2018-08): extend to any p-adic rings and fields and implement several algorithms.

dwork_expansion(*bd=20*, *a=0*)

Return the value of a function defined by Dwork.

Used to compute the p-adic Gamma function, see *gamma()*.

INPUT:

- bd – integer. Precision bound, defaults to 20
- a – integer. Offset parameter, defaults to 0

OUTPUT:

A p-adic integer.

Note: This is based on GP code written by Fernando Rodriguez Villegas (http://www.ma.utexas.edu/cnt/cnt-frames.html). William Stein sped it up for GP (http://sage.math.washington.edu/home/wstein/www/home/wbhart/pari-2.4.2.alpha/src/basemath/trans2.c). The output is a p-adic integer from Dwork's expansion, used to compute the p-adic gamma function as in [RV2007] section 6.2. The coefficients of the expansion are now cached to speed up multiple evaluation, as in the trace formula for hypergeometric motives.

EXAMPLES:

```
sage: R = Zp(17)
sage: x = R(5+3*17+13*17^2+6*17^3+12*17^5+10*17^(14)+5*17^(17)+O(17^(19)))
sage: x.dwork_expansion(18)
16 + 7*17 + 11*17^2 + 4*17^3 + 8*17^4 + 10*17^5 + 11*17^6 + 6*17^7
+ 17^8 + 8*17^10 + 13*17^11 + 9*17^12 + 15*17^13  + 2*17^14 + 6*17^15
+ 7*17^16 + 6*17^17 + O(17^18)

sage: R = Zp(5)
sage: x = R(3*5^2+4*5^3+1*5^4+2*5^5+1*5^(10)+O(5^(20)))
sage: x.dwork_expansion()
4 + 4*5 + 4*5^2 + 4*5^3 + 2*5^4 + 4*5^5 + 5^7 + 3*5^9 + 4*5^10 + 3*5^11
+ 5^13 + 4*5^14 + 2*5^15 + 2*5^16 + 2*5^17 + 3*5^18 + O(5^20)
```

exp(*aprec=None*, *algorithm=None*)

Compute the p-adic exponential of this element if the exponential series converges.

INPUT:

- `aprec` – an integer or `None` (default: `None`); if specified, computes only up to the indicated precision
- `algorithm` – `'generic'`, `'binary_splitting'`, `'newton'` or `None` (default)

  The `'generic'` algorithm evaluates naively the series defining the exponential, namely

  $$\exp(x) = 1 + x + x^2/2 + x^3/6 + x^4/24 + \cdots .$$

  Its binary complexity is quadratic with respect to the precision.

  The `'binary_splitting'` algorithm is faster, it has a quasi-linear complexity.

  The `'newton'` algorithms solve the equation $\log(x) = \texttt{self}$ using a Newton scheme. It runs roughly as fast as the computation of the logarithm.

  By default, we use the `'binary_splitting'` if it is available. If it is not, we use the `'newton'` algorithm if a fast algorithm for computing the logarithm is available. Otherwise we switch to the `'generic'` algorithm.

EXAMPLES:

log() and *exp()* are inverse to each other:

```
sage: Z13 = Zp(13, 10)
sage: a = Z13(14); a
1 + 13 + O(13^10)
sage: a.log().exp()
1 + 13 + O(13^10)
```

An error occurs if this is called with an element for which the exponential series does not converge:

```
sage: Z13.one().exp()
Traceback (most recent call last):
...
ValueError: Exponential does not converge for that input.
```

The next few examples illustrate precision when computing p-adic exponentials:

```
sage: R = Zp(5,10)
sage: e = R(2*5 + 2*5**2 + 4*5**3 + 3*5**4
....:         + 5**5 + 3*5**7 + 2*5**8 + 4*5**9).add_bigoh(10); e
2*5 + 2*5^2 + 4*5^3 + 3*5^4 + 5^5 + 3*5^7 + 2*5^8 + 4*5^9 + O(5^10)
sage: e.exp()*R.teichmuller(4)
4 + 2*5 + 3*5^3 + O(5^10)
```

```
sage: K = Qp(5,10)
sage: e = K(2*5 + 2*5**2 + 4*5**3 + 3*5**4
....:         + 5**5 + 3*5**7 + 2*5**8 + 4*5**9).add_bigoh(10); e
2*5 + 2*5^2 + 4*5^3 + 3*5^4 + 5^5 + 3*5^7 + 2*5^8 + 4*5^9 + O(5^10)
sage: e.exp()*K.teichmuller(4)
4 + 2*5 + 3*5^3 + O(5^10)
```

Logarithms and exponentials in extension fields. First, in an Eisenstein extension:

```
sage: # needs sage.libs.ntl
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^4 + 15*x^2 + 625*x - 5
sage: W.<w> = R.ext(f)
sage: z = 1 + w^2 + 4*w^7; z
1 + w^2 + 4*w^7 + O(w^20)
sage: z.log().exp()
1 + w^2 + 4*w^7 + O(w^20)
```

Now an unramified example:

```
sage: # needs sage.libs.ntl
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: g = x^3 + 3*x + 3
sage: A.<a> = R.ext(g)
sage: b = 1 + 5*(1 + a^2) + 5^3*(3 + 2*a); b
1 + (a^2 + 1)*5 + (2*a + 3)*5^3 + O(5^5)
sage: b.log().exp()
1 + (a^2 + 1)*5 + (2*a + 3)*5^3 + O(5^5)
```

AUTHORS:

- Genya Zaytman (2007-02-15)
- Amnon Besser, Marc Masdeu (2012-02-23): Complete rewrite
- Julian Rueth (2013-02-14): Added doctests, fixed some corner cases
- Xavier Caruso (2017-06): Added binary splitting and Newton algorithms

gamma(*algorithm='pari'*)

Return the value of the p-adic Gamma function.

INPUT:

- `algorithm` – string. Can be set to `'pari'` to call the PARI function, or `'sage'` to call the function implemented in Sage. The default is `'pari'` since PARI is about 10 times faster than Sage.

OUTPUT:

- a p-adic integer

Note: This is based on GP code written by Fernando Rodriguez Villegas (http://www.ma.utexas.edu/cnt/cnt-frames.html). William Stein sped it up for GP (http://sage.math.washington.edu/home/wstein/www/home/wbhart/pari-2.4.2.alpha/src/basemath/trans2.c). The `'sage'` version uses dwork_expansion() to compute the p-adic gamma function of self as in [RV2007] section 6.2.

EXAMPLES:

This example illustrates `x.gamma()` for x a p-adic unit:

```
sage: R = Zp(7)
sage: x = R(2+3*7^2+4*7^3+O(7^20))
sage: x.gamma('pari')
1 + 2*7^2 + 4*7^3 + 5*7^4 + 3*7^5 + 7^8 + 7^9 + 4*7^10 + 3*7^12
+ 7^13 + 5*7^14 + 3*7^15 + 2*7^16 + 2*7^17 + 5*7^18 + 4*7^19 + O(7^20)
sage: x.gamma('sage')
1 + 2*7^2 + 4*7^3 + 5*7^4 + 3*7^5 + 7^8 + 7^9 + 4*7^10 + 3*7^12
+ 7^13 + 5*7^14 + 3*7^15 + 2*7^16 + 2*7^17 + 5*7^18 + 4*7^19 + O(7^20)
sage: x.gamma('pari') == x.gamma('sage')
True
```

Now `x.gamma()` for x a p-adic integer but not a unit:

```
sage: R = Zp(17)
sage: x = R(17+17^2+3*17^3+12*17^8+O(17^13))
sage: x.gamma('pari')
1 + 12*17 + 13*17^2 + 13*17^3 + 10*17^4 + 7*17^5 + 16*17^7
+ 13*17^9 + 4*17^10 + 9*17^11 + 17^12 + O(17^13)
sage: x.gamma('sage')
1 + 12*17 + 13*17^2 + 13*17^3 + 10*17^4 + 7*17^5 + 16*17^7
+ 13*17^9 + 4*17^10 + 9*17^11 + 17^12 + O(17^13)
sage: x.gamma('pari') == x.gamma('sage')
True
```

Finally, this function is not defined if x is not a p-adic integer:

```
sage: K = Qp(7)
sage: x = K(7^-5 + 2*7^-4 + 5*7^-3 + 2*7^-2 + 3*7^-1 + 3 + 3*7
....:       + 7^3 + 4*7^4 + 5*7^5 + 6*7^8 + 3*7^9 + 6*7^10 + 5*7^11 + 6*7^12
....:       + 3*7^13 + 5*7^14 + O(7^15))
sage: x.gamma()
Traceback (most recent call last):
...
ValueError: The p-adic gamma function only works on elements of Zp
```

gcd(*other*)

Return a greatest common divisor of `self` and `other`.

INPUT:

- `other` – an element in the same ring as `self`

AUTHORS:

- Julian Rueth (2012-10-19): initial version

Note: Since the elements are only given with finite precision, their greatest common divisor is in general not unique (not even up to units). For example $O(3)$ is a representative for the elements 0 and 3 in the 3-adic ring $\mathbf{Z}_3$. The greatest common divisor of $O(3)$ and $O(3)$ could be (among others) 3 or 0 which have different valuation. The algorithm implemented here, will return an element of minimal valuation among the possible greatest common divisors.

EXAMPLES:

The greatest common divisor is either zero or a power of the uniformizing parameter:

```
sage: R = Zp(3)
sage: R.zero().gcd(R.zero())
0
sage: R(3).gcd(9)
3 + O(3^21)
```

A non-zero result is always lifted to the maximal precision possible in the ring:

```
sage: a = R(3,2); a
3 + O(3^2)
sage: b = R(9,3); b
3^2 + O(3^3)
sage: a.gcd(b)
3 + O(3^21)
sage: a.gcd(0)
3 + O(3^21)
```

If both elements are zero, then the result is zero with the precision set to the smallest of their precisions:

```
sage: a = R.zero(); a
0
sage: b = R(0,2); b
O(3^2)
sage: a.gcd(b)
O(3^2)
```

One could argue that it is mathematically correct to return $9 + O(3^{22})$ instead. However, this would lead to some confusing behaviour:

```
sage: alternative_gcd = R(9,22); alternative_gcd
3^2 + O(3^22)
sage: a.is_zero()
True
sage: b.is_zero()
True
sage: alternative_gcd.is_zero()
False
```

If exactly one element is zero, then the result depends on the valuation of the other element:

```
sage: R(0,3).gcd(3^4)
O(3^3)
```

(continues on next page)

(continued from previous page)

```
sage: R(0,4).gcd(3^4)
O(3^4)
sage: R(0,5).gcd(3^4)
3^4 + O(3^24)
```

Over a field, the greatest common divisor is either zero (possibly with finite precision) or one:

```
sage: K = Qp(3)
sage: K(3).gcd(0)
1 + O(3^20)
sage: K.zero().gcd(0)
0
sage: K.zero().gcd(K(0,2))
O(3^2)
sage: K(3).gcd(4)
1 + O(3^20)
```

is_prime()

Return whether this element is prime in its parent.

EXAMPLES:

```
sage: A = Zp(2)
sage: A(1).is_prime()
False
sage: A(2).is_prime()
True

sage: K = A.fraction_field()
sage: K(2).is_prime()
False
```

```
sage: # needs sage.libs.ntl
sage: x = polygen(ZZ, 'x')
sage: B.<pi> = A.extension(x^5 - 2)
sage: pi.is_prime()                                                   #␣
↪needs sage.symbolic
True
sage: B(2).is_prime()
False
```

is_square()

Returns whether this element is a square

INPUT:

- self – a p-adic element

EXAMPLES:

```
sage: R = Zp(3,20,'capped-rel')
sage: R(0).is_square()
True
sage: R(1).is_square()
```

(continues on next page)

(continued from previous page)

```
True
sage: R(2).is_square()
False
```

is_squarefree()

Return whether this element is squarefree, i.e., whether there exists no non-unit g such that g^2 divides this element.

EXAMPLES:

The zero element is never squarefree:

```
sage: K = Qp(2)
sage: K.zero().is_squarefree()
False
```

In p-adic rings, only elements of valuation at most 1 are squarefree:

```
sage: R = Zp(2)
sage: R(1).is_squarefree()
True
sage: R(2).is_squarefree()
True
sage: R(4).is_squarefree()
False
```

This works only if the precision is known sufficiently well:

```
sage: R(0,1).is_squarefree()
Traceback (most recent call last):
...
PrecisionError: element not known to sufficient precision to decide␣
↪squarefreeness
sage: R(0,2).is_squarefree()
False
sage: R(1,1).is_squarefree()
True
```

For fields we are not so strict about the precision and treat inexact zeros as the zero element:

```
K(0,0).is_squarefree()
False
```

log(*p_branch=None*, *pi_branch=None*, *aprec=None*, *change_frac=False*, *algorithm=None*)

Compute the p-adic logarithm of this element.

The usual power series for the logarithm with values in the additive group of a p-adic ring only converges for 1-units (units congruent to 1 modulo p). However, there is a unique extension of the logarithm to a homomorphism defined on all the units: If $u = a \cdot v$ is a unit with $v \equiv 1 \pmod{p}$ and a a Teichmuller representative, then we define $log(u) = log(v)$. This is the correct extension because the units U split as a product $U = V \times \langle w \rangle$, where V is the subgroup of 1-units and w is a fundamental root of unity. The $\langle w \rangle$ factor is torsion, so must go to 0 under any homomorphism to the fraction field, which is a torsion free group.

INPUT:

- p_branch – an element in the base ring or its fraction field; the implementation will choose the branch of the logarithm which sends p to branch
- pi_branch – an element in the base ring or its fraction field; the implementation will choose the branch of the logarithm which sends the uniformizer to branch; you may specify at most one of p_branch and pi_branch, and must specify one of them if this element is not a unit
- aprec – an integer or None (default: None); if not None, then the result will only be correct to precision aprec
- change_frac – In general the codomain of the logarithm should be in the p-adic field, however, for most neighborhoods of 1, it lies in the ring of integers. This flag decides if the codomain should be the same as the input (default) or if it should change to the fraction field of the input.
- algorithm – 'generic', 'binary_splitting' or None (default) The generic algorithm evaluates naively the series defining the log, namely

$$\log(1-x) = -x - 1/2x^2 - 1/3x^3 - 1/4x^4 - 1/5x^5 - \cdots.$$

  Its binary complexity is quadratic with respect to the precision.

  The 'binary_splitting' algorithm is faster, it has a quasi-linear complexity. By default, we use 'binary_splitting' if it is available. Otherwise we switch to the 'generic' algorithm.

Note: What some other systems do:

- PARI: Seems to define the logarithm for units not congruent to 1 as we do.
- MAGMA: Only implements logarithm for 1-units (version 2.19-2)

Todo: There is a soft-linear time algorithm for logarithm described by Dan Bernstein at http://cr.yp.to/lineartime/multapps-20041007.pdf

EXAMPLES:

```
sage: Z13 = Zp(13, 10)
sage: a = Z13(14); a
1 + 13 + O(13^10)
sage: a.log()
13 + 6*13^2 + 2*13^3 + 5*13^4 + 10*13^6 + 13^7 + 11*13^8 + 8*13^9 + O(13^10)

sage: Q13 = Qp(13, 10)
sage: a = Q13(14); a
1 + 13 + O(13^10)
sage: a.log()
13 + 6*13^2 + 2*13^3 + 5*13^4 + 10*13^6 + 13^7 + 11*13^8 + 8*13^9 + O(13^10)
```

Note that the relative precision decreases when we take log. Precisely the absolute precision on log(a) agrees with the relative precision on a thanks to the relation $d\log(a) = da/a$.

The call log(a) works as well:

```
sage: log(a)
13 + 6*13^2 + 2*13^3 + 5*13^4 + 10*13^6 + 13^7 + 11*13^8 + 8*13^9 + O(13^10)
sage: log(a) == a.log()
True
```

The logarithm is not only defined for 1-units:

```
sage: R = Zp(5,10)
sage: a = R(2)
sage: a.log()
2*5 + 3*5^2 + 2*5^3 + 4*5^4 + 2*5^6 + 2*5^7 + 4*5^8 + 2*5^9 + O(5^10)
```

If you want to take the logarithm of a non-unit you must specify either `p_branch` or `pi_branch`:

```
sage: b = R(5)
sage: b.log()
Traceback (most recent call last):
...
ValueError: you must specify a branch of the logarithm for non-units
sage: b.log(p_branch=4)
4 + O(5^10)
sage: c = R(10)
sage: c.log(p_branch=4)
4 + 2*5 + 3*5^2 + 2*5^3 + 4*5^4 + 2*5^6 + 2*5^7 + 4*5^8 + 2*5^9 + O(5^10)
```

The branch parameters are only relevant for elements of non-zero valuation:

```
sage: a.log(p_branch=0)
2*5 + 3*5^2 + 2*5^3 + 4*5^4 + 2*5^6 + 2*5^7 + 4*5^8 + 2*5^9 + O(5^10)
sage: a.log(p_branch=1)
2*5 + 3*5^2 + 2*5^3 + 4*5^4 + 2*5^6 + 2*5^7 + 4*5^8 + 2*5^9 + O(5^10)
```

Logarithms can also be computed in extension fields. First, in an Eisenstein extension:

```
sage: R = Zp(5,5)
sage: S.<x> = ZZ[]
sage: f = x^4 + 15*x^2 + 625*x - 5
sage: W.<w> = R.ext(f)                                                    #␣
↪needs sage.libs.ntl
sage: z = 1 + w^2 + 4*w^7; z                                              #␣
↪needs sage.libs.ntl
1 + w^2 + 4*w^7 + O(w^20)
sage: z.log()                                                             #␣
↪needs sage.libs.ntl
w^2 + 2*w^4 + 3*w^6 + 4*w^7 + w^9 + 4*w^10 + 4*w^11 + 4*w^12
 + 3*w^14 + w^15 + w^17 + 3*w^18 + 3*w^19 + O(w^20)
```

In an extension, there will usually be a difference between specifying `p_branch` and `pi_branch`:

```
sage: # needs sage.libs.ntl
sage: b = W(5)
sage: b.log()
Traceback (most recent call last):
...
ValueError: you must specify a branch of the logarithm for non-units
sage: b.log(p_branch=0)
O(w^20)
sage: b.log(p_branch=w)
w + O(w^20)
```

(continues on next page)

(continued from previous page)

```
sage: b.log(pi_branch=0)
3*w^2 + 2*w^4 + 2*w^6 + 3*w^8 + 4*w^10 + w^13 + w^14 + 2*w^15
 + 2*w^16 + w^18 + 4*w^19 + O(w^20)
sage: b.unit_part().log()
3*w^2 + 2*w^4 + 2*w^6 + 3*w^8 + 4*w^10 + w^13 + w^14 + 2*w^15
 + 2*w^16 + w^18 + 4*w^19 + O(w^20)
sage: y = w^2 * 4*w^7; y
4*w^9 + O(w^29)
sage: y.log(p_branch=0)
2*w^2 + 2*w^4 + 2*w^6 + 2*w^8 + w^10 + w^12 + 4*w^13 + 4*w^14 + 3*w^15
 + 4*w^16 + 4*w^17 + w^18 + 4*w^19 + O(w^20)
sage: y.log(p_branch=w)
w + 2*w^2 + 2*w^4 + 4*w^5 + 2*w^6 + 2*w^7 + 2*w^8 + 4*w^9 + w^10
 + 3*w^11 + w^12 + 4*w^14 + 4*w^16 + 2*w^17 + w^19 + O(w^20)
```

Check that log is multiplicative:

```
sage: y.log(p_branch=0) + z.log() - (y*z).log(p_branch=0)                    #␣
↪needs sage.libs.ntl
O(w^20)
```

Now an unramified example:

```
sage: # needs sage.libs.ntl
sage: g = x^3 + 3*x + 3
sage: A.<a> = R.ext(g)
sage: b = 1 + 5*(1 + a^2) + 5^3*(3 + 2*a)
sage: b.log()
(a^2 + 1)*5 + (3*a^2 + 4*a + 2)*5^2 + (3*a^2 + 2*a)*5^3
 + (3*a^2 + 2*a + 2)*5^4 + O(5^5)
```

Check that log is multiplicative:

```
sage: # needs sage.libs.ntl
sage: c = 3 + 5^2*(2 + 4*a)
sage: b.log() + c.log() - (b*c).log()
O(5^5)
```

We illustrate the effect of the `precision` argument:

```
sage: R = ZpCA(7,10)
sage: x = R(41152263); x
5 + 3*7^2 + 4*7^3 + 3*7^4 + 5*7^5 + 6*7^6 + 7^9 + O(7^10)
sage: x.log(aprec = 5)
7 + 3*7^2 + 4*7^3 + 3*7^4 + O(7^5)
sage: x.log(aprec = 7)
7 + 3*7^2 + 4*7^3 + 3*7^4 + 7^5 + 3*7^6 + O(7^7)
sage: x.log()
7 + 3*7^2 + 4*7^3 + 3*7^4 + 7^5 + 3*7^6 + 7^7 + 3*7^8 + 4*7^9 + O(7^10)
```

The logarithm is not defined for zero:

```
sage: R.zero().log()
Traceback (most recent call last):
...
ValueError: logarithm is not defined at zero
```

For elements in a p-adic ring, the logarithm will be returned in the same ring:

```
sage: x = R(2)
sage: x.log().parent()
7-adic Ring with capped absolute precision 10
sage: x = R(14)
sage: x.log(p_branch=0).parent()
7-adic Ring with capped absolute precision 10
```

This is not possible if the logarithm has negative valuation:

```
sage: R = ZpCA(3,10)
sage: S.<x> = R[]
sage: f = x^3 - 3
sage: W.<w> = R.ext(f)                                                  #␣
↪needs sage.libs.ntl sage.rings.padics
sage: w.log(p_branch=2)                                                 #␣
↪needs sage.libs.ntl sage.rings.padics
Traceback (most recent call last):
...
ValueError: logarithm is not integral, use change_frac=True
to obtain a result in the fraction field
sage: w.log(p_branch=2, change_frac=True)                               #␣
↪needs sage.libs.ntl sage.rings.padics
2*w^-3 + O(w^24)
```

AUTHORS:

- William Stein: initial version
- David Harvey (2006-09-13): corrected subtle precision bug (need to take denominators into account! – see github issue #53)
- Genya Zaytman (2007-02-14): adapted to new p-adic class
- Amnon Besser, Marc Masdeu (2012-02-21): complete rewrite, valid for generic p-adic rings.
- Soroosh Yazdani (2013-02-1): Fixed a precision issue in `_log_generic()`. This should really fix the issue with divisions.
- Julian Rueth (2013-02-14): Added doctests, some changes for capped-absolute implementations.
- Xavier Caruso (2017-06): Added binary splitting type algorithms over Qp

minimal_polynomial(*name='x'*, *base=None*)

Returns the minimal polynomial of this element over `base`

INPUT:

- `name` – string (default: `'x'`): the name of the variable
- `base` – a ring (default: the base ring of the parent): the base ring over which the minimal polynomial is computed

EXAMPLES:

```
sage: Zp(5,5)(1/3).minimal_polynomial('x')                                    #␣
↪needs sage.libs.ntl
(1 + O(5^5))*x + 3 + 5 + 3*5^2 + 5^3 + 3*5^4 + O(5^5)

sage: Zp(5,5)(1/3).minimal_polynomial('foo')                                  #␣
↪needs sage.libs.ntl
(1 + O(5^5))*foo + 3 + 5 + 3*5^2 + 5^3 + 3*5^4 + O(5^5)
```

```
sage: # needs sage.libs.ntl
sage: K.<a> = QqCR(2^3,5)
sage: S.<x> = K[]
sage: L.<pi> = K.extension(x^4 - 2*a)
sage: pi.minimal_polynomial()                                                 #␣
↪needs sage.symbolic
(1 + O(2^5))*x^4 + a*2 + a*2^2 + a*2^3 + a*2^4 + a*2^5 + O(2^6)
sage: (pi^2).minimal_polynomial()                                             #␣
↪needs sage.symbolic
(1 + O(2^5))*x^2 + a*2 + a*2^2 + a*2^3 + a*2^4 + a*2^5 + O(2^6)
sage: (1/pi).minimal_polynomial()                                             #␣
↪needs sage.symbolic
(1 + O(2^5))*x^4 + (a^2 + 1)*2^-1 + O(2^4)
sage: elt = L.random_element()
sage: P = elt.minimal_polynomial()  # not tested, known bug (see :trac:`32111`)
sage: P(elt) == 0  # not tested
True
```

multiplicative_order(*prec=None*)

Returns the multiplicative order of `self`, where `self` is considered to be one if it is one modulo p^{prec}.

INPUT:

- `self` – a p-adic element
- `prec` – an integer

OUTPUT:

- integer – the multiplicative order of `self`

EXAMPLES:

```
sage: K = Qp(5,20,'capped-rel')
sage: K(-1).multiplicative_order(20)
2
sage: K(1).multiplicative_order(20)
1
sage: K(2).multiplicative_order(20)
+Infinity
sage: K(5).multiplicative_order(20)
+Infinity
sage: K(1/5).multiplicative_order(20)
+Infinity
sage: K.zeta().multiplicative_order(20)
4
```

Over unramified extensions:

```
sage: # needs sage.libs.ntl
sage: L1.<a> = Qq(5^3)
sage: c = L1.teichmuller(a)
sage: c.multiplicative_order()
124
sage: c^124
1 + O(5^20)
```

Over totally ramified extensions:

```
sage: # needs sage.libs.ntl
sage: x = polygen(ZZ, 'x')
sage: L2.<pi> = Qp(5).extension(x^4 + 5*x^3 + 10*x^2 + 10*x + 5)
sage: u = 1 + pi
sage: u.multiplicative_order()
5
sage: v = L2.teichmuller(2)
sage: v.multiplicative_order()
4
sage: (u*v).multiplicative_order()
20
```

norm(*base=None*)

Return the norm of this p-adic element over `base`.

> **Warning:** This is not the p-adic absolute value. This is a field theoretic norm down to a base ring. If you want the p-adic absolute value, use the method `abs()` instead.

INPUT:

- `base` – a subring of the parent (default: base ring)

OUTPUT:

The norm of this p-adic element over the given base.

EXAMPLES:

```
sage: Zp(5)(5).norm()                                                    #␣
↪needs sage.libs.ntl
5 + O(5^21)
```

```
sage: # needs sage.libs.ntl
sage: K.<a> = QqCR(2^3,5)
sage: S.<x> = K[]
sage: L.<pi> = K.extension(x^4 - 2*a)
sage: pi.norm()  # norm over K                                           #␣
↪needs sage.symbolic
a*2 + a*2^2 + a*2^3 + a*2^4 + a*2^5 + O(2^6)
sage: (pi^2).norm()                                                      #␣
↪needs sage.symbolic
a^2*2^2 + O(2^7)
sage: pi.norm()^2                                                        #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.symbolic
a^2*2^2 + O(2^7)
```

nth_root(*n*, *all=False*)

Return the n-th root of this element.

INPUT:

- `n` – an integer
- `all` – a boolean (default: `False`): if `True`, return all n-th roots of this element, instead of just one.

EXAMPLES:

```
sage: A = Zp(5,10)
sage: x = A(61376); x
1 + 5^3 + 3*5^4 + 4*5^5 + 3*5^6 + O(5^10)
sage: y = x.nth_root(4); y
2 + 5 + 2*5^2 + 4*5^3 + 3*5^4 + 5^6 + O(5^10)
sage: y^4 == x
True

sage: x.nth_root(4, all=True)
[2 + 5 + 2*5^2 + 4*5^3 + 3*5^4 + 5^6 + O(5^10),
 4 + 4*5 + 4*5^2 + 4*5^4 + 3*5^5 + 5^6 + 3*5^7 + 5^8 + 5^9 + O(5^10),
 3 + 3*5 + 2*5^2 + 5^4 + 4*5^5 + 3*5^6 + 4*5^7 + 4*5^8 + 4*5^9 + O(5^10),
 1 + 4*5^3 + 5^5 + 3*5^6 + 5^7 + 3*5^8 + 3*5^9 + O(5^10)]
```

When n is divisible by the underlying prime p, we are losing precision (which is consistent with the fact that raising to the p-th power increases precision):

```
sage: z = x.nth_root(5); z
1 + 5^2 + 3*5^3 + 2*5^4 + 5^5 + 3*5^7 + 2*5^8 + O(5^9)
sage: z^5
1 + 5^3 + 3*5^4 + 4*5^5 + 3*5^6 + O(5^10)
```

Everything works over extensions as well:

```
sage: # needs sage.libs.ntl
sage: W.<a> = Zq(5^3)
sage: S.<x> = W[]
sage: R.<pi> = W.extension(x^7 - 5)
sage: R(5).nth_root(7)
pi + O(pi^141)
sage: R(5).nth_root(7, all=True)
[pi + O(pi^141)]
```

An error is raised if the given element is not an n-th power in the ring:

```
sage: R(5).nth_root(11)                                                      #␣
↪needs sage.libs.ntl
Traceback (most recent call last):
...
ValueError: this element is not a nth power
```

Similarly, when precision on the input is too small, an error is raised:

```
sage: # needs sage.libs.ntl
sage: x = R(1,6); x
1 + O(pi^6)
sage: x.nth_root(5)
Traceback (most recent call last):
...
PrecisionError: not enough precision to be sure that this element is a nth power
```

Check that github issue #30314 is fixed:

```
sage: # needs sage.libs.ntl
sage: K = Qp(29)
sage: x = polygen(K)
sage: L.<a> = K.extension(x^2 - 29)
sage: L(4).nth_root(2)
2 + O(a^40)
```

ordp(*p=None*)

Return the valuation of `self`, normalized so that the valuation of p is 1.

INPUT:

- `self` – a p-adic element
- `p` – a prime (default: `None`). If specified, will make sure that `p == self.parent().prime()`

Note: The optional argument p is used for consistency with the valuation methods on integers and rationals.

OUTPUT:

integer – the valuation of `self`, normalized so that the valuation of p is 1

EXAMPLES:

```
sage: R = Zp(5,20,'capped-rel')
sage: R(0).ordp()
+Infinity
sage: R(1).ordp()
0
sage: R(2).ordp()
0
sage: R(5).ordp()
1
sage: R(10).ordp()
1
sage: R(25).ordp()
2
sage: R(50).ordp()
2
sage: R(1/2).ordp()
0
```

polylog(*n*, *p_branch=0*)

Return $Li_n(\mathrm{self})$, the n-th p-adic polylogarithm of this element.

INPUT:

- n – a non-negative integer
- p_branch – an element in the base ring or its fraction field; the implementation will choose the branch of the logarithm which sends p to p_branch

EXAMPLES:

The n-th polylogarithm of -1 is 0 for even n:

```
sage: Qp(13)(-1).polylog(6) == 0                                              #␣
↪needs sage.rings.real_mpfr sage.symbolic
True
```

We can check some identities, for example those mentioned in [DCW2016]:

```
sage: x = Qp(7, prec=30)(1/3)
sage: (x^2).polylog(4) - 8*x.polylog(4) - 8*(-x).polylog(4) == 0              #␣
↪needs sage.symbolic
True
```

```
sage: x = Qp(5, prec=30)(4)
sage: x.polylog(2) + (1/x).polylog(2) + x.log(0)**2/2 == 0                    #␣
↪needs sage.symbolic
True
```

```
sage: x = Qp(11, prec=30)(2)
sage: x.polylog(2) + (1-x).polylog(2) + x.log(0)**2*(1-x).log(0) == 0         #␣
↪needs sage.symbolic
True
```

$Li_1(z) = -\log(1-z)$ for $|z| < 1$:

```
sage: Qp(5)(10).polylog(1) == -Qp(5)(1-10).log(0)
True
```

The dilogarithm of 1 is zero:

```
sage: Qp(5)(1).polylog(2)                                                     #␣
↪needs sage.rings.real_mpfr sage.symbolic
O(5^20)
```

The cubing relation holds for the trilogarithm at 1:

```
sage: K = Qp(7)
sage: z = K.zeta(3)
sage: -8*K(1).polylog(3) == 9*(K(z).polylog(3) + K(z^2).polylog(3))           #␣
↪needs sage.rings.padics sage.rings.real_mpfr sage.symbolic
True
```

The polylogarithm of 0 is 0:

```
sage: Qp(11)(0).polylog(7)
0
```

Only polylogarithms for positive n are defined:

```
sage: Qp(11)(2).polylog(-1)
Traceback (most recent call last):
...
ValueError: polylogarithm only implemented for n at least 0
```

Check that github issue #29222 is fixed:

```
sage: K = Qp(7)
sage: print(K(1 + 7^11).polylog(4))                                    #␣
↪needs sage.symbolic
6*7^14 + 3*7^15 + 7^16 + 7^17 + O(7^18)
```

ALGORITHM:

The algorithm of Besser-de Jeu, as described in [BdJ2008] is used.

AUTHORS:

- Jennifer Balakrishnan - Initial implementation
- Alex J. Best (2017-07-21) - Extended to other residue disks

---

Todo:

- Implement for extensions.
- Use the change method to create K from `self.parent()`.

---

rational_reconstruction()

Returns a rational approximation to this p-adic number.

This will raise an `ArithmeticError` if there are no valid approximations to the unit part with numerator and denominator bounded by `sqrt(p^absprec / 2)`.

See also:

`_rational_()`

OUTPUT:

rational – an approximation to `self`

EXAMPLES:

```
sage: R = Zp(5,20,'capped-rel')
sage: for i in range(11):
....:     for j in range(1,10):
....:         if j == 5:
....:             continue
....:         assert i/j == R(i/j).rational_reconstruction()
```

square_root(*extend=True*, *all=False*, *algorithm=None*)

Return the square root of this p-adic number.

INPUT:

- `self` – a p-adic element.
- `extend` – a boolean (default: `True`); if `True`, return a square root in an extension if necessary; if `False` and no root exists in the given ring or field, raise a `ValueError`.

- `all` – a boolean (default: `False`); if `True`, return a list of all square roots.
- `algorithm` – `"pari"`, `"sage"` or `None` (default: `None`); Sage provides an implementation for any extension of $\mathbf{Q}_p$, whereas only square roots over $\mathbf{Q}_p$ are implemented in PARI; the default is `"pari"` if the ground field is $\mathbf{Q}_p$, `"sage"` otherwise.

OUTPUT:

The square root or the list of all square roots of this p-adic number.

NOTE:

The square root is chosen (resp. the square roots are ordered) in a deterministic way.

EXAMPLES:

```
sage: R = Zp(3, 20)
sage: R(0).square_root()
0

sage: R(1).square_root()
1 + O(3^20)

sage: R(2).square_root(extend=False)
Traceback (most recent call last):
...
ValueError: element is not a square

sage: -R(4).square_root()
2 + O(3^20)

sage: R(9).square_root()
3 + O(3^21)
```

When $p = 2$, the precision of the square root is less than the input:

```
sage: R2 = Zp(2, 20)
sage: R2(1).square_root()
1 + O(2^19)
sage: R2(4).square_root()
2 + O(2^20)

sage: R.<t> = Zq(2^10, 10)                                                    #␣
↪needs sage.libs.ntl
sage: u = 1 + 8*t                                                             #␣
↪needs sage.libs.ntl
sage: u.square_root()                                                         #␣
↪needs sage.libs.ntl
1 + t*2^2 + t^2*2^3 + t^2*2^4 + (t^4 + t^3 + t^2)*2^5 + (t^4 + t^2)*2^6
 + (t^5 + t^2)*2^7 + (t^6 + t^5 + t^4 + t^2)*2^8 + O(2^9)

sage: # needs sage.libs.ntl
sage: x = polygen(ZZ, 'x')
sage: R.<a> = Zp(2).extension(x^3 - 2)
sage: u = R(1 + a^4 + a^5 + a^7 + a^8, 10); u
1 + a^4 + a^5 + a^7 + a^8 + O(a^10)
```

(continues on next page)

(continued from previous page)

```
sage: v = u.square_root(); v
1 + a^2 + a^4 + a^6 + O(a^7)
```

However, observe that the precision increases to its original value when we recompute the square of the square root:

```
sage: v^2                                                               #␣
↪needs sage.libs.ntl
1 + a^4 + a^5 + a^7 + a^8 + O(a^10)
```

If the input does not have enough precision in order to determine if the given element has a square root in the ground field, an error is raised:

```
sage: # needs sage.libs.ntl
sage: R(1, 6).square_root()
Traceback (most recent call last):
...
PrecisionError: not enough precision to be sure that this element has a square␣
↪root
sage: R(1, 7).square_root()
1 + O(a^4)
sage: R(1+a^6, 7).square_root(extend=False)
Traceback (most recent call last):
...
ValueError: element is not a square
```

In particular, an error is raised when we try to compute the square root of an inexact zero.

str(*mode=None*)

Return a string representation of `self`.

EXAMPLES:

```
sage: Zp(5,5,print_mode='bars')(1/3).str()[3:]
'1|3|1|3|2'
```

trace(*base=None*)

Returns the trace of this p-adic element over the base ring

INPUT:

- `base` – a subring of the parent (default: base ring)

OUTPUT:

The trace of this p-adic element over the given base.

EXAMPLES:

```
sage: Zp(5,5)(5).trace()                                                #␣
↪needs sage.libs.ntl
5 + O(5^6)

sage: # needs sage.libs.ntl
sage: K.<a> = QqCR(2^3,7)
sage: S.<x> = K[]
```

(continues on next page)

(continued from previous page)

```
sage: L.<pi> = K.extension(x^4 - 4*a*x^3 + 2*a)
sage: pi.trace()  # trace over K                                          #␣
↪needs sage.symbolic
a*2^2 + O(2^8)
sage: (pi+1).trace()                                                      #␣
↪needs sage.symbolic
(a + 1)*2^2 + O(2^7)
```

val_unit()

Return `(self.valuation(), self.unit_part())`. To be overridden in derived classes.

EXAMPLES:

```
sage: Zp(5,5)(5).val_unit()
(1, 1 + O(5^5))
```

valuation(*p=None*)

Return the valuation of this element.

INPUT:

- `self` – a p-adic element
- `p` – a prime (default: `None`). If specified, will make sure that `p == self.parent().prime()`

Note: The optional argument p is used for consistency with the valuation methods on integers and rationals.

OUTPUT:

integer – the valuation of `self`

EXAMPLES:

```
sage: R = Zp(17, 4,'capped-rel')
sage: a = R(2*17^2)
sage: a.valuation()
2
sage: R = Zp(5, 4,'capped-rel')
sage: R(0).valuation()
+Infinity
```

xgcd(*other*)

Compute the extended gcd of this element and `other`.

INPUT:

- `other` – an element in the same ring

OUTPUT:

A tuple `r`, `s`, `t` such that `r` is a greatest common divisor of this element and `other` and `r = s*self + t*other`.

AUTHORS:

- Julian Rueth (2012-10-19): initial version

Note: Since the elements are only given with finite precision, their greatest common divisor is in general not unique (not even up to units). For example $O(3)$ is a representative for the elements 0 and 3 in the 3-adic ring $\mathbf{Z}_3$. The greatest common divisor of $O(3)$ and $O(3)$ could be (among others) 3 or 0 which have different valuation. The algorithm implemented here, will return an element of minimal valuation among the possible greatest common divisors.

EXAMPLES:

The greatest common divisor is either zero or a power of the uniformizing parameter:

```
sage: R = Zp(3)
sage: R.zero().xgcd(R.zero())
(0, 1 + O(3^20), 0)
sage: R(3).xgcd(9)
(3 + O(3^21), 1 + O(3^20), 0)
```

Unlike for `gcd()`, the result is not lifted to the maximal precision possible in the ring; it is such that `r = s*self + t*other` holds true:

```
sage: a = R(3,2); a
3 + O(3^2)
sage: b = R(9,3); b
3^2 + O(3^3)
sage: a.xgcd(b)
(3 + O(3^2), 1 + O(3), 0)
sage: a.xgcd(0)
(3 + O(3^2), 1 + O(3), 0)
```

If both elements are zero, then the result is zero with the precision set to the smallest of their precisions:

```
sage: a = R.zero(); a
0
sage: b = R(0,2); b
O(3^2)
sage: a.xgcd(b)
(O(3^2), 0, 1 + O(3^20))
```

If only one element is zero, then the result depends on its precision:

```
sage: # needs sage.rings.padics
sage: R(9).xgcd(R(0,1))
(O(3), 0, 1 + O(3^20))
sage: R(9).xgcd(R(0,2))
(O(3^2), 0, 1 + O(3^20))
sage: R(9).xgcd(R(0,3))
(3^2 + O(3^22), 1 + O(3^20), 0)
sage: R(9).xgcd(R(0,4))
(3^2 + O(3^22), 1 + O(3^20), 0)
```

Over a field, the greatest common divisor is either zero (possibly with finite precision) or one:

```
sage: K = Qp(3)
sage: K(3).xgcd(0)
```

(continues on next page)

(continued from previous page)

```
(1 + O(3^20), 3^-1 + O(3^19), 0)
sage: K.zero().xgcd(0)
(0, 1 + O(3^20), 0)
sage: K.zero().xgcd(K(0,2))
(O(3^2), 0, 1 + O(3^20))
sage: K(3).xgcd(4)
(1 + O(3^20), 3^-1 + O(3^19), 0)
```

CHAPTER

FOURTEEN

P-ADIC CAPPED RELATIVE ELEMENTS

Elements of p-adic Rings with Capped Relative Precision

AUTHORS:

- David Roe: initial version, rewriting to use templates (2012-3-1)
- Genya Zaytman: documentation
- David Harvey: doctests

class sage.rings.padics.padic_capped_relative_element.**CRElement**

Bases: *pAdicTemplateElement*

add_bigoh(*absprec*)

Return a new element with absolute precision decreased to absprec.

INPUT:

- absprec – an integer or infinity

OUTPUT:

an equal element with precision set to the minimum of self's precision and absprec

EXAMPLES:

```
sage: R = Zp(7,4,'capped-rel','series'); a = R(8); a.add_bigoh(1)
1 + O(7)
sage: b = R(0); b.add_bigoh(3)
O(7^3)
sage: R = Qp(7,4); a = R(8); a.add_bigoh(1)
1 + O(7)
sage: b = R(0); b.add_bigoh(3)
O(7^3)
```

The precision never increases:

```
sage: R(4).add_bigoh(2).add_bigoh(4)
4 + O(7^2)
```

Another example that illustrates that the precision does not increase:

```
sage: k = Qp(3,5)
sage: a = k(1234123412/3^70); a
2*3^-70 + 3^-69 + 3^-68 + 3^-67 + O(3^-65)
sage: a.add_bigoh(2)
```

(continues on next page)

(continued from previous page)

```
2*3^-70 + 3^-69 + 3^-68 + 3^-67 + O(3^-65)

sage: k = Qp(5,10)
sage: a = k(1/5^3 + 5^2); a
5^-3 + 5^2 + O(5^7)
sage: a.add_bigoh(2)
5^-3 + O(5^2)
sage: a.add_bigoh(-1)
5^-3 + O(5^-1)
```

is_equal_to(*_right*, *absprec=None*)

Return whether `self` is equal to `right` modulo π^{absprec}.

If `absprec` is `None`, returns `True` if `self` and `right` are equal to the minimum of their precisions.

INPUT:

- `right` – a p-adic element
- `absprec` – an integer, infinity, or `None`

EXAMPLES:

```
sage: R = Zp(5, 10); a = R(0); b = R(0, 3); c = R(75, 5)
sage: aa = a + 625; bb = b + 625; cc = c + 625
sage: a.is_equal_to(aa), a.is_equal_to(aa, 4), a.is_equal_to(aa, 5)
(False, True, False)
sage: a.is_equal_to(aa, 15)
Traceback (most recent call last):
...
PrecisionError: elements not known to enough precision

sage: a.is_equal_to(a, 50000)
True

sage: a.is_equal_to(b), a.is_equal_to(b, 2)
(True, True)
sage: a.is_equal_to(b, 5)
Traceback (most recent call last):
...
PrecisionError: elements not known to enough precision

sage: b.is_equal_to(b, 5)
Traceback (most recent call last):
...
PrecisionError: elements not known to enough precision

sage: b.is_equal_to(bb, 3)
True
sage: b.is_equal_to(bb, 4)
Traceback (most recent call last):
...
PrecisionError: elements not known to enough precision
```

(continues on next page)

(continued from previous page)

```
sage: c.is_equal_to(b, 2), c.is_equal_to(b, 3)
(True, False)
sage: c.is_equal_to(b, 4)
Traceback (most recent call last):
...
PrecisionError: elements not known to enough precision

sage: c.is_equal_to(cc, 2), c.is_equal_to(cc, 4), c.is_equal_to(cc, 5)
(True, True, False)
```

is_zero(*absprec=None*)

Determine whether this element is zero modulo π^{absprec}.

If `absprec` is `None`, returns `True` if this element is indistinguishable from zero.

INPUT:

- `absprec` – an integer, infinity, or `None`

EXAMPLES:

```
sage: R = Zp(5); a = R(0); b = R(0,5); c = R(75)
sage: a.is_zero(), a.is_zero(6)
(True, True)
sage: b.is_zero(), b.is_zero(5)
(True, True)
sage: c.is_zero(), c.is_zero(2), c.is_zero(3)
(False, True, False)
sage: b.is_zero(6)
Traceback (most recent call last):
...
PrecisionError: not enough precision to determine if element is zero
```

polynomial(*var='x'*)

Return a polynomial over the base ring that yields this element when evaluated at the generator of the parent.

INPUT:

- `var` – string, the variable name for the polynomial

EXAMPLES:

```
sage: # needs sage.libs.ntl
sage: K.<a> = Qq(5^3)
sage: a.polynomial()
(1 + O(5^20))*x + O(5^20)
sage: a.polynomial(var='y')
(1 + O(5^20))*y + O(5^20)
sage: (5*a^2 + K(25, 4)).polynomial()
(5 + O(5^4))*x^2 + O(5^4)*x + 5^2 + O(5^4)
```

precision_absolute()

Returns the absolute precision of this element.

This is the power of the maximal ideal modulo which this element is defined.

EXAMPLES:

```
sage: R = Zp(7,3,'capped-rel'); a = R(7); a.precision_absolute()
4
sage: R = Qp(7,3); a = R(7); a.precision_absolute()
4
sage: R(7^-3).precision_absolute()
0

sage: R(0).precision_absolute()
+Infinity
sage: R(0,7).precision_absolute()
7
```

precision_relative()

Return the relative precision of this element.

This is the power of the maximal ideal modulo which the unit part of `self` is defined.

EXAMPLES:

```
sage: R = Zp(7,3,'capped-rel'); a = R(7); a.precision_relative()
3
sage: R = Qp(7,3); a = R(7); a.precision_relative()
3
sage: a = R(7^-2, -1); a.precision_relative()
1
sage: a
7^-2 + O(7^-1)

sage: R(0).precision_relative()
0
sage: R(0,7).precision_relative()
0
```

unit_part()

Return u, where this element is $\pi^v u$.

EXAMPLES:

```
sage: R = Zp(17,4,'capped-rel')
sage: a = R(18*17)
sage: a.unit_part()
1 + 17 + O(17^4)
sage: type(a)
<class 'sage.rings.padics.padic_capped_relative_element.
↪pAdicCappedRelativeElement'>
sage: R = Qp(17,4,'capped-rel')
sage: a = R(18*17)
sage: a.unit_part()
1 + 17 + O(17^4)
sage: type(a)
<class 'sage.rings.padics.padic_capped_relative_element.
↪pAdicCappedRelativeElement'>
sage: a = R(2*17^2); a
2*17^2 + O(17^6)
```

(continues on next page)

(continued from previous page)

```
sage: a.unit_part()
2 + O(17^4)
sage: b=1/a; b
9*17^-2 + 8*17^-1 + 8 + 8*17 + O(17^2)
sage: b.unit_part()
9 + 8*17 + 8*17^2 + 8*17^3 + O(17^4)
sage: Zp(5)(75).unit_part()
3 + O(5^20)

sage: R(0).unit_part()
Traceback (most recent call last):
...
ValueError: unit part of 0 not defined
sage: R(0,7).unit_part()
O(17^0)
```

val_unit(*p=None*)

Return a pair `(self.valuation(), self.unit_part())`.

INPUT:

- `p` – a prime (default: `None`). If specified, will make sure that `p == self.parent().prime()`

Note: The optional argument `p` is used for consistency with the valuation methods on integers and rationals.

EXAMPLES:

```
sage: R = Zp(5); a = R(75, 20); a
3*5^2 + O(5^20)
sage: a.val_unit()
(2, 3 + O(5^18))
sage: R(0).val_unit()
Traceback (most recent call last):
...
ValueError: unit part of 0 not defined
sage: R(0, 10).val_unit()
(10, O(5^0))
```

class sage.rings.padics.padic_capped_relative_element.**ExpansionIter**

Bases: `object`

An iterator over a p-adic expansion.

This class should not be instantiated directly, but instead using `expansion()`.

INPUT:

- `elt` – the p-adic element
- `prec` – the number of terms to be emitted
- `mode` – either `simple_mode`, `smallest_mode` or `teichmuller_mode`

EXAMPLES:

```
sage: E = Zp(5,4)(373).expansion()
sage: I = iter(E)  # indirect doctest
sage: type(I)
<class 'sage.rings.padics.padic_capped_relative_element.ExpansionIter'>
```

class sage.rings.padics.padic_capped_relative_element.**ExpansionIterable**

Bases: object

An iterable storing a p-adic expansion of an element.

This class should not be instantiated directly, but instead using expansion().

INPUT:

- elt – the p-adic element
- prec – the number of terms to be emitted
- val_shift – how many zeros to add at the beginning of the expansion, or the number of initial terms to truncate (if negative)
- mode – one of the following:
  - 'simple_mode'
  - 'smallest_mode'
  - 'teichmuller_mode'

EXAMPLES:

```
sage: E = Zp(5,4)(373).expansion()  # indirect doctest
sage: type(E)
<class 'sage.rings.padics.padic_capped_relative_element.ExpansionIterable'>
```

class sage.rings.padics.padic_capped_relative_element.**PowComputer_**

Bases: *PowComputer_base*

A PowComputer for a capped-relative padic ring or field.

sage.rings.padics.padic_capped_relative_element.**base_p_list**(*n*, *pos*, *prime_pow*)

Return a base-p list of digits of n.

INPUT:

- n – a positive Integer
- pos – a boolean; if True, then returns the standard base p expansion, otherwise the digits lie in the range $-p/2$ to $p/2$.
- prime_pow – a *PowComputer* giving the prime

EXAMPLES:

```
sage: from sage.rings.padics.padic_capped_relative_element import base_p_list
sage: base_p_list(192837, True, Zp(5).prime_pow)
[2, 2, 3, 2, 3, 1, 2, 2]
sage: 2 + 2*5 + 3*5^2 + 2*5^3 + 3*5^4 + 5^5 + 2*5^6 + 2*5^7
192837
sage: base_p_list(192837, False, Zp(5).prime_pow)
[2, 2, -2, -2, -1, 2, 2, 2]
```

(continues on next page)

(continued from previous page)

```
sage: 2 + 2*5 - 2*5^2 - 2*5^3 - 5^4 + 2*5^5 + 2*5^6 + 2*5^7
192837
```

class sage.rings.padics.padic_capped_relative_element.**pAdicCappedRelativeElement**

Bases: *CRElement*

Constructs new element with given parent and value.

INPUT:

- `x` – value to coerce into a capped relative ring or field
- `absprec` – maximum number of digits of absolute precision
- `relprec` – maximum number of digits of relative precision

EXAMPLES:

```
sage: R = Zp(5, 10, 'capped-rel')
```

Construct from integers:

```
sage: R(3)
3 + O(5^10)
sage: R(75)
3*5^2 + O(5^12)
sage: R(0)
0
sage: R(-1)
4 + 4*5 + 4*5^2 + 4*5^3 + 4*5^4 + 4*5^5 + 4*5^6 + 4*5^7 + 4*5^8 + 4*5^9 + O(5^10)
sage: R(-5)
4*5 + 4*5^2 + 4*5^3 + 4*5^4 + 4*5^5 + 4*5^6 + 4*5^7 + 4*5^8 + 4*5^9 + 4*5^10 + O(5^
↪11)
sage: R(-7*25)
3*5^2 + 3*5^3 + 4*5^4 + 4*5^5 + 4*5^6 + 4*5^7 + 4*5^8 + 4*5^9 + 4*5^10 + 4*5^11 +␣
↪O(5^12)
```

Construct from rationals:

```
sage: R(1/2)
3 + 2*5 + 2*5^2 + 2*5^3 + 2*5^4 + 2*5^5 + 2*5^6 + 2*5^7 + 2*5^8 + 2*5^9 + O(5^10)
sage: R(-7875/874)
3*5^3 + 2*5^4 + 2*5^5 + 5^6 + 3*5^7 + 2*5^8 + 3*5^10 + 3*5^11 + 3*5^12 + O(5^13)
sage: R(15/425)
Traceback (most recent call last):
...
ValueError: p divides the denominator
```

Construct from IntegerMod:

```
sage: R(Integers(125)(3))
3 + O(5^3)
sage: R(Integers(5)(3))
3 + O(5)
sage: R(Integers(5^30)(3))
```

(continues on next page)

(continued from previous page)

```
3 + O(5^10)
sage: R(Integers(5^30)(1+5^23))
1 + O(5^10)
sage: R(Integers(49)(3))
Traceback (most recent call last):
...
TypeError: p does not divide modulus 49
```

```
sage: R(Integers(48)(3))
Traceback (most recent call last):
...
TypeError: p does not divide modulus 48
```

Some other conversions:

```
sage: R(R(5))
5 + O(5^11)
```

Construct from Pari objects:

```
sage: R = Zp(5)
sage: x = pari(123123) ; R(x)                                          #␣
↪needs sage.libs.pari
3 + 4*5 + 4*5^2 + 4*5^3 + 5^4 + 4*5^5 + 2*5^6 + 5^7 + O(5^20)
sage: R(pari(R(5252)))
2 + 2*5^3 + 3*5^4 + 5^5 + O(5^20)
sage: R = Zp(5,prec=5)
sage: R(pari(-1))
4 + 4*5 + 4*5^2 + 4*5^3 + 4*5^4 + O(5^5)
sage: pari(R(-1))                                                      #␣
↪needs sage.libs.pari
4 + 4*5 + 4*5^2 + 4*5^3 + 4*5^4 + O(5^5)
sage: pari(R(0))                                                       #␣
↪needs sage.libs.pari
0
sage: R(pari(R(0,5)))
O(5^5)
```

Todo: doctests for converting from other types of p-adic rings

lift()

Return an integer or rational congruent to `self` modulo `self`'s precision. If a rational is returned, its denominator will equal `p^ordp(self)`.

EXAMPLES:

```
sage: R = Zp(7,4,'capped-rel'); a = R(8); a.lift()
8
sage: R = Qp(7,4); a = R(8); a.lift()
8
```

(continues on next page)

(continued from previous page)

```
sage: R = Qp(7,4); a = R(8/7); a.lift()
8/7
```

residue(*absprec=1*, *field=None*, *check_prec=True*)

Reduce this element modulo p^{absprec}.

INPUT:

- absprec – a non-negative integer (default: 1)
- field – boolean (default None); whether to return an element of $\mathbf{F}_p$ or $\mathbf{Z}/p\mathbf{Z}$
- check_prec – boolean (default True); whether to raise an error if this element has insufficient precision to determine the reduction

OUTPUT:

This element reduced modulo p^{absprec} as an element of $\mathbf{Z}/p^{\mathrm{absprec}}\mathbf{Z}$.

EXAMPLES:

```
sage: R = Zp(7,4)
sage: a = R(8)
sage: a.residue(1)
1
```

This is different from applying % p^n which returns an element in the same ring:

```
sage: b = a.residue(2); b
8
sage: b.parent()
Ring of integers modulo 49
sage: c = a % 7^2; c
1 + 7 + O(7^4)
sage: c.parent()
7-adic Ring with capped relative precision 4
```

For elements in a field, application of % p^n always returns zero, the remainder of the division by p^n:

```
sage: K = Qp(7,4)
sage: a = K(8)
sage: a.residue(2)
8
sage: a % 7^2
1 + 7 + O(7^4)

sage: b = K(1/7)
sage: b.residue()
Traceback (most recent call last):
...
ValueError: element must have non-negative valuation in order to compute residue
```

See also:

mod()

class sage.rings.padics.padic_capped_relative_element.**pAdicCoercion_CR_frac_field**

Bases: RingHomomorphism

The canonical inclusion of $\mathbf{Z}_q$ into its fraction field.

EXAMPLES:

```
sage: # needs sage.libs.flint
sage: R.<a> = ZqCR(27, implementation='FLINT')
sage: K = R.fraction_field()
sage: f = K.coerce_map_from(R); f
Ring morphism:
  From: 3-adic Unramified Extension Ring in a defined by x^3 + 2*x + 1
  To:   3-adic Unramified Extension Field in a defined by x^3 + 2*x + 1
```

is_injective()

Return whether this map is injective.

EXAMPLES:

```
sage: # needs sage.libs.flint
sage: R.<a> = ZqCR(9, implementation='FLINT')
sage: K = R.fraction_field()
sage: f = K.coerce_map_from(R)
sage: f.is_injective()
True
```

is_surjective()

Return whether this map is surjective.

EXAMPLES:

```
sage: # needs sage.libs.flint
sage: R.<a> = ZqCR(9, implementation='FLINT')
sage: K = R.fraction_field()
sage: f = K.coerce_map_from(R)
sage: f.is_surjective()
False
```

section()

Return a map back to the ring that converts elements of non-negative valuation.

EXAMPLES:

```
sage: # needs sage.libs.flint
sage: R.<a> = ZqCR(27, implementation='FLINT')
sage: K = R.fraction_field()
sage: f = K.coerce_map_from(R)
sage: f(K.gen())
a + O(3^20)
sage: f.section()
Generic morphism:
  From: 3-adic Unramified Extension Field in a defined by x^3 + 2*x + 1
  To:   3-adic Unramified Extension Ring in a defined by x^3 + 2*x + 1
```

class sage.rings.padics.padic_capped_relative_element.**pAdicCoercion_QQ_CR**

Bases: RingHomomorphism

The canonical inclusion from the rationals to a capped relative field.

EXAMPLES:

```
sage: f = Qp(5).coerce_map_from(QQ); f
Ring morphism:
  From: Rational Field
  To:   5-adic Field with capped relative precision 20
```

section()

Returns a map back to the rationals that approximates an element by a rational number.

EXAMPLES:

```
sage: f = Qp(5).coerce_map_from(QQ).section()
sage: f(Qp(5)(1/4))
1/4
sage: f(Qp(5)(1/5))
1/5
```

class sage.rings.padics.padic_capped_relative_element.**pAdicCoercion_ZZ_CR**

Bases: RingHomomorphism

The canonical inclusion from the integer ring to a capped relative ring.

EXAMPLES:

```
sage: f = Zp(5).coerce_map_from(ZZ); f
Ring morphism:
  From: Integer Ring
  To:   5-adic Ring with capped relative precision 20
```

section()

Returns a map back to the ring of integers that approximates an element by an integer.

EXAMPLES:

```
sage: f = Zp(5).coerce_map_from(ZZ).section()
sage: f(Zp(5)(-1)) - 5^20
-1
```

class sage.rings.padics.padic_capped_relative_element.**pAdicConvert_CR_QQ**

Bases: RingMap

The map from the capped relative ring back to the rationals that returns a rational approximation of its input.

EXAMPLES:

```
sage: f = Qp(5).coerce_map_from(QQ).section(); f
Set-theoretic ring morphism:
  From: 5-adic Field with capped relative precision 20
  To:   Rational Field
```

class sage.rings.padics.padic_capped_relative_element.**pAdicConvert_CR_ZZ**

Bases: RingMap

The map from a capped relative ring back to the ring of integers that returns the smallest non-negative integer approximation to its input which is accurate up to the precision.

Raises a ValueError, if the input is not in the closure of the image of the integers.

EXAMPLES:

```
sage: f = Zp(5).coerce_map_from(ZZ).section(); f
Set-theoretic ring morphism:
  From: 5-adic Ring with capped relative precision 20
  To:   Integer Ring
```

class sage.rings.padics.padic_capped_relative_element.**pAdicConvert_CR_frac_field**

Bases: Morphism

The section of the inclusion from $\mathbf{Z}_q$ to its fraction field.

EXAMPLES:

```
sage: # needs sage.libs.flint
sage: R.<a> = ZqCR(27, implementation='FLINT')
sage: K = R.fraction_field()
sage: f = R.convert_map_from(K); f
Generic morphism:
  From: 3-adic Unramified Extension Field in a defined by x^3 + 2*x + 1
  To:   3-adic Unramified Extension Ring in a defined by x^3 + 2*x + 1
```

class sage.rings.padics.padic_capped_relative_element.**pAdicConvert_QQ_CR**

Bases: Morphism

The inclusion map from the rationals to a capped relative ring that is defined on all elements with non-negative p-adic valuation.

EXAMPLES:

```
sage: f = Zp(5).convert_map_from(QQ); f
Generic morphism:
  From: Rational Field
  To:   5-adic Ring with capped relative precision 20
```

section()

Return the map back to the rationals that returns the smallest non-negative integer approximation to its input which is accurate up to the precision.

EXAMPLES:

```
sage: f = Zp(5,4).convert_map_from(QQ).section()
sage: f(Zp(5,4)(-1))
-1
```

class sage.rings.padics.padic_capped_relative_element.**pAdicTemplateElement**

Bases: *pAdicGenericElement*

A class for common functionality among the p-adic template classes.

INPUT:

- `parent` – a local ring or field
- `x` – data defining this element. Various types are supported, including ints, Integers, Rationals, PARI p-adics, integers mod p^k and other Sage p-adics.
- `absprec` – a cap on the absolute precision of this element
- `relprec` – a cap on the relative precision of this element

EXAMPLES:

```
sage: Zp(17)(17^3, 8, 4)
17^3 + O(17^7)
```

expansion(*n=None*, *lift_mode='simple'*, *start_val=None*)

Return the coefficients in a π-adic expansion. If this is a field element, start at $\pi^{\text{valuation}}$, if a ring element at π^0.

For each lift mode, this function returns a list of a_i so that this element can be expressed as

$$\pi^v \cdot \sum_{i=0}^{\infty} a_i \pi^i,$$

where v is the valuation of this element when the parent is a field, and $v = 0$ otherwise.

Different lift modes affect the choice of a_i. When `lift_mode` is `'simple'`, the resulting a_i will be non-negative: if the residue field is $\mathbf{F}_p$ then they will be integers with $0 \leq a_i < p$; otherwise they will be a list of integers in the same range giving the coefficients of a polynomial in the indeterminant representing the maximal unramified subextension.

Choosing `lift_mode` as `'smallest'` is similar to `'simple'`, but uses a balanced representation $-p/2 < a_i \leq p/2$.

Finally, setting `lift_mode = 'teichmuller'` will yield Teichmuller representatives for the a_i: $a_i^q = a_i$. In this case the a_i will lie in the ring of integers of the maximal unramified subextension of the parent of this element.

INPUT:

- `n` – integer (default `None`). If given, returns the corresponding entry in the expansion. Can also accept a slice (see `slice()`)
- `lift_mode` – `'simple'`, `'smallest'` or `'teichmuller'` (default: `'simple'`)
- `start_val` – start at this valuation rather than the default (0 or the valuation of this element).

OUTPUT:

- If `n` is `None`, an iterable giving a π-adic expansion of this element. For base elements the contents will be integers if `lift_mode` is `'simple'` or `'smallest'`, and elements of `self.parent()` if `lift_mode` is `'teichmuller'`.
- If `n` is an integer, the coefficient of π^n in the π-adic expansion of this element.

Note: Use slice operators to get a particular range.

EXAMPLES:

```
sage: R = Zp(7,6); a = R(12837162817); a
3 + 4*7 + 4*7^2 + 4*7^4 + O(7^6)
sage: E = a.expansion(); E
7-adic expansion of 3 + 4*7 + 4*7^2 + 4*7^4 + O(7^6)
sage: list(E)
[3, 4, 4, 0, 4, 0]
sage: sum([c * 7^i for i, c in enumerate(E)]) == a
True
sage: E = a.expansion(lift_mode='smallest'); E
7-adic expansion of 3 + 4*7 + 4*7^2 + 4*7^4 + O(7^6) (balanced)
sage: list(E)
[3, -3, -2, 1, -3, 1]
sage: sum([c * 7^i for i, c in enumerate(E)]) == a
True
sage: E = a.expansion(lift_mode='teichmuller'); E
7-adic expansion of 3 + 4*7 + 4*7^2 + 4*7^4 + O(7^6) (teichmuller)
sage: list(E)
[3 + 4*7 + 6*7^2 + 3*7^3 + 2*7^5 + O(7^6),
0,
5 + 2*7 + 3*7^3 + O(7^4),
1 + O(7^3),
3 + 4*7 + O(7^2),
5 + O(7)]
sage: sum(c * 7^i for i, c in enumerate(E))
3 + 4*7 + 4*7^2 + 4*7^4 + O(7^6)
```

If the element has positive valuation then the list will start with some zeros:

```
sage: a = R(7^3 * 17)
sage: E = a.expansion(); E
7-adic expansion of 3*7^3 + 2*7^4 + O(7^9)
sage: list(E)
[0, 0, 0, 3, 2, 0, 0, 0, 0]
```

The expansion of 0 is truncated:

```
sage: E = R(0, 7).expansion(); E
7-adic expansion of O(7^7)
sage: len(E)
0
sage: list(E)
[]
```

In fields, on the other hand, the expansion starts at the valuation:

```
sage: R = Qp(7,4); a = R(6*7+7**2); E = a.expansion(); E
7-adic expansion of 6*7 + 7^2 + O(7^5)
sage: list(E)
[6, 1, 0, 0]
sage: list(a.expansion(lift_mode='smallest'))
[-1, 2, 0, 0]
sage: list(a.expansion(lift_mode='teichmuller'))
[6 + 6*7 + 6*7^2 + 6*7^3 + O(7^4),
```

(continues on next page)

(continued from previous page)

```
2 + 4*7 + 6*7^2 + O(7^3),
3 + 4*7 + O(7^2),
3 + O(7)]
```

You can ask for a specific entry in the expansion:

```
sage: a.expansion(1)
6
sage: a.expansion(1, lift_mode='smallest')
-1
sage: a.expansion(2, lift_mode='teichmuller')
2 + 4*7 + 6*7^2 + O(7^3)
```

lift_to_precision(*absprec=None*)

Return another element of the same parent with absolute precision at least `absprec`, congruent to this p-adic element modulo the precision of this element.

INPUT:

- `absprec` – an integer or `None` (default: `None`); the absolute precision of the result. If `None`, lifts to the maximum precision allowed

---

Note: If setting `absprec` that high would violate the precision cap, raises a precision error. Note that the new digits will not necessarily be zero.

EXAMPLES:

```
sage: R = ZpCA(17)
sage: R(-1,2).lift_to_precision(10)
16 + 16*17 + O(17^10)
sage: R(1,15).lift_to_precision(10)
1 + O(17^15)
sage: R(1,15).lift_to_precision(30)
Traceback (most recent call last):
...
PrecisionError: precision higher than allowed by the precision cap
sage: R(-1,2).lift_to_precision().precision_absolute() == R.precision_cap()
True

sage: R = Zp(5); c = R(17,3); c.lift_to_precision(8)
2 + 3*5 + O(5^8)
sage: c.lift_to_precision().precision_relative() == R.precision_cap()
True
```

Fixed modulus elements don't raise errors:

```
sage: R = ZpFM(5); a = R(5); a.lift_to_precision(7)
5
sage: a.lift_to_precision(10000)
5
```

residue(*absprec=1*, *field=None*, *check_prec=True*)

Reduce this element modulo p^{absprec}.

INPUT:

- `absprec` – `0` or 1.
- `field` – boolean (default `None`). For precision 1, whether to return an element of the residue field or a residue ring. Currently unused.
- `check_prec` – boolean (default `True`). Whether to raise an error if this element has insufficient precision to determine the reduction. Errors are never raised for fixed-mod or floating-point types.

OUTPUT:

This element reduced modulo p^{absprec} as an element of the residue field or the null ring.

EXAMPLES:

```
sage: # needs sage.libs.ntl
sage: R.<a> = Zq(27, 4)
sage: (3 + 3*a).residue()
0
sage: (a + 1).residue()
a0 + 1
```

teichmuller_expansion(*n=None*)

Returns an iterator over coefficients $a_0, a_1, \ldots, a_n$ such that

- $a_i^q = a_i$, where q is the cardinality of the residue field,
- this element can be expressed as

$$\pi^v \cdot \sum_{i=0}^{\infty} a_i \pi^i$$

where v is the valuation of this element when the parent is a field, and $v = 0$ otherwise.

- if $a_i \neq 0$, the precision of a_i is i less than the precision of this element (relative in the case that the parent is a field, absolute otherwise)

Note: The coefficients will lie in the ring of integers of the maximal unramified subextension.

INPUT:

- `n` – integer (default `None`). If given, returns the coefficient of π^n in the expansion.

EXAMPLES:

For fields, the expansion starts at the valuation:

```
sage: R = Qp(5,5); list(R(70).teichmuller_expansion())
[4 + 4*5 + 4*5^2 + 4*5^3 + 4*5^4 + O(5^5),
3 + 3*5 + 2*5^2 + 3*5^3 + O(5^4),
2 + 5 + 2*5^2 + O(5^3),
1 + O(5^2),
4 + O(5)]
```

But if you specify `n`, you get the coefficient of π^n:

```
sage: R(70).teichmuller_expansion(2)
3 + 3*5 + 2*5^2 + 3*5^3 + O(5^4)
```

unit_part()

Returns the unit part of this element.

This is the p-adic element u in the same ring so that this element is $\pi^v u$, where π is a uniformizer and v is the valuation of this element.

EXAMPLES:

```
sage: # needs sage.libs.ntl
sage: R.<a> = Zq(125)
sage: (5*a).unit_part()
a + O(5^20)
```

sage.rings.padics.padic_capped_relative_element.**unpickle_cre_v2**(*cls*, *parent*, *unit*, *ordp*, *relprec*)

Unpickles a capped relative element.

EXAMPLES:

```
sage: from sage.rings.padics.padic_capped_relative_element import unpickle_cre_v2
sage: R = Zp(5); a = R(85,6)
sage: b = unpickle_cre_v2(a.__class__, R, 17, 1, 5)
sage: a == b
True
sage: a.precision_relative() == b.precision_relative()
True
```

sage.rings.padics.padic_capped_relative_element.**unpickle_pcre_v1**(*R*, *unit*, *ordp*, *relprec*)

Unpickles a capped relative element.

EXAMPLES:

```
sage: from sage.rings.padics.padic_capped_relative_element import unpickle_pcre_v1
sage: R = Zp(5)
sage: a = unpickle_pcre_v1(R, 17, 2, 5); a
2*5^2 + 3*5^3 + O(5^7)
```

CHAPTER

FIFTEEN

P-ADIC CAPPED ABSOLUTE ELEMENTS

Elements of p-adic Rings with Absolute Precision Cap

AUTHORS:

- David Roe
- Genya Zaytman: documentation
- David Harvey: doctests

class sage.rings.padics.padic_capped_absolute_element.**CAElement**

Bases: *pAdicTemplateElement*

add_bigoh(*absprec*)

Return a new element with absolute precision decreased to `absprec`. The precision never increases.

INPUT:

- `absprec` – an integer or infinity

OUTPUT:

`self` with precision set to the minimum of `self's` precision and `prec`

EXAMPLES:

```
sage: R = Zp(7,4,'capped-abs','series'); a = R(8); a.add_bigoh(1)
1 + O(7)

sage: k = ZpCA(3,5)
sage: a = k(41); a
2 + 3 + 3^2 + 3^3 + O(3^5)
sage: a.add_bigoh(7)
2 + 3 + 3^2 + 3^3 + O(3^5)
sage: a.add_bigoh(3)
2 + 3 + 3^2 + O(3^3)
```

is_equal_to(*_right*, *absprec=None*)

Determine whether the inputs are equal modulo π^{absprec}.

INPUT:

- `right` – a p-adic element with the same parent
- `absprec` – an integer, infinity, or `None`

EXAMPLES:

```
sage: R = ZpCA(2, 6)
sage: R(13).is_equal_to(R(13))
True
sage: R(13).is_equal_to(R(13+2^10))
True
sage: R(13).is_equal_to(R(17), 2)
True
sage: R(13).is_equal_to(R(17), 5)
False
sage: R(13).is_equal_to(R(13+2^10),absprec=10)
Traceback (most recent call last):
...
PrecisionError: elements not known to enough precision
```

is_zero(*absprec=None*)

Determine whether this element is zero modulo π^{absprec}.

If `absprec is None`, returns `True` if this element is indistinguishable from zero.

INPUT:

- `absprec` – an integer, infinity, or `None`

EXAMPLES:

```
sage: R = ZpCA(17, 6)
sage: R(0).is_zero()
True
sage: R(17^6).is_zero()
True
sage: R(17^2).is_zero(absprec=2)
True
sage: R(17^6).is_zero(absprec=10)
Traceback (most recent call last):
...
PrecisionError: not enough precision to determine if element is zero
```

polynomial(*var='x'*)

Return a polynomial over the base ring that yields this element when evaluated at the generator of the parent.

INPUT:

- `var` – string; the variable name for the polynomial

EXAMPLES:

```
sage: # needs sage.libs.flint
sage: R.<a> = ZqCA(5^3)
sage: a.polynomial()
(1 + O(5^20))*x + O(5^20)
sage: a.polynomial(var='y')
(1 + O(5^20))*y + O(5^20)
sage: (5*a^2 + R(25, 4)).polynomial()
(5 + O(5^4))*x^2 + O(5^4)*x + 5^2 + O(5^4)
```

precision_absolute()

The absolute precision of this element.

This is the power of the maximal ideal modulo which this element is defined.

EXAMPLES:

```
sage: R = Zp(7,4,'capped-abs'); a = R(7); a.precision_absolute()
4
```

precision_relative()

The relative precision of this element.

This is the power of the maximal ideal modulo which the unit part of this element is defined.

EXAMPLES:

```
sage: R = Zp(7,4,'capped-abs'); a = R(7); a.precision_relative()
3
```

unit_part()

Return the unit part of this element.

EXAMPLES:

```
sage: R = Zp(17,4,'capped-abs', 'val-unit')
sage: a = R(18*17)
sage: a.unit_part()
18 + O(17^3)
sage: type(a)
<class 'sage.rings.padics.padic_capped_absolute_element.
↪pAdicCappedAbsoluteElement'>
sage: R(0).unit_part()
O(17^0)
```

val_unit()

Return a 2-tuple, the first element set to the valuation of this element, and the second to the unit part of this element.

For a zero element, the unit part is `O(p^0)`.

EXAMPLES:

```
sage: R = ZpCA(5)
sage: a = R(75, 6); b = a - a
sage: a.val_unit()
(2, 3 + O(5^4))
sage: b.val_unit()
(6, O(5^0))
```

class sage.rings.padics.padic_capped_absolute_element.**ExpansionIter**

Bases: `object`

An iterator over a p-adic expansion.

This class should not be instantiated directly, but instead using `expansion()`.

INPUT:

- `elt` – the p-adic element
- `prec` – the number of terms to be emitted
- `mode` – either `simple_mode`, `smallest_mode` or `teichmuller_mode`

EXAMPLES:

```
sage: E = Zp(5,4)(373).expansion()
sage: I = iter(E)  # indirect doctest
sage: type(I)
<class 'sage.rings.padics.padic_capped_relative_element.ExpansionIter'>
```

class `sage.rings.padics.padic_capped_absolute_element.`**`ExpansionIterable`**

Bases: `object`

An iterable storing a p-adic expansion of an element.

This class should not be instantiated directly, but instead using `expansion()`.

INPUT:

- `elt` – the p-adic element
- `prec` – the number of terms to be emitted
- `val_shift` – how many zeros to add at the beginning of the expansion, or the number of initial terms to truncate (if negative)
- `mode` – one of the following:
  - `'simple_mode'`
  - `'smallest_mode'`
  - `'teichmuller_mode'`

EXAMPLES:

```
sage: E = Zp(5,4)(373).expansion()  # indirect doctest
sage: type(E)
<class 'sage.rings.padics.padic_capped_relative_element.ExpansionIterable'>
```

class `sage.rings.padics.padic_capped_absolute_element.`**`PowComputer_`**

Bases: *`PowComputer_base`*

A PowComputer for a capped-absolute padic ring.

`sage.rings.padics.padic_capped_absolute_element.`**`make_pAdicCappedAbsoluteElement`**(*parent*, *x*, *absprec*)

Unpickles a capped absolute element.

EXAMPLES:

```
sage: from sage.rings.padics.padic_capped_absolute_element import make_
↪pAdicCappedAbsoluteElement
sage: R = ZpCA(5)
sage: a = make_pAdicCappedAbsoluteElement(R, 17*25, 5); a
2*5^2 + 3*5^3 + O(5^5)
```

class sage.rings.padics.padic_capped_absolute_element.**pAdicCappedAbsoluteElement**

Bases: *CAElement*

Constructs new element with given parent and value.

INPUT:

- `x` – value to coerce into a capped absolute ring
- `absprec` – maximum number of digits of absolute precision
- `relprec` – maximum number of digits of relative precision

EXAMPLES:

```
sage: R = ZpCA(3, 5)
sage: R(2)
2 + O(3^5)
sage: R(2, absprec=2)
2 + O(3^2)
sage: R(3, relprec=2)
3 + O(3^3)
sage: R(Qp(3)(10))
1 + 3^2 + O(3^5)
sage: R(pari(6))
2*3 + O(3^5)
sage: R(pari(1/2))
2 + 3 + 3^2 + 3^3 + 3^4 + O(3^5)
sage: R(1/2)
2 + 3 + 3^2 + 3^3 + 3^4 + O(3^5)
sage: R(mod(-1, 3^7))
2 + 2*3 + 2*3^2 + 2*3^3 + 2*3^4 + O(3^5)
sage: R(mod(-1, 3^2))
2 + 2*3 + O(3^2)
sage: R(3 + O(3^2))
3 + O(3^2)
```

lift()

EXAMPLES:

```
sage: R = ZpCA(3)
sage: R(10).lift()
10
sage: R(-1).lift()
3486784400
```

multiplicative_order()

Return the minimum possible multiplicative order of this element.

OUTPUT:

The multiplicative order of self. This is the minimum multiplicative order of all elements of $\mathbf{Z}_p$ lifting `self` to infinite precision.

EXAMPLES:

```
sage: R = ZpCA(7, 6)
sage: R(1/3)
5 + 4*7 + 4*7^2 + 4*7^3 + 4*7^4 + 4*7^5 + O(7^6)
sage: R(1/3).multiplicative_order()
+Infinity
sage: R(7).multiplicative_order()
+Infinity
sage: R(1).multiplicative_order()
1
sage: R(-1).multiplicative_order()
2
sage: R.teichmuller(3).multiplicative_order()
6
```

residue(*absprec=1*, *field=None*, *check_prec=True*)

Reduces `self` modulo p^{absprec}.

INPUT:

- `absprec` – a non-negative integer (default: 1)
- `field` – boolean (default `None`). Whether to return an element of GF(p) or Zmod(p).
- `check_prec` – boolean (default `True`). Whether to raise an error if this element has insufficient precision to determine the reduction.

OUTPUT:

This element reduced modulo p^{absprec} as an element of $\mathbf{Z}/p^{\mathrm{absprec}}\mathbf{Z}$

EXAMPLES:

```
sage: R = Zp(7,10,'capped-abs')
sage: a = R(8)
sage: a.residue(1)
1
```

This is different from applying `% p^n` which returns an element in the same ring:

```
sage: b = a.residue(2); b
8
sage: b.parent()
Ring of integers modulo 49
sage: c = a % 7^2; c
1 + 7 + O(7^10)
sage: c.parent()
7-adic Ring with capped absolute precision 10
```

Note that reduction of `c` dropped to the precision of the unit part of `7^2`, see `_mod_()`:

```
sage: R(7^2).unit_part()
1 + O(7^8)
```

See also:

`_mod_()`

class sage.rings.padics.padic_capped_absolute_element.**pAdicCoercion_CA_frac_field**

Bases: RingHomomorphism

The canonical inclusion of Zq into its fraction field.

EXAMPLES:

```
sage: # needs sage.libs.flint
sage: R.<a> = ZqCA(27, implementation='FLINT')
sage: K = R.fraction_field()
sage: f = K.coerce_map_from(R); f
Ring morphism:
  From: 3-adic Unramified Extension Ring in a defined by x^3 + 2*x + 1
  To:   3-adic Unramified Extension Field in a defined by x^3 + 2*x + 1
```

is_injective()

Return whether this map is injective.

EXAMPLES:

```
sage: # needs sage.libs.flint
sage: R.<a> = ZqCA(9, implementation='FLINT')
sage: K = R.fraction_field()
sage: f = K.coerce_map_from(R)
sage: f.is_injective()
True
```

is_surjective()

Return whether this map is surjective.

EXAMPLES:

```
sage: # needs sage.libs.flint
sage: R.<a> = ZqCA(9, implementation='FLINT')
sage: K = R.fraction_field()
sage: f = K.coerce_map_from(R)
sage: f.is_surjective()
False
```

section()

Return a map back to the ring that converts elements of non-negative valuation.

EXAMPLES:

```
sage: # needs sage.libs.flint
sage: R.<a> = ZqCA(27, implementation='FLINT')
sage: K = R.fraction_field()
sage: f = K.coerce_map_from(R)
sage: f(K.gen())
a + O(3^20)
sage: f.section()
Generic morphism:
  From: 3-adic Unramified Extension Field in a defined by x^3 + 2*x + 1
  To:   3-adic Unramified Extension Ring in a defined by x^3 + 2*x + 1
```

class sage.rings.padics.padic_capped_absolute_element.**pAdicCoercion_ZZ_CA**

Bases: RingHomomorphism

The canonical inclusion from the ring of integers to a capped absolute ring.

EXAMPLES:

```
sage: f = ZpCA(5).coerce_map_from(ZZ); f
Ring morphism:
  From: Integer Ring
  To:   5-adic Ring with capped absolute precision 20
```

section()

Return a map back to the ring of integers that approximates an element by an integer.

EXAMPLES:

```
sage: f = ZpCA(5).coerce_map_from(ZZ).section()
sage: f(ZpCA(5)(-1)) - 5^20
-1
```

class sage.rings.padics.padic_capped_absolute_element.**pAdicConvert_CA_ZZ**

Bases: RingMap

The map from a capped absolute ring back to the ring of integers that returns the smallest non-negative integer approximation to its input which is accurate up to the precision.

Raises a ValueError if the input is not in the closure of the image of the ring of integers.

EXAMPLES:

```
sage: f = ZpCA(5).coerce_map_from(ZZ).section(); f
Set-theoretic ring morphism:
  From: 5-adic Ring with capped absolute precision 20
  To:   Integer Ring
```

class sage.rings.padics.padic_capped_absolute_element.**pAdicConvert_CA_frac_field**

Bases: Morphism

The section of the inclusion from $\mathbf{Z}_q$ to its fraction field.

EXAMPLES:

```
sage: # needs sage.libs.flint
sage: R.<a> = ZqCA(27, implementation='FLINT')
sage: K = R.fraction_field()
sage: f = R.convert_map_from(K); f
Generic morphism:
  From: 3-adic Unramified Extension Field in a defined by x^3 + 2*x + 1
  To:   3-adic Unramified Extension Ring in a defined by x^3 + 2*x + 1
```

class sage.rings.padics.padic_capped_absolute_element.**pAdicConvert_QQ_CA**

Bases: Morphism

The inclusion map from the rationals to a capped absolute ring that is defined on all elements with non-negative p-adic valuation.

EXAMPLES:

```
sage: f = ZpCA(5).convert_map_from(QQ); f
Generic morphism:
  From: Rational Field
  To:   5-adic Ring with capped absolute precision 20
```

class sage.rings.padics.padic_capped_absolute_element.**pAdicTemplateElement**

Bases: *pAdicGenericElement*

A class for common functionality among the p-adic template classes.

INPUT:

- `parent` – a local ring or field
- `x` – data defining this element. Various types are supported, including ints, Integers, Rationals, PARI p-adics, integers mod p^k and other Sage p-adics.
- `absprec` – a cap on the absolute precision of this element
- `relprec` – a cap on the relative precision of this element

EXAMPLES:

```
sage: Zp(17)(17^3, 8, 4)
17^3 + O(17^7)
```

expansion(*n=None*, *lift_mode='simple'*, *start_val=None*)

Return the coefficients in a π-adic expansion. If this is a field element, start at $\pi^{\mathrm{valuation}}$, if a ring element at π^0.

For each lift mode, this function returns a list of a_i so that this element can be expressed as

$$\pi^v \cdot \sum_{i=0}^{\infty} a_i \pi^i,$$

where v is the valuation of this element when the parent is a field, and $v = 0$ otherwise.

Different lift modes affect the choice of a_i. When `lift_mode` is `'simple'`, the resulting a_i will be non-negative: if the residue field is $\mathbf{F}_p$ then they will be integers with $0 \le a_i < p$; otherwise they will be a list of integers in the same range giving the coefficients of a polynomial in the indeterminant representing the maximal unramified subextension.

Choosing `lift_mode` as `'smallest'` is similar to `'simple'`, but uses a balanced representation $-p/2 < a_i \le p/2$.

Finally, setting `lift_mode = 'teichmuller'` will yield Teichmuller representatives for the a_i: $a_i^q = a_i$. In this case the a_i will lie in the ring of integers of the maximal unramified subextension of the parent of this element.

INPUT:

- `n` – integer (default `None`). If given, returns the corresponding entry in the expansion. Can also accept a slice (see `slice()`)
- `lift_mode` – `'simple'`, `'smallest'` or `'teichmuller'` (default: `'simple'`)
- `start_val` – start at this valuation rather than the default (0 or the valuation of this element).

OUTPUT:

- If n is None, an iterable giving a π-adic expansion of this element. For base elements the contents will be integers if lift_mode is 'simple' or 'smallest', and elements of self.parent() if lift_mode is 'teichmuller'.
- If n is an integer, the coefficient of π^n in the π-adic expansion of this element.

Note: Use slice operators to get a particular range.

EXAMPLES:

```
sage: R = Zp(7,6); a = R(12837162817); a
3 + 4*7 + 4*7^2 + 4*7^4 + O(7^6)
sage: E = a.expansion(); E
7-adic expansion of 3 + 4*7 + 4*7^2 + 4*7^4 + O(7^6)
sage: list(E)
[3, 4, 4, 0, 4, 0]
sage: sum([c * 7^i for i, c in enumerate(E)]) == a
True
sage: E = a.expansion(lift_mode='smallest'); E
7-adic expansion of 3 + 4*7 + 4*7^2 + 4*7^4 + O(7^6) (balanced)
sage: list(E)
[3, -3, -2, 1, -3, 1]
sage: sum([c * 7^i for i, c in enumerate(E)]) == a
True
sage: E = a.expansion(lift_mode='teichmuller'); E
7-adic expansion of 3 + 4*7 + 4*7^2 + 4*7^4 + O(7^6) (teichmuller)
sage: list(E)
[3 + 4*7 + 6*7^2 + 3*7^3 + 2*7^5 + O(7^6),
0,
5 + 2*7 + 3*7^3 + O(7^4),
1 + O(7^3),
3 + 4*7 + O(7^2),
5 + O(7)]
sage: sum(c * 7^i for i, c in enumerate(E))
3 + 4*7 + 4*7^2 + 4*7^4 + O(7^6)
```

If the element has positive valuation then the list will start with some zeros:

```
sage: a = R(7^3 * 17)
sage: E = a.expansion(); E
7-adic expansion of 3*7^3 + 2*7^4 + O(7^9)
sage: list(E)
[0, 0, 0, 3, 2, 0, 0, 0, 0]
```

The expansion of 0 is truncated:

```
sage: E = R(0, 7).expansion(); E
7-adic expansion of O(7^7)
sage: len(E)
0
sage: list(E)
[]
```

In fields, on the other hand, the expansion starts at the valuation:

```
sage: R = Qp(7,4); a = R(6*7+7**2); E = a.expansion(); E
7-adic expansion of 6*7 + 7^2 + O(7^5)
sage: list(E)
[6, 1, 0, 0]
sage: list(a.expansion(lift_mode='smallest'))
[-1, 2, 0, 0]
sage: list(a.expansion(lift_mode='teichmuller'))
[6 + 6*7 + 6*7^2 + 6*7^3 + O(7^4),
2 + 4*7 + 6*7^2 + O(7^3),
3 + 4*7 + O(7^2),
3 + O(7)]
```

You can ask for a specific entry in the expansion:

```
sage: a.expansion(1)
6
sage: a.expansion(1, lift_mode='smallest')
-1
sage: a.expansion(2, lift_mode='teichmuller')
2 + 4*7 + 6*7^2 + O(7^3)
```

lift_to_precision(*absprec=None*)

Return another element of the same parent with absolute precision at least `absprec`, congruent to this p-adic element modulo the precision of this element.

INPUT:

- `absprec` – an integer or `None` (default: `None`); the absolute precision of the result. If `None`, lifts to the maximum precision allowed

---

Note: If setting `absprec` that high would violate the precision cap, raises a precision error. Note that the new digits will not necessarily be zero.

EXAMPLES:

```
sage: R = ZpCA(17)
sage: R(-1,2).lift_to_precision(10)
16 + 16*17 + O(17^10)
sage: R(1,15).lift_to_precision(10)
1 + O(17^15)
sage: R(1,15).lift_to_precision(30)
Traceback (most recent call last):
...
PrecisionError: precision higher than allowed by the precision cap
sage: R(-1,2).lift_to_precision().precision_absolute() == R.precision_cap()
True

sage: R = Zp(5); c = R(17,3); c.lift_to_precision(8)
2 + 3*5 + O(5^8)
sage: c.lift_to_precision().precision_relative() == R.precision_cap()
True
```

Fixed modulus elements don't raise errors:

```
sage: R = ZpFM(5); a = R(5); a.lift_to_precision(7)
5
sage: a.lift_to_precision(10000)
5
```

residue(*absprec=1*, *field=None*, *check_prec=True*)

Reduce this element modulo p^{absprec}.

INPUT:

- `absprec` – `0` or 1.
- `field` – boolean (default `None`). For precision 1, whether to return an element of the residue field or a residue ring. Currently unused.
- `check_prec` – boolean (default `True`). Whether to raise an error if this element has insufficient precision to determine the reduction. Errors are never raised for fixed-mod or floating-point types.

OUTPUT:

This element reduced modulo p^{absprec} as an element of the residue field or the null ring.

EXAMPLES:

```
sage: # needs sage.libs.ntl
sage: R.<a> = Zq(27, 4)
sage: (3 + 3*a).residue()
0
sage: (a + 1).residue()
a0 + 1
```

teichmuller_expansion(*n=None*)

Returns an iterator over coefficients $a_0, a_1, \ldots, a_n$ such that

- $a_i^q = a_i$, where q is the cardinality of the residue field,
- this element can be expressed as

$$\pi^v \cdot \sum_{i=0}^{\infty} a_i \pi^i$$

where v is the valuation of this element when the parent is a field, and $v = 0$ otherwise.

- if $a_i \neq 0$, the precision of a_i is i less than the precision of this element (relative in the case that the parent is a field, absolute otherwise)

Note: The coefficients will lie in the ring of integers of the maximal unramified subextension.

INPUT:

- `n` – integer (default `None`). If given, returns the coefficient of π^n in the expansion.

EXAMPLES:

For fields, the expansion starts at the valuation:

```
sage: R = Qp(5,5); list(R(70).teichmuller_expansion())
[4 + 4*5 + 4*5^2 + 4*5^3 + 4*5^4 + O(5^5),
3 + 3*5 + 2*5^2 + 3*5^3 + O(5^4),
```

(continues on next page)

(continued from previous page)

```
2 + 5 + 2*5^2 + O(5^3),
1 + O(5^2),
4 + O(5)]
```

But if you specify n, you get the coefficient of π^n:

```
sage: R(70).teichmuller_expansion(2)
3 + 3*5 + 2*5^2 + 3*5^3 + O(5^4)
```

unit_part()

Returns the unit part of this element.

This is the p-adic element u in the same ring so that this element is $\pi^v u$, where π is a uniformizer and v is the valuation of this element.

EXAMPLES:

```
sage: # needs sage.libs.ntl
sage: R.<a> = Zq(125)
sage: (5*a).unit_part()
a + O(5^20)
```

sage.rings.padics.padic_capped_absolute_element.**unpickle_cae_v2**(*cls*, *parent*, *value*, *absprec*)

Unpickle capped absolute elements.

INPUT:

- cls – the class of the capped absolute element
- parent – a p-adic ring
- value – a Python object wrapping a celement, of the kind accepted by the cunpickle function
- absprec – a Python int or Sage integer

EXAMPLES:

```
sage: from sage.rings.padics.padic_capped_absolute_element import unpickle_cae_v2,␣
↪pAdicCappedAbsoluteElement
sage: R = ZpCA(5,8)
sage: a = unpickle_cae_v2(pAdicCappedAbsoluteElement, R, 42, int(6)); a
2 + 3*5 + 5^2 + O(5^6)
sage: a.parent() is R
True
```

CHAPTER

SIXTEEN

P-ADIC FIXED-MOD ELEMENT

Elements of p-adic Rings with Fixed Modulus

AUTHORS:

- David Roe
- Genya Zaytman: documentation
- David Harvey: doctests

class sage.rings.padics.padic_fixed_mod_element.**ExpansionIter**

Bases: object

An iterator over a p-adic expansion.

This class should not be instantiated directly, but instead using expansion().

INPUT:

- elt – the p-adic element
- prec – the number of terms to be emitted
- mode – either simple_mode, smallest_mode or teichmuller_mode

EXAMPLES:

```
sage: E = Zp(5,4)(373).expansion()
sage: I = iter(E)  # indirect doctest
sage: type(I)
<class 'sage.rings.padics.padic_capped_relative_element.ExpansionIter'>
```

class sage.rings.padics.padic_fixed_mod_element.**ExpansionIterable**

Bases: object

An iterable storing a p-adic expansion of an element.

This class should not be instantiated directly, but instead using expansion().

INPUT:

- elt – the p-adic element
- prec – the number of terms to be emitted
- val_shift – how many zeros to add at the beginning of the expansion, or the number of initial terms to truncate (if negative)
- mode – one of the following:
  - 'simple_mode'

– 'smallest_mode'

– 'teichmuller_mode'

EXAMPLES:

```
sage: E = Zp(5,4)(373).expansion()  # indirect doctest
sage: type(E)
<class 'sage.rings.padics.padic_capped_relative_element.ExpansionIterable'>
```

class sage.rings.padics.padic_fixed_mod_element.**FMElement**

Bases: *pAdicTemplateElement*

add_bigoh(*absprec*)

Return a new element truncated modulo π^{absprec}.

INPUT:

- absprec – an integer or infinity

OUTPUT:

a new element truncated modulo π^{absprec}.

EXAMPLES:

```
sage: R = Zp(7,4,'fixed-mod','series'); a = R(8); a.add_bigoh(1)
1
```

is_equal_to(*_right*, *absprec=None*)

Return whether this element is equal to right modulo p^{absprec}.

If absprec is None, returns if self == 0.

INPUT:

- right – a p-adic element with the same parent
- absprec – a positive integer or None (default: None)

EXAMPLES:

```
sage: R = ZpFM(2, 6)
sage: R(13).is_equal_to(R(13))
True
sage: R(13).is_equal_to(R(13+2^10))
True
sage: R(13).is_equal_to(R(17), 2)
True
sage: R(13).is_equal_to(R(17), 5)
False
```

is_zero(*absprec=None*)

Returns whether self is zero modulo π^{absprec}.

INPUT:

- absprec – an integer

EXAMPLES:

```
sage: R = ZpFM(17, 6)
sage: R(0).is_zero()
True
sage: R(17^6).is_zero()
True
sage: R(17^2).is_zero(absprec=2)
True
```

polynomial(*var='x'*)

Return a polynomial over the base ring that yields this element when evaluated at the generator of the parent.

INPUT:

- `var` – string, the variable name for the polynomial

EXAMPLES:

```
sage: # needs sage.libs.flint
sage: R.<a> = ZqFM(5^3)
sage: a.polynomial()
x
sage: a.polynomial(var='y')
y
sage: (5*a^2 + 25).polynomial()
5*x^2 + 5^2
```

precision_absolute()

The absolute precision of this element.

EXAMPLES:

```
sage: R = Zp(7,4,'fixed-mod'); a = R(7); a.precision_absolute()
4
```

precision_relative()

The relative precision of this element.

EXAMPLES:

```
sage: R = Zp(7,4,'fixed-mod'); a = R(7); a.precision_relative()
3
sage: a = R(0); a.precision_relative()
0
```

unit_part()

Return the unit part of `self`.

If the valuation of `self` is positive, then the high digits of the result will be zero.

EXAMPLES:

```
sage: R = Zp(17, 4, 'fixed-mod')
sage: R(5).unit_part()
5
sage: R(18*17).unit_part()
1 + 17
```

(continues on next page)

(continued from previous page)

```
sage: R(0).unit_part()
0
sage: type(R(5).unit_part())
<class 'sage.rings.padics.padic_fixed_mod_element.pAdicFixedModElement'>
sage: R = ZpFM(5, 5); a = R(75); a.unit_part()
3
```

val_unit()

Return a 2-tuple, the first element set to the valuation of `self`, and the second to the unit part of `self`.

If `self == 0`, then the unit part is `O(p^self.parent().precision_cap())`.

EXAMPLES:

```
sage: R = ZpFM(5,5)
sage: a = R(75); b = a - a
sage: a.val_unit()
(2, 3)
sage: b.val_unit()
(5, 0)
```

class sage.rings.padics.padic_fixed_mod_element.**PowComputer_**

Bases: *PowComputer_base*

A PowComputer for a fixed-modulus padic ring.

sage.rings.padics.padic_fixed_mod_element.**make_pAdicFixedModElement**(*parent*, *value*)

Unpickles a fixed modulus element.

EXAMPLES:

```
sage: from sage.rings.padics.padic_fixed_mod_element import make_
↪pAdicFixedModElement
sage: R = ZpFM(5)
sage: a = make_pAdicFixedModElement(R, 17*25); a
2*5^2 + 3*5^3
```

class sage.rings.padics.padic_fixed_mod_element.**pAdicCoercion_FM_frac_field**

Bases: `RingHomomorphism`

The canonical inclusion of $\mathbf{Z}_q$ into its fraction field.

EXAMPLES:

```
sage: # needs sage.libs.flint
sage: R.<a> = ZqFM(27, implementation='FLINT')
sage: K = R.fraction_field()
sage: f = K.coerce_map_from(R); f
Ring morphism:
  From: 3-adic Unramified Extension Ring in a defined by x^3 + 2*x + 1
  To:   3-adic Unramified Extension Field in a defined by x^3 + 2*x + 1
```

is_injective()

Return whether this map is injective.

EXAMPLES:

```
sage: # needs sage.libs.flint
sage: R.<a> = ZqFM(9)
sage: K = R.fraction_field()
sage: f = K.coerce_map_from(R)
sage: f.is_injective()
True
```

is_surjective()

Return whether this map is surjective.

EXAMPLES:

```
sage: # needs sage.libs.flint
sage: R.<a> = ZqFM(9)
sage: K = R.fraction_field()
sage: f = K.coerce_map_from(R)
sage: f.is_surjective()
False
```

section()

Return a map back to the ring that converts elements of non-negative valuation.

EXAMPLES:

```
sage: # needs sage.libs.flint
sage: R.<a> = ZqFM(27)
sage: K = R.fraction_field()
sage: f = K.coerce_map_from(R)
sage: f.section()(K.gen())
a
```

class sage.rings.padics.padic_fixed_mod_element.**pAdicCoercion_ZZ_FM**

Bases: RingHomomorphism

The canonical inclusion from $\mathbf{Z}$ to a fixed modulus ring.

EXAMPLES:

```
sage: f = ZpFM(5).coerce_map_from(ZZ); f
Ring morphism:
  From: Integer Ring
  To:   5-adic Ring of fixed modulus 5^20
```

section()

Returns a map back to $\mathbf{Z}$ that approximates an element of this p-adic ring by an integer.

EXAMPLES:

```
sage: f = ZpFM(5).coerce_map_from(ZZ).section()
sage: f(ZpFM(5)(-1)) - 5^20
-1
```

class sage.rings.padics.padic_fixed_mod_element.**pAdicConvert_FM_ZZ**

Bases: RingMap

The map from a fixed modulus ring back to $\mathbf{Z}$ that returns the smallest non-negative integer approximation to its input which is accurate up to the precision.

If the input is not in the closure of the image of $\mathbf{Z}$, raises a ValueError.

EXAMPLES:

```
sage: f = ZpFM(5).coerce_map_from(ZZ).section(); f
Set-theoretic ring morphism:
  From: 5-adic Ring of fixed modulus 5^20
  To:   Integer Ring
```

class sage.rings.padics.padic_fixed_mod_element.**pAdicConvert_FM_frac_field**

Bases: Morphism

The section of the inclusion from $\mathbf{Z}_q$ to its fraction field.

EXAMPLES:

```
sage: # needs sage.libs.flint
sage: R.<a> = ZqFM(27)
sage: K = R.fraction_field()
sage: f = R.convert_map_from(K); f
Generic morphism:
  From: 3-adic Unramified Extension Field in a defined by x^3 + 2*x + 1
  To:   3-adic Unramified Extension Ring in a defined by x^3 + 2*x + 1
```

class sage.rings.padics.padic_fixed_mod_element.**pAdicConvert_QQ_FM**

Bases: Morphism

The inclusion map from $\mathbf{Q}$ to a fixed modulus ring that is defined on all elements with non-negative p-adic valuation.

EXAMPLES:

```
sage: f = ZpFM(5).convert_map_from(QQ); f
Generic morphism:
  From: Rational Field
  To:   5-adic Ring of fixed modulus 5^20
```

class sage.rings.padics.padic_fixed_mod_element.**pAdicFixedModElement**

Bases: *FMElement*

INPUT:

- `parent` – a `pAdicRingFixedMod` object.
- `x` – input data to be converted into the parent.
- `absprec` – ignored; for compatibility with other p-adic rings
- `relprec` – ignored; for compatibility with other p-adic rings

Note: The following types are currently supported for x:

- Integers
- Rationals – denominator must be relatively prime to p
- FixedMod p-adics

- Elements of `IntegerModRing(p^k)` for `k` less than or equal to the modulus

The following types should be supported eventually:

- Finite precision p-adics
- Lazy p-adics
- Elements of local extensions of THIS p-adic ring that actually lie in $\mathbf{Z}_p$

EXAMPLES:

```
sage: R = Zp(5, 20, 'fixed-mod', 'terse')
```

Construct from integers:

```
sage: R(3)
3
sage: R(75)
75
sage: R(0)
0

sage: R(-1)
95367431640624
sage: R(-5)
95367431640620
```

Construct from rationals:

```
sage: R(1/2)
47683715820313
sage: R(-7875/874)
9493096742250
sage: R(15/425)
Traceback (most recent call last):
...
ValueError: p divides denominator
```

Construct from IntegerMod:

```
sage: R(Integers(125)(3))
3
sage: R(Integers(5)(3))
3
sage: R(Integers(5^30)(3))
3
sage: R(Integers(5^30)(1+5^23))
1
sage: R(Integers(49)(3))
Traceback (most recent call last):
...
TypeError: p does not divide modulus 49

sage: R(Integers(48)(3))
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
TypeError: p does not divide modulus 48
```

Some other conversions:

```
sage: R(R(5))
5
```

Todo: doctests for converting from other types of p-adic rings

lift()

Return an integer congruent to `self` modulo the precision.

Warning: Since fixed modulus elements don't track their precision, the result may not be correct modulo $i^{\mathrm{prec}_\mathrm{c}\mathrm{ap}}$ if the element was defined by constructions that lost precision.

EXAMPLES:

```
sage: R = Zp(7,4,'fixed-mod'); a = R(8); a.lift()
8
sage: type(a.lift())
<class 'sage.rings.integer.Integer'>
```

multiplicative_order()

Return the minimum possible multiplicative order of `self`.

OUTPUT:

an integer – the multiplicative order of this element. This is the minimum multiplicative order of all elements of $\mathbf{Z}_p$ lifting this element to infinite precision.

EXAMPLES:

```
sage: R = ZpFM(7, 6)
sage: R(1/3)
5 + 4*7 + 4*7^2 + 4*7^3 + 4*7^4 + 4*7^5
sage: R(1/3).multiplicative_order()
+Infinity
sage: R(7).multiplicative_order()
+Infinity
sage: R(1).multiplicative_order()
1
sage: R(-1).multiplicative_order()
2
sage: R.teichmuller(3).multiplicative_order()
6
```

residue(*absprec=1*, *field=None*, *check_prec=False*)

Reduce `self` modulo p^{absprec}.

INPUT:

- absprec – an integer (default: 1)
- field – boolean (default None). Whether to return an element of GF(p) or Zmod(p).
- check_prec – boolean (default False). No effect (for compatibility with other types).

OUTPUT:

This element reduced modulo p^{absprec} as an element of $\mathbf{Z}/p^{\mathrm{absprec}}\mathbf{Z}$.

EXAMPLES:

```
sage: R = Zp(7,4,'fixed-mod')
sage: a = R(8)
sage: a.residue(1)
1
```

This is different from applying % p^n which returns an element in the same ring:

```
sage: b = a.residue(2); b
8
sage: b.parent()
Ring of integers modulo 49
sage: c = a % 7^2; c
1 + 7
sage: c.parent()
7-adic Ring of fixed modulus 7^4
```

See also:

mod()

class sage.rings.padics.padic_fixed_mod_element.**pAdicTemplateElement**

Bases: *pAdicGenericElement*

A class for common functionality among the p-adic template classes.

INPUT:

- parent – a local ring or field
- x – data defining this element. Various types are supported, including ints, Integers, Rationals, PARI p-adics, integers mod p^k and other Sage p-adics.
- absprec – a cap on the absolute precision of this element
- relprec – a cap on the relative precision of this element

EXAMPLES:

```
sage: Zp(17)(17^3, 8, 4)
17^3 + O(17^7)
```

expansion(*n=None*, *lift_mode='simple'*, *start_val=None*)

Return the coefficients in a π-adic expansion. If this is a field element, start at $\pi^{\mathrm{valuation}}$, if a ring element at π^0.

For each lift mode, this function returns a list of a_i so that this element can be expressed as

$$\pi^v \cdot \sum_{i=0}^{\infty} a_i \pi^i,$$

where v is the valuation of this element when the parent is a field, and $v = 0$ otherwise.

Different lift modes affect the choice of a_i. When `lift_mode` is `'simple'`, the resulting a_i will be non-negative: if the residue field is $\mathbf{F}_p$ then they will be integers with $0 \leq a_i < p$; otherwise they will be a list of integers in the same range giving the coefficients of a polynomial in the indeterminant representing the maximal unramified subextension.

Choosing `lift_mode` as `'smallest'` is similar to `'simple'`, but uses a balanced representation $-p/2 < a_i \leq p/2$.

Finally, setting `lift_mode = 'teichmuller'` will yield Teichmuller representatives for the a_i: $a_i^q = a_i$. In this case the a_i will lie in the ring of integers of the maximal unramified subextension of the parent of this element.

INPUT:

- `n` – integer (default `None`). If given, returns the corresponding entry in the expansion. Can also accept a slice (see `slice()`)
- `lift_mode` – `'simple'`, `'smallest'` or `'teichmuller'` (default: `'simple'`)
- `start_val` – start at this valuation rather than the default (0 or the valuation of this element).

OUTPUT:

- If `n` is `None`, an iterable giving a π-adic expansion of this element. For base elements the contents will be integers if `lift_mode` is `'simple'` or `'smallest'`, and elements of `self.parent()` if `lift_mode` is `'teichmuller'`.
- If `n` is an integer, the coefficient of π^n in the π-adic expansion of this element.

Note: Use slice operators to get a particular range.

EXAMPLES:

```
sage: R = Zp(7,6); a = R(12837162817); a
3 + 4*7 + 4*7^2 + 4*7^4 + O(7^6)
sage: E = a.expansion(); E
7-adic expansion of 3 + 4*7 + 4*7^2 + 4*7^4 + O(7^6)
sage: list(E)
[3, 4, 4, 0, 4, 0]
sage: sum([c * 7^i for i, c in enumerate(E)]) == a
True
sage: E = a.expansion(lift_mode='smallest'); E
7-adic expansion of 3 + 4*7 + 4*7^2 + 4*7^4 + O(7^6) (balanced)
sage: list(E)
[3, -3, -2, 1, -3, 1]
sage: sum([c * 7^i for i, c in enumerate(E)]) == a
True
sage: E = a.expansion(lift_mode='teichmuller'); E
7-adic expansion of 3 + 4*7 + 4*7^2 + 4*7^4 + O(7^6) (teichmuller)
sage: list(E)
[3 + 4*7 + 6*7^2 + 3*7^3 + 2*7^5 + O(7^6),
0,
5 + 2*7 + 3*7^3 + O(7^4),
1 + O(7^3),
3 + 4*7 + O(7^2),
```

(continues on next page)

(continued from previous page)

```
5 + O(7)]
sage: sum(c * 7^i for i, c in enumerate(E))
3 + 4*7 + 4*7^2 + 4*7^4 + O(7^6)
```

If the element has positive valuation then the list will start with some zeros:

```
sage: a = R(7^3 * 17)
sage: E = a.expansion(); E
7-adic expansion of 3*7^3 + 2*7^4 + O(7^9)
sage: list(E)
[0, 0, 0, 3, 2, 0, 0, 0, 0]
```

The expansion of 0 is truncated:

```
sage: E = R(0, 7).expansion(); E
7-adic expansion of O(7^7)
sage: len(E)
0
sage: list(E)
[]
```

In fields, on the other hand, the expansion starts at the valuation:

```
sage: R = Qp(7,4); a = R(6*7+7**2); E = a.expansion(); E
7-adic expansion of 6*7 + 7^2 + O(7^5)
sage: list(E)
[6, 1, 0, 0]
sage: list(a.expansion(lift_mode='smallest'))
[-1, 2, 0, 0]
sage: list(a.expansion(lift_mode='teichmuller'))
[6 + 6*7 + 6*7^2 + 6*7^3 + O(7^4),
2 + 4*7 + 6*7^2 + O(7^3),
3 + 4*7 + O(7^2),
3 + O(7)]
```

You can ask for a specific entry in the expansion:

```
sage: a.expansion(1)
6
sage: a.expansion(1, lift_mode='smallest')
-1
sage: a.expansion(2, lift_mode='teichmuller')
2 + 4*7 + 6*7^2 + O(7^3)
```

lift_to_precision(*absprec=None*)

Return another element of the same parent with absolute precision at least `absprec`, congruent to this p-adic element modulo the precision of this element.

INPUT:

- `absprec` – an integer or `None` (default: `None`); the absolute precision of the result. If `None`, lifts to the maximum precision allowed

Note: If setting `absprec` that high would violate the precision cap, raises a precision error. Note that the new digits will not necessarily be zero.

EXAMPLES:

```
sage: R = ZpCA(17)
sage: R(-1,2).lift_to_precision(10)
16 + 16*17 + O(17^10)
sage: R(1,15).lift_to_precision(10)
1 + O(17^15)
sage: R(1,15).lift_to_precision(30)
Traceback (most recent call last):
...
PrecisionError: precision higher than allowed by the precision cap
sage: R(-1,2).lift_to_precision().precision_absolute() == R.precision_cap()
True

sage: R = Zp(5); c = R(17,3); c.lift_to_precision(8)
2 + 3*5 + O(5^8)
sage: c.lift_to_precision().precision_relative() == R.precision_cap()
True
```

Fixed modulus elements don't raise errors:

```
sage: R = ZpFM(5); a = R(5); a.lift_to_precision(7)
5
sage: a.lift_to_precision(10000)
5
```

residue(*absprec=1*, *field=None*, *check_prec=True*)

Reduce this element modulo p^{absprec}.

INPUT:

- `absprec` – `0` or 1.
- `field` – boolean (default `None`). For precision 1, whether to return an element of the residue field or a residue ring. Currently unused.
- `check_prec` – boolean (default `True`). Whether to raise an error if this element has insufficient precision to determine the reduction. Errors are never raised for fixed-mod or floating-point types.

OUTPUT:

This element reduced modulo p^{absprec} as an element of the residue field or the null ring.

EXAMPLES:

```
sage: # needs sage.libs.ntl
sage: R.<a> = Zq(27, 4)
sage: (3 + 3*a).residue()
0
sage: (a + 1).residue()
a0 + 1
```

teichmuller_expansion(*n=None*)

Returns an iterator over coefficients $a_0, a_1, \dots, a_n$ such that

- $a_i^q = a_i$, where q is the cardinality of the residue field,

- this element can be expressed as

$$\pi^v \cdot \sum_{i=0}^{\infty} a_i \pi^i$$

where v is the valuation of this element when the parent is a field, and $v = 0$ otherwise.

- if $a_i \neq 0$, the precision of a_i is i less than the precision of this element (relative in the case that the parent is a field, absolute otherwise)

Note: The coefficients will lie in the ring of integers of the maximal unramified subextension.

INPUT:

- n – integer (default None). If given, returns the coefficient of π^n in the expansion.

EXAMPLES:

For fields, the expansion starts at the valuation:

```
sage: R = Qp(5,5); list(R(70).teichmuller_expansion())
[4 + 4*5 + 4*5^2 + 4*5^3 + 4*5^4 + O(5^5),
3 + 3*5 + 2*5^2 + 3*5^3 + O(5^4),
2 + 5 + 2*5^2 + O(5^3),
1 + O(5^2),
4 + O(5)]
```

But if you specify n, you get the coefficient of π^n:

```
sage: R(70).teichmuller_expansion(2)
3 + 3*5 + 2*5^2 + 3*5^3 + O(5^4)
```

unit_part()

Returns the unit part of this element.

This is the p-adic element u in the same ring so that this element is $\pi^v u$, where π is a uniformizer and v is the valuation of this element.

EXAMPLES:

```
sage: # needs sage.libs.ntl
sage: R.<a> = Zq(125)
sage: (5*a).unit_part()
a + O(5^20)
```

sage.rings.padics.padic_fixed_mod_element.**unpickle_fme_v2**(*cls*, *parent*, *value*)

Unpickles a fixed-mod element.

EXAMPLES:

```
sage: from sage.rings.padics.padic_fixed_mod_element import pAdicFixedModElement,␣
↪unpickle_fme_v2
sage: R = ZpFM(5)
sage: a = unpickle_fme_v2(pAdicFixedModElement, R, 17*25); a
2*5^2 + 3*5^3
sage: a.parent() is R
True
```

CHAPTER

SEVENTEEN

P-ADIC EXTENSION ELEMENT

A common superclass for all elements of extension rings and field of $\mathbf{Z}_p$ and $\mathbf{Q}_p$.

AUTHORS:

- David Roe (2007): initial version
- Julian Rueth (2012-10-18): added residue

class sage.rings.padics.padic_ext_element.**pAdicExtElement**

Bases: *pAdicGenericElement*

frobenius(*arithmetic=True*)

Return the image of this element under the Frobenius automorphism applied to its parent.

INPUT:

- `arithmetic` – whether to apply the arithmetic Frobenius (acting by raising to the p-th power on the residue field). If `False` is provided, the image of geometric Frobenius (raising to the $(1/p)$-th power on the residue field) will be returned instead.

EXAMPLES:

```
sage: R.<a> = Zq(5^4,3)
sage: a.frobenius()
(a^3 + a^2 + 3*a) + (3*a + 1)*5 + (2*a^3 + 2*a^2 + 2*a)*5^2 + O(5^3)
sage: f = R.defining_polynomial()
sage: f(a)
O(5^3)
sage: f(a.frobenius())
O(5^3)
sage: for i in range(4): a = a.frobenius()
sage: a
a + O(5^3)

sage: K.<a> = Qq(7^3,4)
sage: b = (a+1)/7
sage: c = b.frobenius(); c
(3*a^2 + 5*a + 1)*7^-1 + (6*a^2 + 6*a + 6) + (4*a^2 + 3*a + 4)*7 + (6*a^2 + a +
↪6)*7^2 + O(7^3)
sage: c.frobenius().frobenius()
(a + 1)*7^-1 + O(7^3)
```

An error will be raised if the parent of self is a ramified extension:

```
sage: x = polygen(ZZ, 'x')
sage: K.<a> = Qp(5).extension(x^2 - 5)
sage: a.frobenius()
Traceback (most recent call last):
...
NotImplementedError: Frobenius automorphism only implemented for unramified␣
↪extensions
```

residue(*absprec=1*, *field=None*, *check_prec=True*)

Reduces this element modulo π^{absprec}.

INPUT:

- `absprec` – a non-negative integer (default: 1)
- `field` – boolean (default `None`). For precision 1, whether to return an element of the residue field or a residue ring. Currently unused.
- `check_prec` – boolean (default `True`). Whether to raise an error if this element has insufficient precision to determine the reduction. Errors are never raised for fixed-mod or floating-point types.

OUTPUT:

This element reduced modulo π^{absprec}.

If `absprec` is zero, then as an element of $\mathbf{Z}/(1)$.

If `absprec` is one, then as an element of the residue field.

Note: Only implemented for `absprec` less than or equal to one.

AUTHORS:

- Julian Rueth (2012-10-18): initial version

EXAMPLES:

Unramified case:

```
sage: # needs sage.libs.flint
sage: R = ZpCA(3,5)
sage: S.<a> = R[]
sage: W.<a> = R.extension(a^2 + 9*a + 1)
sage: (a + 1).residue(1)
a0 + 1
sage: a.residue(2)
Traceback (most recent call last):
...
NotImplementedError: reduction modulo p^n with n>1
```

Eisenstein case:

```
sage: R = ZpCA(3,5)
sage: S.<a> = R[]
sage: W.<a> = R.extension(a^2 + 9*a + 3)
sage: (a + 1).residue(1)
1
```

(continues on next page)

(continued from previous page)

```
sage: a.residue(2)
Traceback (most recent call last):
...
NotImplementedError: residue() not implemented in extensions for absprec larger␣
↪than one
```

CHAPTER

EIGHTEEN

P-ADIC ZZ_PX ELEMENT

A common superclass implementing features shared by all elements that use NTL's ZZ_pX as the fundamental data type.

AUTHORS:

- David Roe

class sage.rings.padics.padic_ZZ_pX_element.**pAdicZZpXElement**

Bases: *pAdicExtElement*

Initialization

EXAMPLES:

```
sage: A = Zp(next_prime(50000),10)
sage: S.<x> = A[]
sage: B.<t> = A.ext(x^2 + next_prime(50000))  # indirect doctest
```

norm(*base=None*)

Return the absolute or relative norm of this element.

Note: This is not the p-adic absolute value. This is a field theoretic norm down to a ground ring. If you want the p-adic absolute value, use the abs() function instead.

If base is given then base must be a subfield of the parent L of self, in which case the norm is the relative norm from L to base.

In all other cases, the norm is the absolute norm down to $\mathbf{Q}_p$ or $\mathbf{Z}_p$.

EXAMPLES:

```
sage: R = ZpCR(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: ((1+2*w)^5).norm()
1 + 5^2 + O(5^5)
sage: ((1+2*w)).norm()^5
1 + 5^2 + O(5^5)
```

trace(*base=None*)

Return the absolute or relative trace of this element.

If `base` is given then `base` must be a subfield of the parent L of `self`, in which case the norm is the relative norm from L to `base`.

In all other cases, the norm is the absolute norm down to $\mathbf{Q}_p$ or $\mathbf{Z}_p$.

EXAMPLES:

```
sage: R = ZpCR(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: a = (2+3*w)^7
sage: b = (6+w^3)^5
sage: a.trace()
3*5 + 2*5^2 + 3*5^3 + 2*5^4 + O(5^5)
sage: a.trace() + b.trace()
4*5 + 5^2 + 5^3 + 2*5^4 + O(5^5)
sage: (a+b).trace()
4*5 + 5^2 + 5^3 + 2*5^4 + O(5^5)
```

CHAPTER

NINETEEN

P-ADIC ZZ_PX CR ELEMENT

This file implements elements of Eisenstein and unramified extensions of $\mathbf{Z}_p$ and $\mathbf{Q}_p$ with capped relative precision.

For the parent class see *sage.rings.padics.padic_extension_leaves*.

The underlying implementation is through NTL's ZZ_pX class. Each element contains the following data:

- ordp (long) – A power of the uniformizer to scale the unit by. For unramified extensions this uniformizer is p, for Eisenstein extensions it is not. A value equal to the maximum value of a long indicates that the element is an exact zero.
- relprec (long) – A signed integer giving the precision to which this element is defined. For nonzero relprec, the absolute value gives the power of the uniformizer modulo which the unit is defined. A positive value indicates that the element is normalized (ie unit is actually a unit: in the case of Eisenstein extensions the constant term is not divisible by p, in the case of unramified extensions that there is at least one coefficient that is not divisible by p). A negative value indicates that the element may or may not be normalized. A zero value indicates that the element is zero to some precision. If so, ordp gives the absolute precision of the element. If ordp is greater than maxordp, then the element is an exact zero.
- unit (ZZ_pX_c) – An ntl ZZ_pX storing the unit part. The variable x is the uniformizer in the case of Eisenstein extensions. If the element is not normalized, the unit may or may not actually be a unit. This ZZ_pX is created with global ntl modulus determined by the absolute value of relprec. If relprec is 0, **unit is not initialized**, or destructed if normalized and found to be zero. Otherwise, let r be relprec and e be the ramification index over $\mathbf{Q}_p$ or $\mathbf{Z}_p$. Then the modulus of unit is given by $p^{ceil(r/e)}$. Note that all kinds of problems arise if you try to mix moduli. ZZ_pX_conv_modulus gives a semi-safe way to convert between different moduli without having to pass through ZZX.
- prime_pow (some subclass of PowComputer_ZZ_pX) – a class, identical among all elements with the same parent, holding common data.
  - prime_pow.deg – The degree of the extension
  - prime_pow.e – The ramification index
  - prime_pow.f – The inertia degree
  - prime_pow.prec_cap – the unramified precision cap. For Eisenstein extensions this is the smallest power of p that is zero.
  - prime_pow.ram_prec_cap – the ramified precision cap. For Eisenstein extensions this will be the smallest power of x that is indistinguishable from zero.
  - prime_pow.pow_ZZ_tmp, prime_pow.pow_mpz_t_tmp``, prime_pow.pow_Integer – functions for accessing powers of p. The first two return pointers. See *sage.rings.padics.pow_computer_ext* for examples and important warnings.

- `prime_pow.get_context`, `prime_pow.get_context_capdiv`, `prime_pow.get_top_context` – obtain an `ntl_ZZ_pContext_class` corresponding to p^n. The capdiv version divides by `prime_pow.e` as appropriate. `top_context` corresponds to $p^{\text{prec\_cap}}$.
- `prime_pow.restore_context`, `prime_pow.restore_context_capdiv`, `prime_pow.restore_top_context` – restores the given context.
- `prime_pow.get_modulus`, `get_modulus_capdiv`, `get_top_modulus` – Returns a `ZZ_pX_Modulus_c*` pointing to a polynomial modulus defined modulo p^n (appropriately divided by `prime_pow.e` in the capdiv case).

EXAMPLES:

An Eisenstein extension:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f); W
5-adic Eisenstein Extension Ring in w defined by x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: z = (1+w)^5; z
1 + w^5 + w^6 + 2*w^7 + 4*w^8 + 3*w^10 + w^12 + 4*w^13 + 4*w^14 + 4*w^15 + 4*w^16 + 4*w^
↪17 + 4*w^20 + w^21 + 4*w^24 + O(w^25)
sage: y = z >> 1; y
w^4 + w^5 + 2*w^6 + 4*w^7 + 3*w^9 + w^11 + 4*w^12 + 4*w^13 + 4*w^14 + 4*w^15 + 4*w^16 +␣
↪4*w^19 + w^20 + 4*w^23 + O(w^24)
sage: y.valuation()
4
sage: y.precision_relative()
20
sage: y.precision_absolute()
24
sage: z - (y << 1)
1 + O(w^25)
sage: (1/w)^12+w
w^-12 + w + O(w^13)
sage: (1/w).parent()
5-adic Eisenstein Extension Field in w defined by x^5 + 75*x^3 - 15*x^2 + 125*x - 5
```

Unramified extensions:

```
sage: g = x^3 + 3*x + 3
sage: A.<a> = R.ext(g)
sage: z = (1+a)^5; z
(2*a^2 + 4*a) + (3*a^2 + 3*a + 1)*5 + (4*a^2 + 3*a + 4)*5^2 + (4*a^2 + 4*a + 4)*5^3 +␣
↪(4*a^2 + 4*a + 4)*5^4 + O(5^5)
sage: z - 1 - 5*a - 10*a^2 - 10*a^3 - 5*a^4 - a^5
O(5^5)
sage: y = z >> 1; y
(3*a^2 + 3*a + 1) + (4*a^2 + 3*a + 4)*5 + (4*a^2 + 4*a + 4)*5^2 + (4*a^2 + 4*a + 4)*5^3␣
↪+ O(5^4)
sage: 1/a
(3*a^2 + 4) + (a^2 + 4)*5 + (3*a^2 + 4)*5^2 + (a^2 + 4)*5^3 + (3*a^2 + 4)*5^4 + O(5^5)
sage: FFp = R.residue_field()
sage: R(FFp(3))
3 + O(5)
```

(continues on next page)

(continued from previous page)

```
sage: QQq.<zz> = Qq(25,4)
sage: QQq(FFp(3))
3 + O(5)
sage: FFq = QQq.residue_field(); QQq(FFq(3))
3 + O(5)
sage: zz0 = FFq.gen(); QQq(zz0^2)
(zz + 3) + O(5)
```

Different printing modes:

```
sage: R = Zp(5, print_mode='digits'); S.<x> = R[]; f = x^5 + 75*x^3 - 15*x^2 + 125*x -5;␣
↪W.<w> = R.ext(f)
sage: z = (1+w)^5; repr(z)
'...
↪4110403113210310442221311242000111011201102002023303214332011214403232013144001400444441030421100001
↪'
sage: R = Zp(5, print_mode='bars'); S.<x> = R[]; g = x^3 + 3*x + 3; A.<a> = R.ext(g)
sage: z = (1+a)^5; repr(z)
'...[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4,␣
↪4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4,␣
↪4]|[4, 3, 4]|[1, 3, 3]|[0, 4, 2]'
sage: R = Zp(5, print_mode='terse'); S.<x> = R[]; f = x^5 + 75*x^3 - 15*x^2 + 125*x -5;␣
↪W.<w> = R.ext(f)
sage: z = (1+w)^5; z
6 + 95367431640505*w + 25*w^2 + 95367431640560*w^3 + 5*w^4 + O(w^100)
sage: R = Zp(5, print_mode='val-unit'); S.<x> = R[]; f = x^5 + 75*x^3 - 15*x^2 + 125*x -
↪5; W.<w> = R.ext(f)
sage: y = (1+w)^5 - 1; y
w^5 * (2090041 + 19073486126901*w + 1258902*w^2 + 674*w^3 + 16785*w^4) + O(w^100)
```

You can get at the underlying ntl unit:

```
sage: z._ntl_rep()
[6 95367431640505 25 95367431640560 5]
sage: y._ntl_rep()
[2090041 19073486126901 1258902 674 16785]
sage: y._ntl_rep_abs()
([5 95367431640505 25 95367431640560 5], 0)
```

Note: If you get an error `internal error: can't grow this _ntl_gbigint,` it indicates that moduli are being mixed inappropriately somewhere.

For example, when calling a function with a `ZZ_pX_c` as an argument, it copies. If the modulus is not set to the modulus of the `ZZ_pX_c`, you can get errors.

AUTHORS:

- David Roe (2008-01-01): initial version
- Robert Harron (2011-09): fixes/enhancements
- Julian Rueth (2014-05-09): enable caching through `_cache_key`

sage.rings.padics.padic_ZZ_pX_CR_element.**make_ZZpXCRElement**(*parent*, *unit*, *ordp*, *relprec*, *version*)

Unpickling.

EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: y = W(775, 19); y
w^10 + 4*w^12 + 2*w^14 + w^15 + 2*w^16 + 4*w^17 + w^18 + O(w^19)
sage: loads(dumps(y))  # indirect doctest
w^10 + 4*w^12 + 2*w^14 + w^15 + 2*w^16 + 4*w^17 + w^18 + O(w^19)

sage: from sage.rings.padics.padic_ZZ_pX_CR_element import make_ZZpXCRElement
sage: make_ZZpXCRElement(W, y._ntl_rep(), 3, 9, 0)
w^3 + 4*w^5 + 2*w^7 + w^8 + 2*w^9 + 4*w^10 + w^11 + O(w^12)
```

class sage.rings.padics.padic_ZZ_pX_CR_element.**pAdicZZpXCRElement**

Bases: *pAdicZZpXElement*

Creates an element of a capped relative precision, unramified or Eisenstein extension of $\mathbf{Z}_p$ or $\mathbf{Q}_p$.

INPUT:

- `parent` – either an `EisensteinRingCappedRelative` or `UnramifiedRingCappedRelative`
- `x` – an integer, rational, p-adic element, polynomial, list, integer_mod, pari int/frac/poly_t/pol_mod, an `ntl_ZZ_pX`, an `ntl_ZZ`, an `ntl_ZZ_p`, an `ntl_ZZX`, or something convertible into parent.residue_field()
- `absprec` – an upper bound on the absolute precision of the element created
- `relprec` – an upper bound on the relative precision of the element created
- `empty` – whether to return after initializing to zero (without setting the valuation).

EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: z = (1+w)^5; z  # indirect doctest
1 + w^5 + w^6 + 2*w^7 + 4*w^8 + 3*w^10 + w^12 + 4*w^13 + 4*w^14 + 4*w^15 + 4*w^16 +
→4*w^17 + 4*w^20 + w^21 + 4*w^24 + O(w^25)
sage: W(pari('3 + O(5^3)'))
3 + O(w^15)
sage: W(R(3,3))
3 + O(w^15)
sage: W.<w> = R.ext(x^625 + 915*x^17 - 95)
sage: W(3)
3 + O(w^3125)
sage: W(w, 14)
w + O(w^14)
```

expansion(*n=None*, *lift_mode='simple'*)

Return a list giving a series representation of `self`.

- If `lift_mode == 'simple'` or `'smallest'`, the returned list will consist of integers (in the Eisenstein case) or a list of lists of integers (in the unramified case). `self` can be reconstructed as a sum of elements of the list times powers of the uniformiser (in the Eisenstein case), or as a sum of powers of the p times polynomials in the generator (in the unramified case).
  - If `lift_mode == 'simple'`, all integers will be in the interval $[0, p-1]$.
  - If `lift_mode == 'smallest'` they will be in the interval $[(1-p)/2, p/2]$.
- If `lift_mode == 'teichmuller'`, returns a list of `pAdicZZpXCRElements`, all of which are Teichmuller representatives and such that `self` is the sum of that list times powers of the uniformizer.

Note that zeros are truncated from the returned list if `self.parent()` is a field, so you must use the `valuation` function to fully reconstruct `self`.

INPUT:

- `n` – integer (default `None`). If given, returns the corresponding entry in the expansion.

EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: y = W(775, 19); y
w^10 + 4*w^12 + 2*w^14 + w^15 + 2*w^16 + 4*w^17 + w^18 + O(w^19)
sage: (y>>9).expansion()
[0, 1, 0, 4, 0, 2, 1, 2, 4, 1]
sage: (y>>9).expansion(lift_mode='smallest')
[0, 1, 0, -1, 0, 2, 1, 2, 0, 1]
sage: w^10 - w^12 + 2*w^14 + w^15 + 2*w^16 + w^18 + O(w^19)
w^10 + 4*w^12 + 2*w^14 + w^15 + 2*w^16 + 4*w^17 + w^18 + O(w^19)
sage: g = x^3 + 3*x + 3
sage: A.<a> = R.ext(g)
sage: y = 75 + 45*a + 1200*a^2; y
4*a*5 + (3*a^2 + a + 3)*5^2 + 4*a^2*5^3 + a^2*5^4 + O(5^6)
sage: E = y.expansion(); E
5-adic expansion of 4*a*5 + (3*a^2 + a + 3)*5^2 + 4*a^2*5^3 + a^2*5^4 + O(5^6)
sage: list(E)
[[], [0, 4], [3, 1, 3], [0, 0, 4], [0, 0, 1], []]
sage: list(y.expansion(lift_mode='smallest'))
[[], [0, -1], [-2, 2, -2], [1], [0, 0, 2], []]
sage: 5*((-2*5 + 25) + (-1 + 2*5)*a + (-2*5 + 2*125)*a^2)
4*a*5 + (3*a^2 + a + 3)*5^2 + 4*a^2*5^3 + a^2*5^4 + O(5^6)
sage: list(W(0).expansion())
[]
sage: list(W(0,4).expansion())
[]
sage: list(A(0,4).expansion())
[]
```

is_equal_to(*right*, *absprec=None*)

Return whether this element is equal to `right` modulo `self.uniformizer()^absprec`.

If `absprec` is `None`, checks whether this element is equal to `right` modulo the lower of their two precisions.

EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(47); b = W(47 + 25)
sage: a.is_equal_to(b)
False
sage: a.is_equal_to(b, 7)
True
```

is_zero(*absprec=None*)

Return whether the valuation of this element is at least `absprec`. If `absprec` is `None`, checks if this element is indistinguishable from zero.

If this element is an inexact zero of valuation less than `absprec`, raises a *PrecisionError*.

EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: O(w^189).is_zero()
True
sage: W(0).is_zero()
True
sage: a = W(675)
sage: a.is_zero()
False
sage: a.is_zero(7)
True
sage: a.is_zero(21)
False
```

lift_to_precision(*absprec=None*)

Return a `pAdicZZpXCRElement` congruent to this element but with absolute precision at least `absprec`.

INPUT:

- `absprec` – (default `None`) the absolute precision of the result. If `None`, lifts to the maximum precision allowed.

Note: If setting `absprec` that high would violate the precision cap, raises a precision error. If self is an inexact zero and `absprec` is greater than the maximum allowed valuation, raises an error.

Note that the new digits will not necessarily be zero.

EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(345, 17); a
```

(continues on next page)

(continued from previous page)

```
4*w^5 + 3*w^7 + w^9 + 3*w^10 + 2*w^11 + 4*w^12 + w^13 + 2*w^14 + 2*w^15 + O(w^
↪17)
sage: b = a.lift_to_precision(19); b
4*w^5 + 3*w^7 + w^9 + 3*w^10 + 2*w^11 + 4*w^12 + w^13 + 2*w^14 + 2*w^15
 + w^17 + 2*w^18 + O(w^19)
sage: c = a.lift_to_precision(24); c
4*w^5 + 3*w^7 + w^9 + 3*w^10 + 2*w^11 + 4*w^12 + w^13 + 2*w^14 + 2*w^15
 + w^17 + 2*w^18 + 4*w^19 + 4*w^20 + 2*w^21 + 4*w^23 + O(w^24)
sage: a._ntl_rep()
[19 35 118 60 121]
sage: b._ntl_rep()
[19 35 118 60 121]
sage: c._ntl_rep()
[19 35 118 60 121]
sage: a.lift_to_precision().precision_relative() == W.precision_cap()
True
```

matrix_mod_pn()

Return the matrix of right multiplication by the element on the power basis $1, x, x^2, \ldots, x^{d-1}$ for this extension field. Thus the *rows* of this matrix give the images of each of the x^i. The entries of the matrices are `IntegerMod` elements, defined modulo $p^{N/e}$ where N is the absolute precision of this element (unless this element is zero to arbitrary precision; in that case the entries are integer zeros.)

Raises an error if this element has negative valuation.

EXAMPLES:

```
sage: R = ZpCR(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: a = (3+w)^7
sage: a.matrix_mod_pn()
[2757  333 1068  725 2510]
[  50 1507  483  318  725]
[ 500   50 3007 2358  318]
[1590 1375 1695 1032 2358]
[2415  590 2370 2970 1032]
```

polynomial(*var='x'*)

Return a polynomial over the base ring that yields this element when evaluated at the generator of the parent.

INPUT:

- `var` – string, the variable name for the polynomial

EXAMPLES:

```
sage: S.<x> = ZZ[]
sage: W.<w> = Zp(5).extension(x^2 - 5)
sage: (w + W(5, 7)).polynomial()
(1 + O(5^3))*x + 5 + O(5^4)
```

precision_absolute()

Return the absolute precision of this element, i.e., the power of the uniformizer modulo which this element is defined.

EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(75, 19); a
3*w^10 + 2*w^12 + w^14 + w^16 + w^17 + 3*w^18 + O(w^19)
sage: a.valuation()
10
sage: a.precision_absolute()
19
sage: a.precision_relative()
9
sage: a.unit_part()
3 + 2*w^2 + w^4 + w^6 + w^7 + 3*w^8 + O(w^9)
sage: (a.unit_part() - 3).precision_absolute()
9
```

precision_relative()

Return the relative precision of this element, i.e., the power of the uniformizer modulo which the unit part of `self` is defined.

EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(75, 19); a
3*w^10 + 2*w^12 + w^14 + w^16 + w^17 + 3*w^18 + O(w^19)
sage: a.valuation()
10
sage: a.precision_absolute()
19
sage: a.precision_relative()
9
sage: a.unit_part()
3 + 2*w^2 + w^4 + w^6 + w^7 + 3*w^8 + O(w^9)
```

teichmuller_expansion(*n=None*)

Return a list $[a_0, a_1, \ldots, a_n]$ such that

- $a_i^q = a_i$
- `self.unit_part()` = $\sum_{i=0}^{n} a_i \pi^i$, where π is a uniformizer of `self.parent()`
- if $a_i \neq 0$, the absolute precision of a_i is `self.precision_relative() - i`

INPUT:

- `n` – integer (default `None`). If given, returns the corresponding entry in the expansion.

EXAMPLES:

```
sage: R.<a> = ZqCR(5^4,4)
sage: E = a.teichmuller_expansion(); E
5-adic expansion of a + O(5^4) (teichmuller)
sage: list(E)
[a + (2*a^3 + 2*a^2 + 3*a + 4)*5 + (4*a^3 + 3*a^2 + 3*a + 2)*5^2
   + (4*a^2 + 2*a + 2)*5^3 + O(5^4),
 (3*a^3 + 3*a^2 + 2*a + 1) + (a^3 + 4*a^2 + 1)*5 + (a^2 + 4*a + 4)*5^2 + O(5^3),
 (4*a^3 + 2*a^2 + a + 1) + (2*a^3 + 2*a^2 + 2*a + 4)*5 + O(5^2),
 (a^3 + a^2 + a + 4) + O(5)]
sage: sum([c * 5^i for i, c in enumerate(E)])
a + O(5^4)
sage: all(c^625 == c for c in E)
True

sage: S.<x> = ZZ[]
sage: f = x^3 - 98*x + 7
sage: W.<w> = ZpCR(7,3).ext(f)
sage: b = (1+w)^5; L = b.teichmuller_expansion(); L
[1 + O(w^9), 5 + 5*w^3 + w^6 + 4*w^7 + O(w^8), 3 + 3*w^3 + O(w^7),
 3 + 3*w^3 + O(w^6), O(w^5), 4 + 5*w^3 + O(w^4), 3 + O(w^3),
 6 + O(w^2), 6 + O(w)]
sage: sum([w^i*L[i] for i in range(9)]) == b
True
sage: all(L[i]^(7^3) == L[i] for i in range(9))
True

sage: L = W(3).teichmuller_expansion(); L
[3 + 3*w^3 + w^7 + O(w^9), O(w^8), O(w^7), 4 + 5*w^3 + O(w^6),
 O(w^5), O(w^4), 3 + O(w^3), 6 + O(w^2)]
sage: sum([w^i*L[i] for i in range(len(L))])
3 + O(w^9)
```

unit_part()

Return the unit part of this element, ie `self / uniformizer^(self.valuation())`.

EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(75, 19); a
3*w^10 + 2*w^12 + w^14 + w^16 + w^17 + 3*w^18 + O(w^19)
sage: a.valuation()
10
sage: a.precision_absolute()
19
sage: a.precision_relative()
9
sage: a.unit_part()
3 + 2*w^2 + w^4 + w^6 + w^7 + 3*w^8 + O(w^9)
```

CHAPTER

TWENTY

P-ADIC ZZ_PX CA ELEMENT

This file implements elements of Eisenstein and unramified extensions of Zp with capped absolute precision.

For the parent class see padic_extension_leaves.pyx.

The underlying implementation is through NTL's ZZ_pX class. Each element contains the following data:

- absprec (long) – An integer giving the precision to which this element is defined. This is the power of the uniformizer modulo which the element is well defined.
- value (ZZ_pX_c) – An ntl ZZ_pX storing the value. The variable x is the uniformizer in the case of Eisenstein extensions. This ZZ_pX is created with global ntl modulus determined by absprec. Let a be absprec and e be the ramification index over $\mathbf{Q}_p$ or $\mathbf{Z}_p$. Then the modulus is given by $p^{ceil(a/e)}$. Note that all kinds of problems arise if you try to mix moduli. ZZ_pX_conv_modulus gives a semi-safe way to convert between different moduli without having to pass through ZZX.
- prime_pow (some subclass of PowComputer_ZZ_pX) – a class, identical among all elements with the same parent, holding common data.
  - prime_pow.deg – The degree of the extension
  - prime_pow.e – The ramification index
  - prime_pow.f – The inertia degree
  - prime_pow.prec_cap – the unramified precision cap. For Eisenstein extensions this is the smallest power of p that is zero.
  - prime_pow.ram_prec_cap – the ramified precision cap. For Eisenstein extensions this will be the smallest power of x that is indistinguishable from zero.
  - prime_pow.pow_ZZ_tmp, prime_pow.pow_mpz_t_tmp``, prime_pow.pow_Integer – functions for accessing powers of p. The first two return pointers. See sage/rings/padics/pow_computer_ext for examples and important warnings.
  - prime_pow.get_context, prime_pow.get_context_capdiv, prime_pow.get_top_context – obtain an ntl_ZZ_pContext_class corresponding to p^n. The capdiv version divides by prime_pow.e as appropriate. top_context corresponds to $p^{\text{prec\_cap}}$.
  - prime_pow.restore_context, prime_pow.restore_context_capdiv, prime_pow.restore_top_context – restores the given context.
  - prime_pow.get_modulus, get_modulus_capdiv, get_top_modulus – Returns a ZZ_pX_Modulus_c* pointing to a polynomial modulus defined modulo p^n (appropriately divided by prime_pow.e in the capdiv case).

EXAMPLES:

An Eisenstein extension:

```
sage: R = ZpCA(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f); W
5-adic Eisenstein Extension Ring in w defined by x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: z = (1+w)^5; z
1 + w^5 + w^6 + 2*w^7 + 4*w^8 + 3*w^10 + w^12 + 4*w^13 + 4*w^14 + 4*w^15
  + 4*w^16 + 4*w^17 + 4*w^20 + w^21 + 4*w^24 + O(w^25)
sage: y = z >> 1; y
w^4 + w^5 + 2*w^6 + 4*w^7 + 3*w^9 + w^11 + 4*w^12 + 4*w^13 + 4*w^14 + 4*w^15
  + 4*w^16 + 4*w^19 + w^20 + 4*w^23 + O(w^24)
sage: y.valuation()
4
sage: y.precision_relative()
20
sage: y.precision_absolute()
24
sage: z - (y << 1)
1 + O(w^25)
sage: (1/w)^12+w
w^-12 + w + O(w^12)
sage: (1/w).parent()
5-adic Eisenstein Extension Field in w defined by x^5 + 75*x^3 - 15*x^2 + 125*x - 5
```

An unramified extension:

```
sage: # needs sage.libs.flint
sage: g = x^3 + 3*x + 3
sage: A.<a> = R.ext(g)
sage: z = (1+a)^5; z
(2*a^2 + 4*a) + (3*a^2 + 3*a + 1)*5 + (4*a^2 + 3*a + 4)*5^2
+ (4*a^2 + 4*a + 4)*5^3 + (4*a^2 + 4*a + 4)*5^4 + O(5^5)
sage: z - 1 - 5*a - 10*a^2 - 10*a^3 - 5*a^4 - a^5
O(5^5)
sage: y = z >> 1; y
(3*a^2 + 3*a + 1) + (4*a^2 + 3*a + 4)*5 + (4*a^2 + 4*a + 4)*5^2
+ (4*a^2 + 4*a + 4)*5^3 + O(5^4)
sage: 1/a
(3*a^2 + 4) + (a^2 + 4)*5 + (3*a^2 + 4)*5^2 + (a^2 + 4)*5^3 + (3*a^2 + 4)*5^4 + O(5^5)
sage: FFA = A.residue_field()
sage: a0 = FFA.gen(); A(a0^3)
(2*a + 2) + O(5)
```

Different printing modes:

```
sage: # needs sage.libs.flint
sage: R = ZpCA(5, print_mode='digits'); S.<x> = ZZ[]; f = x^5 + 75*x^3 - 15*x^2 + 125*x -
↪5; W.<w> = R.ext(f)
sage: z = (1+w)^5; repr(z)
'...
↪4110403113210310442221311242000111011201102002023303214332011214403232013144001400444441030421100001
↪'
sage: R = ZpCA(5, print_mode='bars'); S.<x> = ZZ[]; g = x^3 + 3*x + 3; A.<a> = R.ext(g)
```

(continues on next page)

(continued from previous page)

```
sage: z = (1+a)^5; repr(z)
'...[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4,␣
↪4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4,␣
↪4]|[4, 3, 4]|[1, 3, 3]|[0, 4, 2]'
sage: R = ZpCA(5, print_mode='terse'); S.<x> = ZZ[]; f = x^5 + 75*x^3 - 15*x^2 + 125*x -
↪5; W.<w> = R.ext(f)
sage: z = (1+w)^5; z
6 + 95367431640505*w + 25*w^2 + 95367431640560*w^3 + 5*w^4 + O(w^100)
sage: R = ZpCA(5, print_mode='val-unit'); S.<x> = ZZ[]; f = x^5 + 75*x^3 - 15*x^2 +␣
↪125*x -5; W.<w> = R.ext(f)
sage: y = (1+w)^5 - 1; y
w^5 * (2090041 + 19073486126901*w + 1258902*w^2 + 674*w^3 + 16785*w^4) + O(w^100)
```

You can get at the underlying ntl representation:

```
sage: # needs sage.libs.flint
sage: z._ntl_rep()
[6 95367431640505 25 95367431640560 5]
sage: y._ntl_rep()
[5 95367431640505 25 95367431640560 5]
sage: y._ntl_rep_abs()
([5 95367431640505 25 95367431640560 5], 0)
```

Note: If you get an error `internal error: can't grow this _ntl_gbigint`, it indicates that moduli are being mixed inappropriately somewhere.

For example, when calling a function with a `ZZ_pX_c` as an argument, it copies. If the modulus is not set to the modulus of the `ZZ_pX_c`, you can get errors.

AUTHORS:

- David Roe (2008-01-01): initial version
- Robert Harron (2011-09): fixes/enhancements
- Julian Rueth (2012-10-15): fixed an initialization bug

`sage.rings.padics.padic_ZZ_pX_CA_element.`**`make_ZZpXCAElement`**(*parent*, *value*, *absprec*, *version*)

For pickling. Makes a `pAdicZZpXCAElement` with given `parent`, `value`, `absprec`.

EXAMPLES:

```
sage: from sage.rings.padics.padic_ZZ_pX_CA_element import make_ZZpXCAElement
sage: R = ZpCA(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: make_ZZpXCAElement(W, ntl.ZZ_pX([3,2,4],5^3),13,0)
3 + 2*w + 4*w^2 + O(w^13)
```

class `sage.rings.padics.padic_ZZ_pX_CA_element.`**`pAdicZZpXCAElement`**

Bases: *pAdicZZpXElement*

Creates an element of a capped absolute precision, unramified or Eisenstein extension of Zp or Qp.

INPUT:

- `parent` – either an `EisensteinRingCappedAbsolute` or `UnramifiedRingCappedAbsolute`
- x – an integer, rational, p-adic element, polynomial, list, integer_mod, pari int/frac/poly_t/pol_mod, an `ntl_ZZ_pX`, an `ntl_ZZ`, an `ntl_ZZ_p`, an `ntl_ZZX`, or something convertible into parent.residue_field()
- `absprec` – an upper bound on the absolute precision of the element created
- `relprec` – an upper bound on the relative precision of the element created
- `empty` – whether to return after initializing to zero.

EXAMPLES:

```
sage: R = ZpCA(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: z = (1+w)^5; z  # indirect doctest
1 + w^5 + w^6 + 2*w^7 + 4*w^8 + 3*w^10 + w^12 + 4*w^13 + 4*w^14 + 4*w^15
  + 4*w^16 + 4*w^17 + 4*w^20 + w^21 + 4*w^24 + O(w^25)
sage: W(R(3,3))
3 + O(w^15)
sage: W(pari('3 + O(5^3)'))
3 + O(w^15)
sage: W(w, 14)
w + O(w^14)
```

expansion(*n=None*, *lift_mode='simple'*)

Return a list giving a series representation of `self`.

- If `lift_mode == 'simple'` or `'smallest'`, the returned list will consist of integers (in the Eisenstein case) or a list of lists of integers (in the unramified case). `self` can be reconstructed as a sum of elements of the list times powers of the uniformiser (in the Eisenstein case), or as a sum of powers of p times polynomials in the generator (in the unramified case).
  - If `lift_mode == 'simple'`, all integers will be in the interval $[0, p-1]$
  - If `lift_mod == 'smallest'` they will be in the interval $[(1-p)/2, p/2]$.
- If `lift_mode == 'teichmuller'`, returns a list of `pAdicZZpXCAElements`, all of which are Teichmuller representatives and such that `self` is the sum of that list times powers of the uniformizer.

INPUT:

- `n` – integer (default `None`). If given, returns the corresponding entry in the expansion.

EXAMPLES:

```
sage: R = ZpCA(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: y = W(775, 19); y
w^10 + 4*w^12 + 2*w^14 + w^15 + 2*w^16 + 4*w^17 + w^18 + O(w^19)
sage: (y>>9).expansion()
[0, 1, 0, 4, 0, 2, 1, 2, 4, 1]
sage: (y>>9).expansion(lift_mode='smallest')
[0, 1, 0, -1, 0, 2, 1, 2, 0, 1]
```

(continues on next page)

(continued from previous page)

```
sage: w^10 - w^12 + 2*w^14 + w^15 + 2*w^16 + w^18 + O(w^19)
w^10 + 4*w^12 + 2*w^14 + w^15 + 2*w^16 + 4*w^17 + w^18 + O(w^19)
sage: g = x^3 + 3*x + 3

sage: # needs sage.libs.flint
sage: A.<a> = R.ext(g)
sage: y = 75 + 45*a + 1200*a^2; y
4*a*5 + (3*a^2 + a + 3)*5^2 + 4*a^2*5^3 + a^2*5^4 + O(5^5)
sage: E = y.expansion(); E
5-adic expansion of 4*a*5 + (3*a^2 + a + 3)*5^2 + 4*a^2*5^3 + a^2*5^4 + O(5^5)
sage: list(E)
[[], [0, 4], [3, 1, 3], [0, 0, 4], [0, 0, 1]]
sage: list(y.expansion(lift_mode='smallest'))
[[], [0, -1], [-2, 2, -2], [1], [0, 0, 2]]
sage: 5*((-2*5 + 25) + (-1 + 2*5)*a + (-2*5 + 2*125)*a^2)
4*a*5 + (3*a^2 + a + 3)*5^2 + 4*a^2*5^3 + a^2*5^4 + O(5^5)
sage: W(0).expansion()
[]
sage: list(A(0,4).expansion())
[]
```

Check that github issue #25879 has been resolved:

```
sage: K = ZpCA(3,5)
sage: R.<a> = K[]
sage: L.<a> = K.extension(a^2 - 3)
sage: a.residue()
0
```

is_equal_to(*right*, *absprec=None*)

Returns whether `self` is equal to `right` modulo `self.uniformizer()^absprec`.

If `absprec` is `None`, returns if `self` is equal to `right` modulo the lower of their two precisions.

EXAMPLES:

```
sage: R = ZpCA(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(47); b = W(47 + 25)
sage: a.is_equal_to(b)
False
sage: a.is_equal_to(b, 7)
True
```

is_zero(*absprec=None*)

Return whether the valuation of `self` is at least `absprec`.

If `absprec` is `None`, returns if `self` is indistinguishable from zero.

If `self` is an inexact zero of valuation less than `absprec`, raises a `PrecisionError`.

EXAMPLES:

```
sage: R = ZpCA(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: O(w^189).is_zero()
True
sage: W(0).is_zero()
True
sage: a = W(675)
sage: a.is_zero()
False
sage: a.is_zero(7)
True
sage: a.is_zero(21)
False
```

lift_to_precision(*absprec=None*)

Returns a `pAdicZZpXCAElement` congruent to `self` but with absolute precision at least `absprec`.

INPUT:

- `absprec` – (default `None`) the absolute precision of the result. If `None`, lifts to the maximum precision allowed.

Note: If setting `absprec` that high would violate the precision cap, raises a precision error.

Note that the new digits will not necessarily be zero.

EXAMPLES:

```
sage: R = ZpCA(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(345, 17); a
4*w^5 + 3*w^7 + w^9 + 3*w^10 + 2*w^11 + 4*w^12 + w^13 + 2*w^14 + 2*w^15 + O(w^
↪17)
sage: b = a.lift_to_precision(19); b  # indirect doctest
4*w^5 + 3*w^7 + w^9 + 3*w^10 + 2*w^11 + 4*w^12 + w^13 + 2*w^14 + 2*w^15 + w^17␣
↪+ 2*w^18 + O(w^19)
sage: c = a.lift_to_precision(24); c
4*w^5 + 3*w^7 + w^9 + 3*w^10 + 2*w^11 + 4*w^12 + w^13 + 2*w^14 + 2*w^15 + w^17␣
↪+ 2*w^18 + 4*w^19 + 4*w^20 + 2*w^21 + 4*w^23 + O(w^24)
sage: a._ntl_rep()
[345]
sage: b._ntl_rep()
[345]
sage: c._ntl_rep()
[345]
sage: a.lift_to_precision().precision_absolute() == W.precision_cap()
True
```

matrix_mod_pn()

Return the matrix of right multiplication by the element on the power basis $1, x, x^2, \ldots, x^{d-1}$ for this extension field. Thus the *rows* of this matrix give the images of each of the x^i. The entries of the matrices are `IntegerMod` elements, defined modulo `p^(self.absprec() / e)`.

EXAMPLES:

```
sage: R = ZpCA(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: a = (3+w)^7
sage: a.matrix_mod_pn()                                                    #␣
↪needs sage.geometry.polyhedron
[2757  333 1068  725 2510]
[  50 1507  483  318  725]
[ 500   50 3007 2358  318]
[1590 1375 1695 1032 2358]
[2415  590 2370 2970 1032]
```

polynomial(*var='x'*)

Return a polynomial over the base ring that yields this element when evaluated at the generator of the parent.

INPUT:

- `var` – string, the variable name for the polynomial

EXAMPLES:

```
sage: S.<x> = ZZ[]
sage: W.<w> = ZpCA(5).extension(x^2 - 5)
sage: (w + W(5, 7)).polynomial()
(1 + O(5^3))*x + 5 + O(5^4)
```

precision_absolute()

Returns the absolute precision of `self`, ie the power of the uniformizer modulo which this element is defined.

EXAMPLES:

```
sage: R = ZpCA(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(75, 19); a
3*w^10 + 2*w^12 + w^14 + w^16 + w^17 + 3*w^18 + O(w^19)
sage: a.valuation()
10
sage: a.precision_absolute()
19
sage: a.precision_relative()
9
sage: a.unit_part()
3 + 2*w^2 + w^4 + w^6 + w^7 + 3*w^8 + O(w^9)
```

precision_relative()

Returns the relative precision of `self`, ie the power of the uniformizer modulo which the unit part of `self` is defined.

EXAMPLES:

```
sage: R = ZpCA(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(75, 19); a
3*w^10 + 2*w^12 + w^14 + w^16 + w^17 + 3*w^18 + O(w^19)
sage: a.valuation()
10
sage: a.precision_absolute()
19
sage: a.precision_relative()
9
sage: a.unit_part()
3 + 2*w^2 + w^4 + w^6 + w^7 + 3*w^8 + O(w^9)
```

teichmuller_expansion(*n=None*)

Returns a list $[a_0, a_1, \ldots, a_n]$ such that

- $a_i^q = a_i$
- `self.unit_part()` = $\sum_{i=0}^{n} a_i \pi^i$, where π is a uniformizer of self.parent()
- if $a_i \neq 0$, the absolute precision of a_i is `self.precision_relative() - i`

INPUT:

- `n` – integer (default `None`). If given, returns the corresponding entry in the expansion.

EXAMPLES:

```
sage: R.<a> = Zq(5^4,4)
sage: E = a.teichmuller_expansion(); E
5-adic expansion of a + O(5^4) (teichmuller)
sage: list(E)
[a + (2*a^3 + 2*a^2 + 3*a + 4)*5 + (4*a^3 + 3*a^2 + 3*a + 2)*5^2 + (4*a^2 + 2*a␣
↪+ 2)*5^3 + O(5^4),
 (3*a^3 + 3*a^2 + 2*a + 1) + (a^3 + 4*a^2 + 1)*5 + (a^2 + 4*a + 4)*5^2 + O(5^3),
 (4*a^3 + 2*a^2 + a + 1) + (2*a^3 + 2*a^2 + 2*a + 4)*5 + O(5^2),
 (a^3 + a^2 + a + 4) + O(5)]
sage: sum([c * 5^i for i, c in enumerate(E)])
a + O(5^4)
sage: all(c^625 == c for c in E)
True

sage: S.<x> = ZZ[]
sage: f = x^3 - 98*x + 7
sage: W.<w> = ZpCA(7,3).ext(f)
sage: b = (1+w)^5; L = b.teichmuller_expansion(); L
[1 + O(w^9), 5 + 5*w^3 + w^6 + 4*w^7 + O(w^8), 3 + 3*w^3 + O(w^7),
 3 + 3*w^3 + O(w^6), O(w^5), 4 + 5*w^3 + O(w^4), 3 + O(w^3), 6 + O(w^2), 6 +␣
↪O(w)]
sage: sum([w^i*L[i] for i in range(9)]) == b
True
sage: all(L[i]^(7^3) == L[i] for i in range(9))
```

(continues on next page)

(continued from previous page)

```
True

sage: L = W(3).teichmuller_expansion(); L
[3 + 3*w^3 + w^7 + O(w^9), O(w^8), O(w^7), 4 + 5*w^3 + O(w^6), O(w^5),
 O(w^4), 3 + O(w^3), 6 + O(w^2)]
sage: sum([w^i*L[i] for i in range(len(L))])
3 + O(w^9)
```

to_fraction_field()

Returns `self` cast into the fraction field of `self.parent()`.

EXAMPLES:

```
sage: R = ZpCA(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: z = (1 + w)^5; z
1 + w^5 + w^6 + 2*w^7 + 4*w^8 + 3*w^10 + w^12 + 4*w^13 + 4*w^14 + 4*w^15 + 4*w^
↪16 + 4*w^17 + 4*w^20 + w^21 + 4*w^24 + O(w^25)
sage: y = z.to_fraction_field(); y  # indirect doctest
1 + w^5 + w^6 + 2*w^7 + 4*w^8 + 3*w^10 + w^12 + 4*w^13 + 4*w^14 + 4*w^15 + 4*w^
↪16 + 4*w^17 + 4*w^20 + w^21 + 4*w^24 + O(w^25)
sage: y.parent()
5-adic Eisenstein Extension Field in w defined by x^5 + 75*x^3 - 15*x^2 + 125*x␣
↪- 5
```

unit_part()

Returns the unit part of `self`, ie `self / uniformizer^(self.valuation())`

EXAMPLES:

```
sage: R = ZpCA(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(75, 19); a
3*w^10 + 2*w^12 + w^14 + w^16 + w^17 + 3*w^18 + O(w^19)
sage: a.valuation()
10
sage: a.precision_absolute()
19
sage: a.precision_relative()
9
sage: a.unit_part()
3 + 2*w^2 + w^4 + w^6 + w^7 + 3*w^8 + O(w^9)
```

CHAPTER

TWENTYONE

P-ADIC ZZ_PX FM ELEMENT

This file implements elements of Eisenstein and unramified extensions of $\mathbf{Z}_p$ with fixed modulus precision.

For the parent class see padic_extension_leaves.pyx.

The underlying implementation is through NTL's ZZ_pX class. Each element contains the following data:

- value (ZZ_pX_c) – An ntl ZZ_pX storing the value. The variable x is the uniformizer in the case of Eisenstein extensions. This ZZ_pX is created with global ntl modulus determined by the parent's precision cap and shared among all elements.
- prime_pow (some subclass of PowComputer_ZZ_pX) – a class, identical among all elements with the same parent, holding common data.
  - prime_pow.deg – the degree of the extension
  - prime_pow.e – the ramification index
  - prime_pow.f – the inertia degree
  - prime_pow.prec_cap – the unramified precision cap: for Eisenstein extensions this is the smallest power of p that is zero
  - prime_pow.ram_prec_cap – the ramified precision cap: for Eisenstein extensions this will be the smallest power of x that is indistinguishable from zero
  - prime_pow.pow_ZZ_tmp, prime_pow.pow_mpz_t_tmp``, prime_pow.pow_Integer – functions for accessing powers of p. The first two return pointers. See sage/rings/padics/pow_computer_ext for examples and important warnings.
  - prime_pow.get_context, prime_pow.get_context_capdiv, prime_pow.get_top_context – obtain an ntl_ZZ_pContext_class corresponding to p^n. The capdiv version divides by prime_pow.e as appropriate. top_context corresponds to $p^{\text{prec\_cap}}$.
  - prime_pow.restore_context, prime_pow.restore_context_capdiv, prime_pow.restore_top_context – restores the given context
  - prime_pow.get_modulus, get_modulus_capdiv, get_top_modulus – Returns a ZZ_pX_Modulus_c* pointing to a polynomial modulus defined modulo p^n (appropriately divided by prime_pow.e in the capdiv case).

EXAMPLES:

An Eisenstein extension:

```
sage: R = ZpFM(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f); W
```

(continues on next page)

(continued from previous page)

```
5-adic Eisenstein Extension Ring in w defined by x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: z = (1+w)^5; z
1 + w^5 + w^6 + 2*w^7 + 4*w^8 + 3*w^10 + w^12 + 4*w^13 + 4*w^14 + 4*w^15 + 4*w^16 + 4*w^
↪17 + 4*w^20 + w^21 + 4*w^24
sage: y = z >> 1; y
w^4 + w^5 + 2*w^6 + 4*w^7 + 3*w^9 + w^11 + 4*w^12 + 4*w^13 + 4*w^14 + 4*w^15 + 4*w^16 +␣
↪4*w^19 + w^20 + 4*w^23 + 4*w^24
sage: y.valuation()
4
sage: y.precision_relative()
21
sage: y.precision_absolute()
25
sage: z - (y << 1)
1
```

An unramified extension:

```
sage: # needs sage.libs.flint
sage: g = x^3 + 3*x + 3
sage: A.<a> = R.ext(g)
sage: z = (1+a)^5; z
(2*a^2 + 4*a) + (3*a^2 + 3*a + 1)*5 + (4*a^2 + 3*a + 4)*5^2 + (4*a^2 + 4*a + 4)*5^3 +␣
↪(4*a^2 + 4*a + 4)*5^4
sage: z - 1 - 5*a - 10*a^2 - 10*a^3 - 5*a^4 - a^5
0
sage: y = z >> 1; y
(3*a^2 + 3*a + 1) + (4*a^2 + 3*a + 4)*5 + (4*a^2 + 4*a + 4)*5^2 + (4*a^2 + 4*a + 4)*5^3
sage: 1/a
(3*a^2 + 4) + (a^2 + 4)*5 + (3*a^2 + 4)*5^2 + (a^2 + 4)*5^3 + (3*a^2 + 4)*5^4
```

Different printing modes:

```
sage: # needs sage.libs.flint
sage: R = ZpFM(5, print_mode='digits'); S.<x> = R[]; f = x^5 + 75*x^3 - 15*x^2 + 125*x -
↪5; W.<w> = R.ext(f)
sage: z = (1+w)^5; repr(z)
'...
↪4110403113210310442221311242000111011201102002023303214332011214403232013144001400444441030421100001
↪'
sage: R = ZpFM(5, print_mode='bars'); S.<x> = R[]; g = x^3 + 3*x + 3; A.<a> = R.ext(g)
sage: z = (1+a)^5; repr(z)
'...[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4,␣
↪4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4,␣
↪4]|[4, 3, 4]|[1, 3, 3]|[0, 4, 2]'
sage: R = ZpFM(5, print_mode='terse'); S.<x> = R[]; f = x^5 + 75*x^3 - 15*x^2 + 125*x -5;
↪ W.<w> = R.ext(f)
sage: z = (1+w)^5; z
6 + 95367431640505*w + 25*w^2 + 95367431640560*w^3 + 5*w^4
sage: R = ZpFM(5, print_mode='val-unit'); S.<x> = R[]; f = x^5 + 75*x^3 - 15*x^2 + 125*x␣
↪-5; W.<w> = R.ext(f)
sage: y = (1+w)^5 - 1; y
w^5 * (2090041 + 19073486126901*w + 1258902*w^2 + 57220458985049*w^3 + 16785*w^4)
```

AUTHORS:

- David Roe (2008-01-01) initial version

sage.rings.padics.padic_ZZ_pX_FM_element.**make_ZZpXFMElement**(*parent*, *f*)

Create a new pAdicZZpXFMElement out of an ntl_ZZ_pX f, with parent parent. For use with pickling.

EXAMPLES:

```
sage: R = ZpFM(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: z = (1 + w)^5 - 1
sage: loads(dumps(z)) == z  # indirect doctest
True
```

class sage.rings.padics.padic_ZZ_pX_FM_element.**pAdicZZpXFMElement**

Bases: *pAdicZZpXElement*

Creates an element of a fixed modulus, unramified or eisenstein extension of $\mathbf{Z}_p$ or $\mathbf{Q}_p$.

INPUT:

- parent – either an EisensteinRingFixedMod or UnramifiedRingFixedMod
- x – an integer, rational, p-adic element, polynomial, list, integer_mod, pari int/frac/poly_t/pol_mod, an ntl_ZZ_pX, an ntl_ZZX, an ntl_ZZ, or an ntl_ZZ_p
- absprec – not used
- relprec – not used
- empty – whether to return after initializing to zero (without setting anything)

EXAMPLES:

```
sage: R = ZpFM(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: z = (1+w)^5; z  # indirect doctest
1 + w^5 + w^6 + 2*w^7 + 4*w^8 + 3*w^10 + w^12 + 4*w^13 + 4*w^14 + 4*w^15 + 4*w^16 +
↪4*w^17 + 4*w^20 + w^21 + 4*w^24
```

add_bigoh(*absprec*)

Return a new element truncated modulo π^{absprec}.

This is only implemented for unramified extension at this point.

INPUT:

- absprec – an integer

OUTPUT:

A new element truncated modulo π^{absprec}.

EXAMPLES:

```
sage: R = Zp(7,4,'fixed-mod')
sage: a = R(1 + 7 + 7^2)
sage: a.add_bigoh(1)
1
```

expansion(*n=None*, *lift_mode='simple'*)

Return a list giving a series representation of this element.

- If `lift_mode == 'simple' or 'smallest'`, the returned list will consist of
  - integers (in the eisenstein case) or
  - lists of integers (in the unramified case).
- this element can be reconstructed as
  - a sum of elements of the list times powers of the uniformiser (in the eisenstein case), or
  - as a sum of powers of the p times polynomials in the generator (in the unramified case).
- If `lift_mode == 'simple'`, all integers will be in the range $[0, p-1]$,
- If `lift_mode == 'smallest'` they will be in the range $[(1-p)/2, p/2]$.
- If `lift_mode == 'teichmuller'`, returns a list of `pAdicZZpXCRElements`, all of which are Teichmuller representatives and such that this element is the sum of that list times powers of the uniformizer.

INPUT:

- `n` – integer (default `None`); if given, returns the corresponding entry in the expansion

EXAMPLES:

```
sage: R = ZpFM(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: y = W(775); y
w^10 + 4*w^12 + 2*w^14 + w^15 + 2*w^16 + 4*w^17 + w^18 + w^20 + 2*w^21 + 3*w^22␣
↪+ w^23 + w^24
sage: (y>>9).expansion()
[0, 1, 0, 4, 0, 2, 1, 2, 4, 1, 0, 1, 2, 3, 1, 1, 4, 1, 2, 4, 1, 0, 0, 3]
sage: (y>>9).expansion(lift_mode='smallest')
[0, 1, 0, -1, 0, 2, 1, 2, 0, 1, 2, 1, 1, -1, -1, 2, -2, 0, -2, -2, -2, 0, -2, -
↪2, 2]
sage: w^10 - w^12 + 2*w^14 + w^15 + 2*w^16 + w^18 + 2*w^19 + w^20 + w^21 - w^22␣
↪- w^23 + 2*w^24
w^10 + 4*w^12 + 2*w^14 + w^15 + 2*w^16 + 4*w^17 + w^18 + w^20 + 2*w^21 + 3*w^22␣
↪+ w^23 + w^24
sage: g = x^3 + 3*x + 3

sage: # needs sage.libs.flint
sage: A.<a> = R.ext(g)
sage: y = 75 + 45*a + 1200*a^2; y
4*a*5 + (3*a^2 + a + 3)*5^2 + 4*a^2*5^3 + a^2*5^4
sage: E = y.expansion(); E
5-adic expansion of 4*a*5 + (3*a^2 + a + 3)*5^2 + 4*a^2*5^3 + a^2*5^4
sage: list(E)
```

(continues on next page)

(continued from previous page)

```
[[], [0, 4], [3, 1, 3], [0, 0, 4], [0, 0, 1]]
sage: list(y.expansion(lift_mode='smallest'))
[[], [0, -1], [-2, 2, -2], [1], [0, 0, 2]]
sage: 5*((-2*5 + 25) + (-1 + 2*5)*a + (-2*5 + 2*125)*a^2)
4*a*5 + (3*a^2 + a + 3)*5^2 + 4*a^2*5^3 + a^2*5^4
sage: W(0).expansion()
[]
sage: list(A(0,4).expansion())
[]
```

Check that github issue #25879 has been resolved:

```
sage: K = ZpCA(3,5)
sage: R.<a> = K[]
sage: L.<a> = K.extension(a^2 - 3)
sage: a.residue()
0
```

is_equal_to(*right*, *absprec=None*)

Return whether `self` is equal to `right` modulo `self.uniformizer()^absprec`.

If `absprec` is `None`, returns if `self` is equal to `right` modulo the precision cap.

EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(47); b = W(47 + 25)
sage: a.is_equal_to(b)
False
sage: a.is_equal_to(b, 7)
True
```

is_zero(*absprec=None*)

Return whether the valuation of `self` is at least `absprec`; if `absprec` is `None`, return whether `self` is indistinguishable from zero.

EXAMPLES:

```
sage: R = ZpFM(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: O(w^189).is_zero()
True
sage: W(0).is_zero()
True
sage: a = W(675)
sage: a.is_zero()
False
sage: a.is_zero(7)
True
```

(continues on next page)

(continued from previous page)

```
sage: a.is_zero(21)
False
```

lift_to_precision(*absprec=None*)

Return self.

EXAMPLES:

```
sage: R = ZpFM(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: w.lift_to_precision(10000)
w
```

matrix_mod_pn()

Return the matrix of right multiplication by the element on the power basis $1, x, x^2, \ldots, x^{d-1}$ for this extension field.

The **rows** of this matrix give the images of each of the x^i. The entries of the matrices are IntegerMod elements, defined modulo p^(self.absprec() / e).

Raises an error if self has negative valuation.

EXAMPLES:

```
sage: R = ZpFM(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: a = (3+w)^7
sage: a.matrix_mod_pn()                                                    #␣
↪needs sage.geometry.polyhedron
[2757  333 1068  725 2510]
[  50 1507  483  318  725]
[ 500   50 3007 2358  318]
[1590 1375 1695 1032 2358]
[2415  590 2370 2970 1032]
```

norm(*base=None*)

Return the absolute or relative norm of this element.

Note: This is not the p-adic absolute value. This is a field theoretic norm down to a ground ring.

If you want the p-adic absolute value, use the abs() function instead.

If K is given then K must be a subfield of the parent L of self, in which case the norm is the relative norm from L to K. In all other cases, the norm is the absolute norm down to $\mathbf{Q}_p$ or $\mathbf{Z}_p$.

EXAMPLES:

```
sage: R = ZpCR(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
```

(continues on next page)

(continued from previous page)

```
sage: W.<w> = R.ext(f)
sage: ((1+2*w)^5).norm()
1 + 5^2 + O(5^5)
sage: ((1+2*w)).norm()^5
1 + 5^2 + O(5^5)
```

polynomial(*var='x'*)

Return a polynomial over the base ring that yields this element when evaluated at the generator of the parent.

INPUT:

- `var` – string, the variable name for the polynomial

EXAMPLES:

```
sage: S.<x> = ZZ[]
sage: W.<w> = ZpFM(5).extension(x^2 - 5)
sage: (w + 5).polynomial()
x + 5
```

precision_absolute()

Return the absolute precision of `self`, ie the precision cap of `self.parent()`.

EXAMPLES:

```
sage: R = ZpFM(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(75); a
3*w^10 + 2*w^12 + w^14 + w^16 + w^17 + 3*w^18 + 3*w^19 + 2*w^21 + 3*w^22 + 3*w^
↪23
sage: a.valuation()
10
sage: a.precision_absolute()
25
sage: a.precision_relative()
15
sage: a.unit_part()
3 + 2*w^2 + w^4 + w^6 + w^7 + 3*w^8 + 3*w^9 + 2*w^11 + 3*w^12
 + 3*w^13 + w^15 + 4*w^16 + 2*w^17 + w^18 + 3*w^21 + w^22 + 3*w^24
```

precision_relative()

Return the relative precision of `self`, ie the precision cap of `self.parent()` minus the `valuation of self`.

EXAMPLES:

```
sage: R = ZpFM(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(75); a
3*w^10 + 2*w^12 + w^14 + w^16 + w^17 + 3*w^18 + 3*w^19 + 2*w^21 + 3*w^22 + 3*w^
```

(continues on next page)

(continued from previous page)

```
↪23
sage: a.valuation()
10
sage: a.precision_absolute()
25
sage: a.precision_relative()
15
sage: a.unit_part()
3 + 2*w^2 + w^4 + w^6 + w^7 + 3*w^8 + 3*w^9 + 2*w^11 + 3*w^12
 + 3*w^13 + w^15 + 4*w^16 + 2*w^17 + w^18 + 3*w^21 + w^22 + 3*w^24
```

teichmuller_expansion(*n=None*)

Return a list $[a_0, a_1, \ldots, a_n]$ such that

- $a_i^q = a_i$
- `self.unit_part()` = $\sum_{i=0}^{n} a_i \pi^i$, where π is a uniformizer of `self.parent()`

INPUT:

- `n` – integer (default `None`); f given, returns the corresponding entry in the expansion

EXAMPLES:

```
sage: # needs sage.libs.flint
sage: R.<a> = ZqFM(5^4,4)
sage: E = a.teichmuller_expansion(); E
5-adic expansion of a (teichmuller)
sage: list(E)
[a + (2*a^3 + 2*a^2 + 3*a + 4)*5 + (4*a^3 + 3*a^2 + 3*a + 2)*5^2 + (4*a^2 + 2*a␣
↪+ 2)*5^3,
 (3*a^3 + 3*a^2 + 2*a + 1) + (a^3 + 4*a^2 + 1)*5 + (a^2 + 4*a + 4)*5^2 + (4*a^2␣
↪+ a + 3)*5^3,
 (4*a^3 + 2*a^2 + a + 1) + (2*a^3 + 2*a^2 + 2*a + 4)*5 + (3*a^3 + 2*a^2 + a +␣
↪1)*5^2 + (a^3 + a^2 + 2)*5^3,
 (a^3 + a^2 + a + 4) + (3*a^3 + 1)*5 + (3*a^3 + a + 2)*5^2 + (3*a^3 + 3*a^2 +␣
↪3*a + 1)*5^3]
sage: sum([c * 5^i for i, c in enumerate(E)])
a
sage: all(c^625 == c for c in E)
True

sage: S.<x> = ZZ[]
sage: f = x^3 - 98*x + 7
sage: W.<w> = ZpFM(7,3).ext(f)
sage: b = (1+w)^5; L = b.teichmuller_expansion(); L
[1,
 5 + 5*w^3 + w^6 + 4*w^7,
 3 + 3*w^3 + w^7,
 3 + 3*w^3 + w^7,
 0,
 4 + 5*w^3 + w^6 + 4*w^7,
 3 + 3*w^3 + w^7,
 6 + w^3 + 5*w^7,
```

(continues on next page)

(continued from previous page)

```
 6 + w^3 + 5*w^7]
sage: sum([w^i*L[i] for i in range(len(L))]) == b
True
sage: all(L[i]^(7^3) == L[i] for i in range(9))
True

sage: L = W(3).teichmuller_expansion(); L
[3 + 3*w^3 + w^7,
 0,
 0,
 4 + 5*w^3 + w^6 + 4*w^7,
 0,
 0,
 3 + 3*w^3 + w^7,
 6 + w^3 + 5*w^7]
sage: sum([w^i*L[i] for i in range(len(L))])
3
```

trace(*base=None*)

Return the absolute or relative trace of this element.

If K is given then K must be a subfield of the parent L of `self`, in which case the norm is the relative norm from L to K. In all other cases, the norm is the absolute norm down to $\mathbf{Q}_p$ or $\mathbf{Z}_p$.

EXAMPLES:

```
sage: R = ZpCR(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: a = (2+3*w)^7
sage: b = (6+w^3)^5
sage: a.trace()
3*5 + 2*5^2 + 3*5^3 + 2*5^4 + O(5^5)
sage: a.trace() + b.trace()
4*5 + 5^2 + 5^3 + 2*5^4 + O(5^5)
sage: (a+b).trace()
4*5 + 5^2 + 5^3 + 2*5^4 + O(5^5)
```

unit_part()

Return the unit part of `self`, ie `self / uniformizer^(self.valuation())`

> **Warning:** If this element has positive valuation then the unit part is not defined to the full precision of the ring. Asking for the unit part of `ZpFM(5)(0)` will not raise an error, but rather return itself.

EXAMPLES:

```
sage: R = ZpFM(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 + 125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(75); a
```

(continues on next page)

(continued from previous page)

```
3*w^10 + 2*w^12 + w^14 + w^16 + w^17 + 3*w^18 + 3*w^19 + 2*w^21 + 3*w^22 + 3*w^
↪23
sage: a.valuation()
10
sage: a.precision_absolute()
25
sage: a.precision_relative()
15
sage: a.unit_part()
3 + 2*w^2 + w^4 + w^6 + w^7 + 3*w^8 + 3*w^9 + 2*w^11 + 3*w^12
 + 3*w^13 + w^15 + 4*w^16 + 2*w^17 + w^18 + 3*w^21 + w^22 + 3*w^24
```

The unit part inserts nonsense digits if this element has positive valuation:

```
sage: (a-a).unit_part()
0
```

CHAPTER

TWENTYTWO

POWCOMPUTER

A class for computing and caching powers of the same integer.

This class is designed to be used as a field of p-adic rings and fields. Since elements of p-adic rings and fields need to use powers of p over and over, this class precomputes and stores powers of p. There is no reason that the base has to be prime however.

EXAMPLES:

```
sage: X = PowComputer(3, 4, 10)
sage: X(3)
27
sage: X(10) == 3^10
True
```

AUTHORS:

- David Roe

`sage.rings.padics.pow_computer.`**`PowComputer`**(*m*, *cache_limit*, *prec_cap*, *in_field=False*, *prec_type=None*)

Returns a PowComputer that caches the values $1, m, m^2, \ldots, m^C$, where C is `cache_limit`.

Once you create a PowComputer, merely call it to get values out.

You can input any integer, even if it's outside of the precomputed range.

INPUT:

- m – An integer, the base that you want to exponentiate.
- cache_limit – A positive integer that you want to cache powers up to.

EXAMPLES:

```
sage: PC = PowComputer(3, 5, 10)
sage: PC
PowComputer for 3
sage: PC(4)
81
sage: PC(6)
729
sage: PC(-1)
1/3
```

class `sage.rings.padics.pow_computer.`**`PowComputer_base`**

Bases: *PowComputer_class*

Initialization.

class sage.rings.padics.pow_computer.**PowComputer_class**

Bases: SageObject

Initializes self.

INPUT:

- prime – the prime that is the base of the exponentials stored in this pow_computer.
- cache_limit – how high to cache powers of prime.
- prec_cap – data stored for p-adic elements using this pow_computer (so they have C-level access to fields common to all elements of the same parent).
- ram_prec_cap – prec_cap * e
- in_field – same idea as prec_cap
- poly – same idea as prec_cap
- shift_seed – same idea as prec_cap

EXAMPLES:

```
sage: PC = PowComputer(3, 5, 10)
sage: PC.pow_Integer_Integer(2)
9
```

pow_Integer_Integer(n)

Tests the pow_Integer function.

EXAMPLES:

```
sage: PC = PowComputer(3, 5, 10)
sage: PC.pow_Integer_Integer(4)
81
sage: PC.pow_Integer_Integer(6)
729
sage: PC.pow_Integer_Integer(0)
1
sage: PC.pow_Integer_Integer(10)
59049

sage: # needs sage.libs.ntl
sage: PC = PowComputer_ext_maker(3, 5, 10, 20, False, ntl.ZZ_pX([-3,0,1], 3^10),
↪ 'big','e',ntl.ZZ_pX([1],3^10))
sage: PC.pow_Integer_Integer(4)
81
sage: PC.pow_Integer_Integer(6)
729
sage: PC.pow_Integer_Integer(0)
1
sage: PC.pow_Integer_Integer(10)
59049
```

CHAPTER

TWENTYTHREE

POWCOMPUTER_EXT

The classes in this file are designed to be attached to p-adic parents and elements for Cython access to properties of the parent.

In addition to storing the defining polynomial (as an NTL polynomial) at different precisions, they also cache powers of p and data to speed right shifting of elements.

The hierarchy of PowComputers splits first at whether it's for a base ring (Qp or Zp) or an extension.

Among the extension classes (those in this file), they are first split by the type of NTL polynomial (ntl_ZZ_pX or ntl_ZZ_pEX), then by the amount and style of caching (see below). Finally, there are subclasses of the ntl_ZZ_pX PowComputers that cache additional information for Eisenstein extensions.

There are three styles of caching:

- FM: caches powers of p up to the cache_limit, only caches the polynomial modulus and the ntl_ZZ_pContext of precision prec_cap.
- small: Requires cache_limit = prec_cap. Caches p^k for every k up to the cache_limit and caches a polynomial modulus and a ntl_ZZ_pContext for each such power of p.
- big: Caches as the small does up to cache_limit and caches prec_cap. Also has a dictionary that caches values above the cache_limit when they are computed (rather than at ring creation time).

AUTHORS:

- David Roe (2008-01-01) initial version

class sage.rings.padics.pow_computer_ext.**PowComputer_ZZ_pX**

Bases: *PowComputer_ext*

polynomial()

Returns the polynomial (with coefficient precision prec_cap) associated to this PowComputer.

The polynomial is output as an ntl_ZZ_pX.

EXAMPLES:

```
sage: PC = PowComputer_ext_maker(5, 5, 10, 20, False, ntl.ZZ_pX([-5,0,1],5^10),
↪'FM', 'e',ntl.ZZ_pX([1],5^10))
sage: PC.polynomial()
[9765620 0 1]
```

speed_test(*n*, *runs*)

Runs a speed test.

INPUT:

- n – input to a function to be tested (the function needs to be set in the source code).

- runs – The number of runs of that function

OUTPUT:

- The time in seconds that it takes to call the function on n, runs times.

EXAMPLES:

```
sage: PC = PowComputer_ext_maker(5, 10, 10, 20, False, ntl.ZZ_pX([-5, 0, 1], 5^
↪10), 'small', 'e',ntl.ZZ_pX([1],5^10))
sage: PC.speed_test(10, 10^6)  # random
0.0090679999999991878
```

class sage.rings.padics.pow_computer_ext.**PowComputer_ZZ_pX_FM**

Bases: *PowComputer_ZZ_pX*

This class only caches a context and modulus for p^prec_cap.

Designed for use with fixed modulus p-adic rings, in Eisenstein and unramified extensions of $\mathbf{Z}_p$.

class sage.rings.padics.pow_computer_ext.**PowComputer_ZZ_pX_FM_Eis**

Bases: *PowComputer_ZZ_pX_FM*

This class computes and stores low_shifter and high_shifter, which aid in right shifting elements.

class sage.rings.padics.pow_computer_ext.**PowComputer_ZZ_pX_big**

Bases: *PowComputer_ZZ_pX*

This class caches all contexts and moduli between 1 and cache_limit, and also caches for prec_cap. In addition, it stores a dictionary of contexts and moduli of

reset_dictionaries()

Resets the dictionaries. Note that if there are elements lying around that need access to these dictionaries, calling this function and then doing arithmetic with those elements could cause trouble (if the context object gets garbage collected for example. The bugs introduced could be very subtle, because NTL will generate a new context object and use it, but there's the potential for the object to be incompatible with the different context object).

EXAMPLES:

```
sage: A = PowComputer_ext_maker(5, 6, 10, 20, False, ntl.ZZ_pX([-5,0,1],5^10),
↪'big','e',ntl.ZZ_pX([1],5^10))
sage: P = A._get_context_test(8)
sage: A._context_dict()
{8: NTL modulus 390625}
sage: A.reset_dictionaries()
sage: A._context_dict()
{}
```

class sage.rings.padics.pow_computer_ext.**PowComputer_ZZ_pX_big_Eis**

Bases: *PowComputer_ZZ_pX_big*

This class computes and stores low_shifter and high_shifter, which aid in right shifting elements. These are only stored at maximal precision: in order to get lower precision versions just reduce mod p^n.

class sage.rings.padics.pow_computer_ext.**PowComputer_ZZ_pX_small**

Bases: *PowComputer_ZZ_pX*

This class caches contexts and moduli densely between 1 and cache_limit. It requires cache_limit == prec_cap.

It is intended for use with capped relative and capped absolute rings and fields, in Eisenstein and unramified extensions of the base p-adic fields.

class sage.rings.padics.pow_computer_ext.**PowComputer_ZZ_pX_small_Eis**

Bases: *PowComputer_ZZ_pX_small*

This class computes and stores low_shifter and high_shifter, which aid in right shifting elements. These are only stored at maximal precision: in order to get lower precision versions just reduce mod p^n.

class sage.rings.padics.pow_computer_ext.**PowComputer_ext**

Bases: *PowComputer_class*

sage.rings.padics.pow_computer_ext.**PowComputer_ext_maker**(*prime*, *cache_limit*, *prec_cap*, *ram_prec_cap*, *in_field*, *poly*, *prec_type='small'*, *ext_type='u'*, *shift_seed=None*)

Returns a PowComputer that caches the values $1, p, p^2, \ldots, p^C$, where C is `cache_limit`.

Once you create a PowComputer, merely call it to get values out. You can input any integer, even if it's outside of the precomputed range.

INPUT:

- `prime` – An integer, the base that you want to exponentiate.
- `cache_limit` – A positive integer that you want to cache powers up to.
- `prec_cap` – The cap on precisions of elements. For ramified extensions, p^((prec_cap - 1) // e) will be the largest power of p distinguishable from zero
- `in_field` – Boolean indicating whether this PowComputer is attached to a field or not.
- `poly` – An `ntl_ZZ_pX` or `ntl_ZZ_pEX` defining the extension. It should be defined modulo p^((prec_cap - 1) // e + 1)
- `prec_type` – 'FM', 'small', or 'big', defining how caching is done.
- `ext_type` – 'u' = unramified, 'e' = Eisenstein, 't' = two-step
- `shift_seed` – (required only for Eisenstein and two-step) For Eisenstein and two-step extensions, if f = a_n x^n - p a_{n-1} x^{n-1} - … - p a_0 with a_n a unit, then shift_seed should be 1/a_n (a_{n-1} x^{n-1} + … + a_0)

EXAMPLES:

```
sage: PC = PowComputer_ext_maker(5, 10, 10, 20, False, ntl.ZZ_pX([-5, 0, 1], 5^10),
↪'small','e',ntl.ZZ_pX([1],5^10))
sage: PC
PowComputer_ext for 5, with polynomial [9765620 0 1]
```

sage.rings.padics.pow_computer_ext.**ZZ_pX_eis_shift_test**(*_shifter*, *_a*, *_n*, *_finalprec*)

Shifts _a right _n x-adic digits, where x is considered modulo the polynomial in _shifter.

EXAMPLES:

```
sage: from sage.rings.padics.pow_computer_ext import ZZ_pX_eis_shift_test
sage: A = PowComputer_ext_maker(5, 3, 10, 40, False, ntl.ZZ_pX([-5,75,15,0,1],5^10),
↪ 'big', 'e',ntl.ZZ_pX([1,-15,-3],5^10))
sage: ZZ_pX_eis_shift_test(A, [0, 1], 1, 5)
[1]
```

(continues on next page)

(continued from previous page)

```
sage: ZZ_pX_eis_shift_test(A, [0, 0, 1], 1, 5)
[0 1]
sage: ZZ_pX_eis_shift_test(A, [5], 1, 5)
[75 15 0 1]
sage: ZZ_pX_eis_shift_test(A, [1], 1, 5)
[]
sage: ZZ_pX_eis_shift_test(A, [17, 91, 8, -2], 1, 5)
[316 53 3123 3]
sage: ZZ_pX_eis_shift_test(A, [316, 53, 3123, 3], -1, 5)
[15 91 8 3123]
sage: ZZ_pX_eis_shift_test(A, [15, 91, 8, 3123], 1, 5)
[316 53 3123 3]
```

CHAPTER

TWENTYFOUR

P-ADIC PRINTING

This file contains code for printing p-adic elements.

It has been moved here to prevent code duplication and make finding the relevant code easier.

AUTHORS:

- David Roe

sage.rings.padics.padic_printing.**pAdicPrinter**(*ring*, *options={}*)

Create a *pAdicPrinter*.

INPUT:

- ring – a p-adic ring or field.
- options – a dictionary, with keys in 'mode', 'pos', 'ram_name', 'unram_name', 'var_name', 'max_ram_terms', 'max_unram_terms', 'max_terse_terms', 'sep', 'alphabet'; see *pAdicPrinter_class* for the meanings of these keywords.

EXAMPLES:

```
sage: from sage.rings.padics.padic_printing import pAdicPrinter
sage: R = Zp(5)
sage: pAdicPrinter(R, {'sep': '&'})
series printer for 5-adic Ring with capped relative precision 20
```

class sage.rings.padics.padic_printing.**pAdicPrinterDefaults**(*mode='series'*, *pos=True*, *max_ram_terms=-1*, *max_unram_terms=-1*, *max_terse_terms=-1*, *sep='|'*, *alphabet=None*)

Bases: SageObject

This class stores global defaults for p-adic printing.

allow_negatives(*neg=None*)

Controls whether or not to display a balanced representation.

neg=None returns the current value.

EXAMPLES:

```
sage: padic_printing.allow_negatives(True)
sage: padic_printing.allow_negatives()
True
sage: Qp(29)(-1)
```

(continues on next page)

(continued from previous page)

```
-1 + O(29^20)
sage: Qp(29)(-1000)
-14 - 5*29 - 29^2 + O(29^20)
sage: padic_printing.allow_negatives(False)
```

alphabet(*alphabet=None*)

Controls the alphabet used to translate p-adic digits into strings (so that no separator need be used in `'digits'` mode).

`alphabet` should be passed in as a list or tuple.

`alphabet=None` returns the current value.

EXAMPLES:

```
sage: padic_printing.alphabet("abc")
sage: padic_printing.mode('digits')
sage: repr(Qp(3)(1234))
'...bcaacab'

sage: padic_printing.mode('series')
sage: padic_printing.alphabet(('0','1','2','3','4','5','6','7','8','9','A','B',
↪'C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V
↪','W','X','Y','Z','a','b','c','d','e','f','g','h','i','j','k','l','m','n','o',
↪'p','q','r','s','t','u','v','w','x','y','z'))
```

max_poly_terms(*max=None*)

Controls the number of terms appearing when printing polynomial representations in `'terse'` or `'val-unit'` modes.

`max=None` returns the current value.

`max=-1` encodes 'no limit.'

EXAMPLES:

```
sage: padic_printing.max_poly_terms(3)
sage: padic_printing.max_poly_terms()
3
sage: padic_printing.mode('terse')
sage: Zq(7^5, 5, names='a')([2,3,4])^8                                        #␣
↪needs sage.libs.ntl
2570 + 15808*a + 9018*a^2 + ... + O(7^5)

sage: padic_printing.max_poly_terms(-1)
sage: padic_printing.mode('series')
```

max_series_terms(*max=None*)

Controls the maximum number of terms shown when printing in `'series'`, `'digits'` or `'bars'` mode.

`max=None` returns the current value.

`max=-1` encodes 'no limit.'

EXAMPLES:

```
sage: padic_printing.max_series_terms(2)
sage: padic_printing.max_series_terms()
2
sage: Qp(31)(1000)
8 + 31 + ... + O(31^20)
sage: padic_printing.max_series_terms(-1)
sage: Qp(37)(100000)
26 + 37 + 36*37^2 + 37^3 + O(37^20)
```

max_unram_terms(*max=None*)

For rings with non-prime residue fields, controls how many terms appear in the coefficient of each `pi^n` when printing in `'series'` or `'bar'` modes.

`max=None` returns the current value.

`max=-1` encodes 'no limit.'

EXAMPLES:

```
sage: padic_printing.max_unram_terms(2)
sage: padic_printing.max_unram_terms()
2
sage: Zq(5^6, 5, names='a')([1,2,3,-1])^17                                   #␣
↪needs sage.libs.ntl
(3*a^4 + ... + 3) + (a^5 + ... + a)*5 + (3*a^3 + ... + 2)*5^2 + (3*a^5 + ... +␣
↪2)*5^3 + (4*a^5 + ... + 4)*5^4 + O(5^5)

sage: padic_printing.max_unram_terms(-1)
```

mode(*mode=None*)

Set the default printing mode.

`mode=None` returns the current value.

The allowed values for mode are: `'val-unit'`, `'series'`, `'terse'`, `'digits'` and `'bars'`.

EXAMPLES:

```
sage: padic_printing.mode('terse')
sage: padic_printing.mode()
'terse'
sage: Qp(7)(100)
100 + O(7^20)
sage: padic_printing.mode('series')
sage: Qp(11)(100)
1 + 9*11 + O(11^20)
sage: padic_printing.mode('val-unit')
sage: Qp(13)(130)
13 * 10 + O(13^21)
sage: padic_printing.mode('digits')
sage: repr(Qp(17)(100))
'...5F'
sage: repr(Qp(17)(1000))
'...37E'
sage: padic_printing.mode('bars')
sage: repr(Qp(19)(1000))
```

(continues on next page)

(continued from previous page)

```
'...2|14|12'

sage: padic_printing.mode('series')
```

sep(*sep=None*)

Controls the separator used in `'bars'` mode.

`sep=None` returns the current value.

EXAMPLES:

```
sage: padic_printing.sep('][')
sage: padic_printing.sep()
']['
sage: padic_printing.mode('bars')
sage: repr(Qp(61)(-1))
'...
↪60][60][60][60][60][60][60][60][60][60][60][60][60][60][60][60][60][60][60][60][60
↪'

sage: padic_printing.sep('|')
sage: padic_printing.mode('series')
```

class sage.rings.padics.padic_printing.**pAdicPrinter_class**

Bases: `SageObject`

This class stores the printing options for a specific p-adic ring or field, and uses these to compute the representations of elements.

dict()

Return a dictionary storing all of `self`'s printing options.

EXAMPLES:

```
sage: D = Zp(5)._printer.dict(); D['sep']
'|'
```

repr_gen(*elt*, *do_latex*, *pos=None*, *mode=None*, *ram_name=None*)

The entry point for printing an element.

INPUT:

- `elt` – a p-adic element of the appropriate ring to print.
- `do_latex` – whether to return a latex representation or a normal one.

EXAMPLES:

```
sage: R = Zp(5,5); P = R._printer; a = R(-5); a
4*5 + 4*5^2 + 4*5^3 + 4*5^4 + 4*5^5 + O(5^6)
sage: P.repr_gen(a, False, pos=False)
'-5 + O(5^6)'
sage: P.repr_gen(a, False, ram_name='p')
'4*p + 4*p^2 + 4*p^3 + 4*p^4 + 4*p^5 + O(p^6)'
```

richcmp_modes(*other*, *op*)

Return a comparison of the printing modes of `self` and `other`.

Return 0 if and only if all relevant modes are equal (`max_unram_terms` is irrelevant if the ring is totally ramified over the base, for example). This does not check if the rings are equal (to prevent infinite recursion in the comparison functions of p-adic rings), but it does check if the primes are the same (since the prime affects whether `pos` is relevant).

EXAMPLES:

```
sage: R = Qp(7, print_mode='digits', print_pos=True)
sage: S = Qp(7, print_mode='digits', print_pos=False)
sage: R._printer == S._printer
True
sage: R = Qp(7)
sage: S = Qp(7, print_mode='val-unit')
sage: R == S
False
sage: R._printer < S._printer
True
```

CHAPTER

TWENTYFIVE

PRECISION ERROR

The errors in this file indicate various styles of precision problems that can go wrong for p-adics and power series.

AUTHORS:

- David Roe

exception sage.rings.padics.precision_error.**PrecisionError**

Bases: ArithmeticError

CHAPTER

TWENTYSIX

MISCELLANEOUS FUNCTIONS

This file contains several miscellaneous functions used by p-adics.

- `gauss_sum` – compute Gauss sums using the Gross-Koblitz formula.
- `min` – a version of `min` that returns ∞ on empty input.
- `max` – a version of `max` that returns $-\infty$ on empty input.

AUTHORS:

- David Roe
- Adriana Salerno
- Ander Steele
- Kiran Kedlaya (modified gauss_sum 2017/09)

`sage.rings.padics.misc.`**`gauss_sum`**`(`*a*, *p*, *f*, *prec=20*, *factored=False*, *algorithm='pari'*, *parent=None*`)`

Return the Gauss sum $g_q(a)$ as a p-adic number.

The Gauss sum $g_q(a)$ is defined by

$$g_q(a) = \sum_{u \in F_q^*} \omega(u)^{-a} \zeta_q^u,$$

where $q = p^f$, ω is the Teichmüller character and ζ_q is some arbitrary choice of primitive q-th root of unity. The computation is adapted from the main theorem in Alain Robert's paper *The Gross-Koblitz formula revisited*, Rend. Sem. Mat. Univ. Padova 105 (2001), 157–170.

Let p be a prime, f a positive integer, $q = p^f$, and π be the unique root of $f(x) = x^{p-1} + p$ congruent to $\zeta_p - 1$ modulo $(\zeta_p - 1)^2$. Let $0 \le a < q - 1$. Then the Gross-Koblitz formula gives us the value of the Gauss sum $g_q(a)$ as a product of p-adic Gamma functions as follows:

$$g_q(a) = -\pi^s \prod_{0 \le i < f} \Gamma_p(a^{(i)}/(q-1)),$$

where s is the sum of the digits of a in base p and the $a^{(i)}$ have p-adic expansions obtained from cyclic permutations of that of a.

INPUT:

- `a` – integer
- `p` – prime
- `f` – positive integer
- `prec` – positive integer (optional, 20 by default)

- factored – boolean (optional, False by default)
- algorithm – flag passed to p-adic Gamma function (optional, "pari" by default)

OUTPUT:

If factored is False, returns a p-adic number in an Eisenstein extension of $\mathbf{Q}_p$. This number has the form $pi^e * z$ where pi is as above, e is some nonnegative integer, and z is an element of $\mathbf{Z}_p$; if factored is True, the pair (e, z) is returned instead, and the Eisenstein extension is not formed.

Note: This is based on GP code written by Adriana Salerno.

EXAMPLES:

In this example, we verify that $g_3(0) = -1$:

```
sage: from sage.rings.padics.misc import gauss_sum
sage: -gauss_sum(0, 3, 1)                                                   #␣
↪needs sage.libs.ntl sage.rings.padics
1 + O(pi^40)
```

Next, we verify that $g_5(a)g_5(-a) = 5(-1)^a$:

```
sage: from sage.rings.padics.misc import gauss_sum
sage: gauss_sum(2,5,1)^2 - 5                                                #␣
↪needs sage.libs.ntl
O(pi^84)
sage: gauss_sum(1,5,1)*gauss_sum(3,5,1) + 5                                 #␣
↪needs sage.libs.ntl
O(pi^84)
```

Finally, we compute a non-trivial value:

```
sage: from sage.rings.padics.misc import gauss_sum
sage: gauss_sum(2,13,2)                                                     #␣
↪needs sage.libs.ntl
6*pi^2 + 7*pi^14 + 11*pi^26 + 3*pi^62 + 6*pi^74 + 3*pi^86 + 5*pi^98 +
pi^110 + 7*pi^134 + 9*pi^146 + 4*pi^158 + 6*pi^170 + 4*pi^194 +
pi^206 + 6*pi^218 + 9*pi^230 + O(pi^242)
sage: gauss_sum(2,13,2, prec=5, factored=True)                              #␣
↪needs sage.rings.padics
(2, 6 + 6*13 + 10*13^2 + O(13^5))
```

See also:

- sage.arith.misc.gauss_sum() for general finite fields
- sage.modular.dirichlet.DirichletCharacter.gauss_sum() for prime finite fields
- sage.modular.dirichlet.DirichletCharacter.gauss_sum_numerical() for prime finite fields

sage.rings.padics.misc.**max**(*L)

Return the maximum of the inputs, where the maximum of the empty list is $-\infty$.

EXAMPLES:

```
sage: from sage.rings.padics.misc import max
sage: max()
-Infinity
sage: max(2,3)
3
```

sage.rings.padics.misc.**min**(**L*)

Return the minimum of the inputs, where the minimum of the empty list is ∞.

EXAMPLES:

```
sage: from sage.rings.padics.misc import min
sage: min()
+Infinity
sage: min(2,3)
2
```

sage.rings.padics.misc.**precprint**(*prec_type*, *prec_cap*, *p*)

String describing the precision mode on a p-adic ring or field.

EXAMPLES:

```
sage: from sage.rings.padics.misc import precprint
sage: precprint('capped-rel', 12, 2)
'with capped relative precision 12'
sage: precprint('capped-abs', 11, 3)
'with capped absolute precision 11'
sage: precprint('floating-point', 1234, 5)
'with floating precision 1234'
sage: precprint('fixed-mod', 1, 17)
'of fixed modulus 17^1'
```

sage.rings.padics.misc.**trim_zeros**(*L*)

Strips trailing zeros/empty lists from a list.

EXAMPLES:

```
sage: from sage.rings.padics.misc import trim_zeros
sage: trim_zeros([1,0,1,0])
[1, 0, 1]
sage: trim_zeros([[1],[],[2],[],[]])
[[1], [], [2]]
sage: trim_zeros([[],[]])
[]
sage: trim_zeros([])
[]
```

Zeros are also trimmed from nested lists (one deep):

sage: trim_zeros([[1,0]]) [[1]] sage: trim_zeros([[0],[1]]) [[], [1]]

CHAPTER TWENTYSEVEN

THE FUNCTIONS IN THIS FILE ARE USED IN CREATING NEW P-ADIC ELEMENTS.

When creating a p-adic element, the user can specify that the absolute precision be bounded and/or that the relative precision be bounded. Moreover, different p-adic parents impose their own bounds on the relative or absolute precision of their elements. The precision determines to what power of p the defining data will be reduced, but the valuation of the resulting element needs to be determined before the element is created. Moreover, some defining data can impose their own precision bounds on the result.

AUTHORS:

- David Roe (2012-03-01)

CHAPTER

TWENTYEIGHT

FROBENIUS ENDOMORPHISMS ON P-ADIC FIELDS

class sage.rings.padics.morphism.**FrobeniusEndomorphism_padics**

Bases: RingHomomorphism

A class implementing Frobenius endomorphisms on p-adic fields.

is_identity()

Return True if this morphism is the identity morphism.

EXAMPLES:

```
sage: K.<a> = Qq(5^3)
sage: Frob = K.frobenius_endomorphism()
sage: Frob.is_identity()
False
sage: (Frob^3).is_identity()
True
```

is_injective()

Return True since any power of the Frobenius endomorphism over an unramified p-adic field is always injective.

EXAMPLES:

```
sage: K.<a> = Qq(5^3)
sage: Frob = K.frobenius_endomorphism()
sage: Frob.is_injective()
True
```

is_surjective()

Return True since any power of the Frobenius endomorphism over an unramified p-adic field is always surjective.

EXAMPLES:

```
sage: K.<a> = Qq(5^3)
sage: Frob = K.frobenius_endomorphism()
sage: Frob.is_surjective()
True
```

order()

Return the order of this endomorphism.

EXAMPLES:

```
sage: K.<a> = Qq(5^12)
sage: Frob = K.frobenius_endomorphism()
sage: Frob.order()
12
sage: (Frob^2).order()
6
sage: (Frob^9).order()
4
```

power()

Return the smallest integer n such that this endomorphism is the n-th power of the absolute (arithmetic) Frobenius.

EXAMPLES:

```
sage: K.<a> = Qq(5^12)
sage: Frob = K.frobenius_endomorphism()
sage: Frob.power()
1
sage: (Frob^9).power()
9
sage: (Frob^13).power()
1
```

CHAPTER

TWENTYNINE

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

r

INDEX

D

E

F

G

H

I

K

L

M

Q

R

S

T

U

V

X

Z