Sage Reference Manual: p-Adics

Release 6.7

The Sage Development Team

June 24, 2015

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:

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

Secondly, as Cauchy sequences of rationals (or integers, in the case of $\mathbb{Z}_p$) 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 $\mathbb{Z}_p$, and if we allow bounded negative powers of p then we get $\mathbb{Q}_p$.

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 $\mathbb{Z}/p^n\mathbb{Z}$. As $\mathbb{Z}_p/p^n\mathbb{Z}_p \cong \mathbb{Z}/p^n\mathbb{Z}$, this is equivalent to specifying our element modulo $p^n\mathbb{Z}_p$.

The *absolute precision* of a finite approximation $\bar{x} \in \mathbb{Z}/p^n\mathbb{Z}$ to $x \in \mathbb{Z}_p$ 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 $x \in \mathbb{Q}_p$ with $x \neq 0$, we can write $x = p^v u$ where $v \in \mathbf{Z}$ and $u \in \mathbb{Z}_p^\times$. We could thus also store an element of $\mathbb{Q}_p$ (or $\mathbb{Z}_p$) 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 $\mathbb{Z}_p$ in Sage and one representation of $\mathbb{Q}_p$:

- 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 $\mathbb{Z}_p$ is basically a wrapper around $\mathbb{Z}/p^n\mathbb{Z}$, 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 $\mathbb{Z}_p$, 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 + O(5^10)
sage: b = R(105)
sage: b
5 + 4*5^2 + O(5^10)
```

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

```
sage: a + b
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)
```

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 + O(5^10)
```

```
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 + O(5^10)
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 + O(5^10)
```

1.1.2 Capped Absolute Rings

The second type of implementation of $\mathbb{Z}_p$ 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
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 $\mathbb{Z}_p$ and $\mathbb{Q}_p$ 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
Unramified Extension of 5-adic Ring with capped absolute precision 20
in c defined by (1 + O(5^20))*x^3 + (O(5^20))*x^2 + (3 + O(5^20))*x + (3 + O(5^20))
```

1.1.5 Eisenstein Extensions

It is also possible to create Eisenstein extensions of $\mathbb{Z}_p$ and $\mathbb{Q}_p$. 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)
```

You can do arithmetic in this Eisenstein extension:

```
sage: (1 + w)^7
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 $\mathbb{Z}_p$ 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=20*, *print_mode=None*, *halt=40*, *names=None*, *print_pos=None*, *print_sep=None*, *print_alphabet=None*, *print_max_terms=None*, *check=True*)

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
```

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

Bases: `sage.structure.factory.UniqueFactory`

A creation function for p-adic fields.

INPUT:

- `p` – integer: the p in $\mathbb{Q}_p$
- `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'` and `'lazy'` (though `'lazy'` currently doesn't work). See TYPES and PRECISION below
- `print_mode` – string (default: `None`). Valid modes are 'series', 'val-unit', 'terse', 'digits', and 'bars'. See PRINTING below
- `halt` – currently irrelevant (to be used for lazy fields)
- `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.
- `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)

OUTPUT:

- The corresponding 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 = 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 two types of p-adic fields: capped relative fields and lazy 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 = 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)
```

The lazy case will eventually support elements that can increase their precision upon request. It is not currently implemented.

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*
```

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)
```

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)
```

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)
```

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)
'...4230300'
sage: b = R(-70700); repr(b)
'...4444444444444440214200'
sage: c = R(-707/5); repr(c)
'...4444444444444443413.3'
sage: d = R(-707/5^2); repr(d)
'...444444444444444341.33'
```

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. Note that if the valuation of the element is very negative, more digits will be printed.:

```
sage: S = Qp(5, print_mode='digits', print_max_terms=4); b = S(-70700); repr(b)
'...214200'
sage: d = S(-707/5^2); repr(d)
'...41.33'
sage: e = S(-707/5^6); repr(e)
'...?.434133'
sage: f = S(-707/5^6,absprec=-2); repr(f)
'...?.??4133'
sage: g = S(-707/5^4); repr(g)
'...?.4133'
```

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_max_terms=4, print_alphabet=('1','2','3','4','5')); b
'...325311'
```

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 of 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_mode='bars', print_max_terms=4); b = T(-70700); repr(b)
'...2|1|4|2|0|0'
sage: d = T(-707/5^2); repr(d)
'...4|1|.|3|3'
sage: e = T(-707/5^6); repr(e)
'...|.|4|3|4|1|3|3'
sage: f = T(-707/5^6,absprec=-2); repr(f)
'...|.|?|?|4|1|3|3'
sage: g = T(-707/5^4); repr(g)
'...|.|4|1|3|3'
```

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'
```

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=20, type='capped-rel', print_mode=None, halt=40, names=None, ram_name=None, print_pos=None, print_sep=None, print_alphabet=None, print_max_terms=None, check=True*)

Creates a key from input parameters for `Qp`.

See the documentation for `Qp` for more information.

TESTS:

```
sage: Qp.create_key(5,40)
(5, 40, 'capped-rel', 'series', '5', True, '|', (), -1)
```

create_object(*version, key*)

Creates an object using a given key.

See the documentation for `Qp` for more information.

TESTS:

```
sage: Qp.create_object((3,4,2),(5, 41, 'capped-rel', 'series', '5', True, '|', (), -1))
5-adic Field with capped relative precision 41
```

`sage.rings.padics.factory.`**Qq**(*q, prec=20, type='capped-rel', modulus=None, names=None, print_mode=None, halt=40, 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, check=True*)

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

INPUT:

- `q` – integer, list, tuple or `Factorization` object. If `q` is an integer, it is the prime power q in $\mathbb{Q}_q$. 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'` and `'lazy'` (though `'lazy'` doesn't currently work). See TYPES and PRECISION below
- `modulus` – polynomial (default `None`) A polynomial defining an unramified extension of $\mathbb{Q}_p$. 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.
- `halt` – currently irrelevant (to be used for lazy fields)
- `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 reside 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.
- `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
a*5^2 + O(5^22)
sage: b.precision_relative()
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()
22
```

There are two types of unramified p-adic fields: capped relative fields and lazy 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
2 + (2*a + 2)*3 + (2*a + 1)*3^2 + O(3^5)
sage: c = R(3249); c
3^2 + 3^4 + 3^5 + 3^6 + O(3^7)
sage: b + c
2 + (2*a + 2)*3 + (2*a + 2)*3^2 + 3^4 + O(3^5)
```

The lazy case will eventually support elements that can increase their precision upon request. It is not currently implemented.

MODULUS:

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

The modulus can be given in a number of forms.

1. A **polynomial**.

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

```
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)[]
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
(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: 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
(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')
sage: X.<a> = Qq(27, modulus = x^3 + 2*x + 1); X.modulus()
(1 + O(3^20))*x^3 + (O(3^20))*x^2 + (2 + O(3^20))*x + (1 + O(3^20))
sage: X == R
True
```

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

```
sage: GF(125, 'a').modulus()
x^3 + 3*x + 3
sage: Y.<a> = Qq(125); Y.modulus()
(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 $\mathbb{F}_p[x]$ is irreducible).:

```
sage: P.<x> = ZZ[]
sage: Z.<a> = Qq(125, modulus = x^3 + 3*x^2 + x + 1); Z.modulus()
(1 + O(5^20))*x^3 + (3 + O(5^20))*x^2 + (1 + O(5^20))*x + (1 + O(5^20))
sage: Y == Z
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
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)*
sage: ~(3*a+18)
(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
```

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
-1 - b*3 - 3^2 + (b + 1)*3^3 + O(3^20)
sage: -3*(1+2*b)^4
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
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)
'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: 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 +
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 +
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
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 + ...)
sage: V.<f> = Qq(128, prec = 8, print_mode='series', print_max_unram_terms = 0); repr((1+f)^9 -
'(...)*2 + (...)*2^2 + (...)*2^3 + (...)*2^4 + (...)*2^5 + (...)*2^6 + (...)*2^7 + O(2^8)'
```

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
(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
3^3 * (15 + 64*a) + O(3^7)
sage: ~b
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); b = (1+3*a)^9 - 1; b
3^3 * (15 - 17*a) + O(3^7)
sage: ~b
3^-3 * (-40 + a) + O(3)
```

ram_name affects how the prime is printed.:

```
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*
p^-2 * (503009563508519137754940 + 704413692798200940253892*a + 968097057817740999537581*a^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); b = ~(17*(a^3-a+14));
17^-1 * (22110411 + 11317400*a + 20656972*a^2 + ...) + O(17^5)
sage: b*17*(a^3-a+14)
1 + O(17^6)
```

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
(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')
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)
```

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
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
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
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> = Qq(125, print_mode='terse', ram_name='p'); (a - 1/5)^6
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
106251/5^6 + 49994/5^5*a + ... + O(5^14)
```

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
(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
(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^
sage: R.<a> = Qq(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 elements with negative valuation are shown with a decimal point at valuation 0.:

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

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

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

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

```
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)
'...[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: T.<a> = Qq(125, print_mode='bars', print_max_ram_terms=3, print_pos=False); repr((a-5)^6)
'...[0, 0, -1]|[-2]|[-1, -2, -1]'
sage: repr(5*(a-5)^6+50)
'...[0, 0, -1]|[]|[-1, -2, -1]|[]'
```

However, if the element has negative valuation, digits are shown up to the decimal point.:

```
sage: repr((a-1/5)^6)
'...|.|[-2, -1, -1]|[2, 2, 1]|[0, 0, -2]|[0, -1]|[0, -1]|[1]'
```

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

```
sage: U.<a> = Qq(625, print_mode='bars', print_sep=''); b = (a+5)^6; repr(b)
'...[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: 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)
'...[...][...][...][...][][...][...]'
```

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
(False, 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[]
sage: K = Qq([(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.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', print_pos=False, check=True)
```

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

TESTS:

Check that trac ticket #8162 is resolved:

```
sage: R = Qq([(5,3)], names="alpha", check=False); R
Unramified Extension of 5-adic Field with capped relative precision 20 in alpha defined by (1 +
sage: Qq((5, 3), names="alpha") is R
True
sage: Qq(125.factor(), names="alpha") is R
True
```

`sage.rings.padics.factory.`**`QqCR`**(*q, prec=20, modulus=None, names=None, print_mode=None, halt=40, 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, check=True*)

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
Unramified Extension of 5-adic Field with capped relative precision 40 in a defined by (1 + O(5^
```

`sage.rings.padics.factory.`**`ZpCA`**`(`*p*, *prec=20*, *print_mode=None*, *halt=40*, *names=None*, *print_pos=None*, *print_sep=None*, *print_alphabet=None*, *print_max_terms=None*, *check=True*`)`

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=20*, *print_mode=None*, *halt=40*, *names=None*, *print_pos=None*, *print_sep=None*, *print_alphabet=None*, *print_max_terms=None*, *check=True*`)`

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.`**`ZpFM`**`(`*p*, *prec=20*, *print_mode=None*, *halt=40*, *names=None*, *print_pos=None*, *print_sep=None*, *print_alphabet=None*, *print_max_terms=None*, *check=True*`)`

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
```

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

Bases: `sage.structure.factory.UniqueFactory`

A creation function for p-adic rings.

INPUT:

- `p` – integer: the p in $\mathbb{Z}_p$
- `prec` – integer (default: `20`) the precision cap of the ring. Except for the fixed modulus case, 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'` and `'lazy'` (though lazy is not yet implemented). See TYPES and PRECISION below
- `print_mode` – string (default: `None`). Valid modes are `'series'`, `'val-unit'`, `'terse'`, `'digits'`, and `'bars'`. See PRINTING below
- `halt` – currently irrelevant (to be used for lazy fields)
- `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.
- `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)

OUTPUT:

- The corresponding p-adic ring.

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 = 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 four types of p-adic rings: capped relative rings (type= `'capped-rel'`), capped absolute rings (type= `'capped-abs'`), fixed modulus ring (type= `'fixed-mod'`) and lazy rings (type= `'lazy'`).

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 $\mathbb{Z}/p^n\mathbb{Z}$ 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 + O(5^5)
sage: a // 5
1 + 2*5^2 + 5^3 + O(5^5)
```

The lazy case will eventually support elements that can increase their precision upon request. It is not currently implemented.

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*
```

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)
```

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)
```

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)
```

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_mode='digits', print_max_terms=4); b = S(-70700); repr(b)
'...214200'
```

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_max_terms=4, print_alphabet=('1','2','3','4','5')); b
'...325311'
```

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 of 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_mode='bars', print_max_terms=4); b = T(-70700); repr(b)
'...2|1|4|2|0|0'
```

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'
```

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 100
```

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=20*, *type='capped-rel'*, *print_mode=None*, *halt=40*, *names=None*, *ram_name=None*, *print_pos=None*, *print_sep=None*, *print_alphabet=None*, *print_max_terms=None*, *check=True*)

Creates a key from input parameters for `Zp`.

See the documentation for `Zp` for more information.

TESTS:

```
sage: Zp.create_key(5,40)
(5, 40, 'capped-rel', 'series', '5', True, '|', (), -1)
sage: Zp.create_key(5,40,print_mode='digits')
(5, 40, 'capped-rel', 'digits', '5', True, '|', ('0', '1', '2', '3', '4'), -1)
```

create_object (*version*, *key*)

Creates an object using a given key.

See the documentation for `Zp` for more information.

TESTS:

```
sage: Zp.create_object((3,4,2),(5, 41, 'capped-rel', 'series', '5', True, '|', (), -1))
5-adic Ring with capped relative precision 41
```

`sage.rings.padics.factory.`**Zq**(*q*, *prec=20*, *type='capped-abs'*, *modulus=None*, *names=None*, *print_mode=None*, *halt=40*, *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*, *check=True*)

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

INPUT:

- `q` – integer, list or tuple: the prime power in $\mathbb{Q}_q$. 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-rel'` and `'lazy'` (though `'lazy'` doesn't currently work). See TYPES and PRECISION below
- modulus – polynomial (default None) A polynomial defining an unramified extension of $\mathbb{Z}_p$. 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.
- `halt` – currently irrelevant (to be used for lazy fields)
- `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 reside 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.
- `check` – bool (default `True`) whether to check inputs.

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, which gives the number of known p-adic digits:

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

There are four types of unramified p-adic rings: capped relative rings, capped absolute rings, fixed modulus rings, and lazy rings.

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
2 + (2*a + 2)*3 + (2*a + 1)*3^2 + O(3^5)
sage: c = R(3249); c
3^2 + 3^4 + 3^5 + 3^6 + O(3^7)
sage: b + c
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
2*3^-1 + (a + 1) + (a + 1)*3 + a*3^3 + O(3^4)
sage: d.parent()
Unramified Extension of 3-adic Field with capped relative precision 5 in a defined by (1 + O(3^5
```

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: 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
3^2 + 3^4 + O(3^5)
sage: b*c
2*3^3 + (2*a + 2)*3^4 + O(3^5)
sage: b*c >> 1
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: 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 + O(3^5)
sage: c = R(3249); c
3^2 + 3^4 + O(3^5)
sage: b*c
2*3^3 + (2*a + 2)*3^4 + O(3^5)
sage: b*c >> 1
2*3^2 + (2*a + 2)*3^3 + O(3^5)
```

The lazy case will eventually support elements that can increase their precision upon request. It is not currently implemented.

MODULUS:

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

The modulus can be given in a number of forms.

1.A **polynomial**.

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

```
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)[]
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
(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: 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')
sage: X.<a> = Zq(27, modulus = x^3 + 2*x + 1); X.modulus()
(1 + O(3^20))*x^3 + (O(3^20))*x^2 + (2 + O(3^20))*x + (1 + O(3^20))
```

```
sage: X == R
True
```

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

```
sage: GF(125, 'a').modulus()
x^3 + 3*x + 3
sage: Y.<a> = Zq(125); Y.modulus()
(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 $\mathbb{F}_p[x]$ is irreducible).:

```
sage: P.<x> = ZZ[]
sage: Z.<a> = Zq(125, modulus = x^3 + 3*x^2 + x + 1); Z.modulus()
(1 + O(5^20))*x^3 + (3 + O(5^20))*x^2 + (1 + O(5^20))*x + (1 + O(5^20))
sage: Y == Z
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> = 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)*
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
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
-1 - b*3 - 3^2 + (b + 1)*3^3 + O(3^20)
sage: -3*(1+2*b)^4
3 + b*3^2 + 3^3 + (-b - 1)*3^4 + O(3^20)
```

ram_name controls how the prime is printed.:

```
sage: T.<d> = Zq(9, print_mode='series', ram_name='p'); 3*(1+2*d)^4
2*p + (2*d + 2)*p^2 + (2*d + 1)*p^3 + O(p^20)
```

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)
'3 + e*3^2 + 3^3 + (2*e + 2)*3^4 + ... + O(3^20)'
```

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

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

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
(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
3^3 * (15 + 64*a) + O(3^7)
sage: ~b
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
3^3 * (15 - 17*a) + O(3^7)
sage: ~b
3^-3 * (-40 + a) + O(3)
```

ram_name affects how the prime is printed.:

```
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*
p^2 * (90732455187 + 713749771767*a + 579958835561*a^2) + O(p^4)
sage: b * (a^2 + a - 4)^-4
p^2 * 1 + O(p^4)
```

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

```
sage: U.<a> = Zq(17^4, 6, print_mode='val-unit', print_max_terse_terms=3); b = (17*(a^3-a+14)^6)
17 * (772941 + 717522*a + 870707*a^2 + ...) + O(17^6)
```

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
(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> = Zq(125, print_mode='terse')
sage: (a+5)^177
68210977979428 + 90313850704069*a + 73948093055069*a^2 + O(5^20)
sage: (a/5+1)^177
10990518995053/5^177 + 14019905391569/5^177*a + 16727634070694/5^177*a^2 + O(5^-158)
```

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: 5*a + a^2/25
5*a + 1/5^2*a^2 + O(5^16)
```

print_pos controls whether to use a balanced representation or not.:

```
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
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
106251/5^6 + 49994/5^5*a + ... + O(5^14)
```

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
(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> = 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^
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 of the precision from the representation in this mode.:

```
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> = 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: T.<a> = Zq(125, print_mode='bars', print_max_ram_terms=3, print_pos=False); repr((a-5)^6)
'...[0, 0, -1]|[-2]|[-1, -2, -1]'
sage: repr(5*(a-5)^6+50)
'...[0, 0, -1]|[]|[-1, -2, -1]|[]'
```

However, if the element has negative valuation, digits are shown up to the decimal point.:

```
sage: repr((a-1/5)^6)
'...|.|[-2, -1, -1]|[2, 2, 1]|[0, 0, -2]|[0, -1]|[0, -1]|[1]'
```

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

```
sage: U.<a> = Zq(625, print_mode='bars', print_sep=''); b = (a+5)^6; repr(b)
'...[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: 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)
'...[...][...][...][...][][...][...]'
```

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
(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: 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', print_pos=False, check=True)
```

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

TESTS:

```
sage: R = Zq([(5,3)], names="alpha"); R
Unramified Extension of 5-adic Ring with capped absolute precision 20 in alpha defined by (1 + O
sage: Zq((5, 3), names="alpha") is R
True
sage: Zq(125.factor(), names="alpha") is R
True
```

sage.rings.padics.factory.**ZqCA**(*q*, *prec=20*, *modulus=None*, *names=None*, *print_mode=None*, *halt=40*, *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*, *check=True*)

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
Unramified Extension of 5-adic Ring with capped absolute precision 40 in a defined by (1 + O(5^4
```

sage.rings.padics.factory.**ZqCR**(*q*, *prec=20*, *modulus=None*, *names=None*, *print_mode=None*, *halt=40*, *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*, *check=True*)

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
Unramified Extension of 5-adic Ring with capped relative precision 40 in a defined by (1 + O(5^4
```

sage.rings.padics.factory.**ZqFM**(*q*, *prec=20*, *modulus=None*, *names=None*, *print_mode=None*, *halt=40*, *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*, *check=True*)

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
Unramified Extension of 5-adic Ring of fixed modulus 5^40 in a defined by (1 + O(5^40))*x^2 + (4
```

sage.rings.padics.factory.**get_key_base**(*p*, *prec*, *type*, *print_mode*, *halt*, *names*, *ram_name*, *print_pos*, *print_sep*, *print_alphabet*, *print_max_terms*, *check*, *valid_non_lazy_types*)

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

It fills in unspececified values and checks for contradictions in the input. It also standardizes irrelevant options so that duplicate parents aren't created.

EXAMPLES:

```
sage: from sage.rings.padics.factory import get_key_base
sage: get_key_base(11, 5, 'capped-rel', None, 0, None, None, None, ':', None, None, True, ['capp
```

```
(11, 5, 'capped-rel', 'series', '11', True, '|', (), -1)
sage: get_key_base(12, 5, 'capped-rel', 'digits', 0, None, None, None, None, None, None, False,
(12, 5, 'capped-rel', 'digits', '12', True, '|', ('0', '1', '2', '3', '4', '5', '6', '7', '8', '
```

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

Returns 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: 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*)

Returns true iff this monic polynomial is unramified.

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

EXAMPLES:

```
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*)

Returns 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: sage.structure.factory.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)
sage: W
Eisenstein Extension of 5-adic Ring with capped relative precision 3 in w defined by (1 + O(5^3)
sage: W.precision_cap()
12
```

create_key_and_extra_args(*base*, *premodulus*, *prec=None*, *print_mode=None*, *halt=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*, *check=True*, *unram=False*)

Creates a key from input parameters for pAdicExtension.

See the documentation for `Qq` for more information.

TESTS:

```
sage: R = Zp(5,3)
sage: S.<x> = ZZ[]
sage: pAdicExtension.create_key_and_extra_args(R, x^4-15,names='w')
(('e', 5-adic Ring with capped relative precision 3, x^4 - 15, (1 + O(5^3))*x^4 + (O(5^4))*x
```

create_object(*version*, *key*, *shift_seed*)

Creates an object using a given key.

See the documentation for pAdicExtension for more information.

TESTS:

```
sage: R = Zp(5,3)
sage: S.<x> = R[]
sage: pAdicExtension.create_object(version = (3,4,2), key = ('e', R, x^4 - 15, x^4 - 15, ('w
Eisenstein Extension of 5-adic Ring with capped relative precision 3 in w defined by (1 + O(
```

`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)
sage: f = x^2+1
sage: sage.rings.padics.factory.split(f, 10)
Traceback (most recent call last):
...
NotImplementedError: Extensions by general polynomials not yet supported. Please use an unramifi
```

TESTS:

This checks that ticket #6186 is still fixed:

sage: k = Qp(13) sage: x = polygen(k) sage: f = x^2+1 sage: L.<a> = k.extension(f) 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*, *absprec*)

Truncates the unused precision off of a polynomial.

EXAMPLES:

```
sage: R = Zp(5)
sage: S.<x> = R[]
sage: from sage.rings.padics.factory import truncate_to_prec
sage: f = x^4 + (3+O(5^6))*x^3 + O(5^4)
sage: truncate_to_prec(f, 5)
(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: `sage.rings.ring.CommutativeRing`

Initializes self.

EXAMPLES:

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

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

```
sage: TestSuite(R).run()
sage: K = Qp(7)
sage: TestSuite(K).run()
```

TESTS:

```
sage: R = Zp(5, 5, 'fixed-mod')
sage: R._repr_option('element_is_atomic')
False
```

`defining_polynomial`(*var='x'*)

Returns the defining polynomial of this local ring, i.e. just `x`.

INPUT:

- `self` – a local ring
- `var` – string (default: `'x'`) the name of the variable

OUTPUT:

- polynomial – the defining polynomial of this ring as an extension over its ground ring

EXAMPLES:

```
sage: R = Zp(3, 3, 'fixed-mod'); R.defining_polynomial('foo')
(1 + O(3^3))*foo + (O(3^3))
```

`degree`()

Returns the degree of `self` over the ground ring, i.e. 1.

INPUT:

- `self` – a local ring

OUTPUT:

- integer – the degree of this ring, i.e., 1

EXAMPLES:

```
sage: R = Zp(3, 10, 'capped-rel'); R.degree()
1
```

e (*K=None*)

Returns the ramification index over the ground ring: 1 unless overridden.

INPUT:

- `self` – a local ring
- `K` – a subring of `self` (default `None`)

OUTPUT:

- integer – the ramification index of this ring: 1 unless overridden.

EXAMPLES:

```
sage: R = Zp(3, 5, 'capped-rel'); R.e()
1
```

ext (**args, **kwds*)

Constructs an extension of self. See `extension` for more details.

EXAMPLES:

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

f (*K=None*)

Returns the inertia degree over the ground ring: 1 unless overridden.

INPUT:

- `self` – a local ring
- `K` – a subring (default `None`)

OUTPUT:

- integer – the inertia degree of this ring: 1 unless overridden.

EXAMPLES:

```
sage: R = Zp(3, 5, 'capped-rel'); R.f()
1
```

ground_ring ()

Returns `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()

Returns `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(*K=None*)

Returns the inertia degree over `K` (defaults to the ground ring): 1 unless overridden.

INPUT:

- `self` – a local ring
- `K` – a subring of `self` (default None)

OUTPUT:

- integer – the inertia degree of this ring: 1 unless overridden.

EXAMPLES:

```
sage: R = Zp(3, 5, 'capped-rel'); R.inertia_degree()
1
```

inertia_subring()

Returns 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()

Returns 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()

Returns 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()

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

INPUT: self – a p-adic ring

OUTPUT: boolean – whether self is exact, i.e. False.

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

is_finite()

Returns whether this ring is finite, i.e. `False`.

INPUT:

- `self` – a p-adic ring

OUTPUT:

- boolean – whether self is finite, i.e., `False`

EXAMPLES:

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

is_fixed_mod()

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

is_lazy()

Returns whether this p-adic ring bounds precision in a lazy fashion.

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

EXAMPLES:

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

maximal_unramified_subextension()

Returns 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()

Returns the precision cap for `self`.

INPUT:

- `self` – a local ring

OUTPUT:

- integer – `self`'s precision cap

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
```

NOTES:

```
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.  For lazy rings, this gives the
initial precision to which elements are computed.
```

ramification_index(*K=None*)

Returns the ramification index over the ground ring: 1 unless overridden.

INPUT:

- `self` – a local ring

OUTPUT:

- integer – the ramification index of this ring: 1 unless overridden.

EXAMPLES:

```
sage: R = Zp(3, 5, 'capped-rel'); R.ramification_index()
1
```

residue_characteristic()

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

INPUT:

- `self` – a p-adic ring.

OUTPUT:

- integer – the characteristic of the residue field.

EXAMPLES:

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

residue_class_degree(*K=None*)

Returns the inertia degree over the ground ring: 1 unless overridden.

INPUT:

- `self` – a local ring
- `K` – a subring (default `None`)

OUTPUT:

•integer – the inertia degree of this ring: 1 unless overridden.

EXAMPLES:

```
sage: R = Zp(3, 5, 'capped-rel'); R.residue_class_degree()
1
```

uniformiser()

Returns 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[]
sage: B.<t> = A.ext(x^2+7)
sage: B.uniformiser()
t + O(t^21)
```

uniformiser_pow(n)

Returns 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

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)
```

NOTES:

```
For more documentation see localvars in parent_gens.pyx
```

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

Bases: sage.rings.ring.PrincipalIdealDomain, sage.rings.padics.local_generic.LocalGeneric

Initialization.

INPUTS:

```
- 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()

Returns the characteristic of self, which is always 0.

INPUT:

self – a p-adic parent

OUTPUT:

integer – self's characteristic, i.e., 0

EXAMPLES:

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

extension(*modulus*, *prec=None*, *names=None*, *print_mode=None*, *halt=None*, ***kwds*)

Create an extension of this p-adic ring.

EXAMPLES:

```
sage: k = Qp(5)
sage: R.<x> = k[]
sage: l.<w> = k.extension(x^2-5); l
Eisenstein Extension of 5-adic Field with capped relative precision 20 in w defined by (1 +

sage: F = list(Qp(19)['x'](cyclotomic_polynomial(5)).factor())[0][0]
sage: L = Qp(19).extension(F, names='a')
sage: L
Unramified Extension of 19-adic Field with capped relative precision 20 in a defined by (1 +
```

frobenius_endomorphism(*n=1*)

INPUT:

- n – an integer (default: 1)

OUTPUT:

The n-th power of the absolute arithmetic Frobenius endomorphism on this field.

EXAMPLES:

```
sage: K.<a> = Qq(3^5)
sage: Frob = K.frobenius_endomorphism(); Frob
Frobenius endomorphism on Unramified Extension of 3-adic Field ... lifting a |--> a^3 on the
sage: Frob(a) == a.frobenius()
True
```

We can specify a power:

```
sage: K.frobenius_endomorphism(2)
Frobenius endomorphism on Unramified Extension of 3-adic Field ... lifting a |--> a^(3^2) on
```

The result is simplified if possible:

```
sage: K.frobenius_endomorphism(6)
Frobenius endomorphism on Unramified Extension of 3-adic Field ... lifting a |--> a^3 on the
sage: K.frobenius_endomorphism(5)
Identity endomorphism of Unramified Extension of 3-adic Field ...
```

Comparisons work:

```
sage: K.frobenius_endomorphism(6) == Frob
True
```

gens()

Returns a list of generators.

EXAMPLES:

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

ngens()

Returns 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()
1
```

prime()

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

INPUT:

self – a p-adic parent

OUTPUT:

integer – the characteristic of the residue field

EXAMPLES:

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

print_mode()

Returns the current print mode as a string.

INPUT:

self – a p-adic field

OUTPUT:

string – self's print mode

EXAMPLES:

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

residue_characteristic()

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

INPUT:

self – a p-adic ring

OUTPUT:

integer – the characteristic of the residue field

EXAMPLES:

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

residue_class_field()

Returns the residue class field.

INPUT:

self – a p-adic ring

OUTPUT:

the residue field

EXAMPLES:

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

residue_field()

Returns the residue class field.

INPUT:

self – a p-adic ring

OUTPUT:

the residue field

EXAMPLES:

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

residue_system()

Returns a list of elements representing all the residue classes.

INPUT:

self – a p-adic ring

OUTPUT:

list of elements – a list of elements representing all the residue classes

EXAMPLES:

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

some_elements()

Returns a list of elements in this ring.

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

EXAMPLES:

```
sage: Zp(2).some_elements()
[0, 1 + O(2^20), 2 + O(2^21)]
```

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

Returns the teichmuller representative of x.

INPUT:

- self – a p-adic ring
- x – something that can be cast into self

OUTPUT:

- element – the teichmuller 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 + O(5^10)
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.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
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
```

AUTHORS:

- Initial version: David Roe
- Quadratic time version: Kiran Kedlaya <kedlaya@math.mit.edu> (3/27/07)

teichmuller_system()

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

INPUT:

- self – a p-adic ring

OUTPUT:

- list of elements – a list of teichmuller 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 + O(3^5), 242 + O(3^5)]
```

NOTES:

Should this return 0 as well?

uniformizer_pow(*n*)

Returns 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 + O(3^5)
sage: R.uniformizer_pow(infinity)
O(3^5)
```

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: sage.rings.padics.local_generic.LocalGeneric

Initializes self.

EXAMPLES:

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

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

```
sage: TestSuite(R).run()
sage: K = Qp(7)
sage: TestSuite(K).run()
```

TESTS:

```
sage: R = Zp(5, 5, 'fixed-mod')
sage: R._repr_option('element_is_atomic')
False
```

is_capped_absolute()

Returns 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: sage.rings.padics.generic_nodes.CappedRelativeGeneric

Initializes self.

EXAMPLES:

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

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

```
sage: TestSuite(R).run()
sage: K = Qp(7)
sage: TestSuite(K).run()
```

TESTS:

```
sage: R = Zp(5, 5, 'fixed-mod')
sage: R._repr_option('element_is_atomic')
False
```

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

Bases: sage.rings.padics.local_generic.LocalGeneric

Initializes self.

EXAMPLES:

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

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

```
sage: TestSuite(R).run()
sage: K = Qp(7)
sage: TestSuite(K).run()
```

TESTS:

```
sage: R = Zp(5, 5, 'fixed-mod')
sage: R._repr_option('element_is_atomic')
False
```

is_capped_relative()

Returns 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: sage.rings.padics.generic_nodes.CappedRelativeGeneric

Initializes self.

EXAMPLES:

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

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

```
sage: TestSuite(R).run()
sage: K = Qp(7)
sage: TestSuite(K).run()
```

TESTS:

```
sage: R = Zp(5, 5, 'fixed-mod')
sage: R._repr_option('element_is_atomic')
False
```

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

Bases: sage.rings.padics.local_generic.LocalGeneric

Initializes self.

EXAMPLES:

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

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

```
sage: TestSuite(R).run()
sage: K = Qp(7)
sage: TestSuite(K).run()
```

TESTS:

```
sage: R = Zp(5, 5, 'fixed-mod')
sage: R._repr_option('element_is_atomic')
False
```

is_fixed_mod()

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

`sage.rings.padics.generic_nodes.`**`is_pAdicField`**`(`*R*`)`

Returns `True` if and only if `R` is a p-adic field.

EXAMPLES:

```
sage: is_pAdicField(Zp(17))
False
sage: is_pAdicField(Qp(17))
True
```

`sage.rings.padics.generic_nodes.`**`is_pAdicRing`**`(`*R*`)`

Returns `True` if and only if `R` is a p-adic ring (not a field).

EXAMPLES:

```
sage: is_pAdicRing(Zp(5))
True
sage: is_pAdicRing(RR)
False
```

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

Bases: `sage.rings.padics.generic_nodes.pAdicRingGeneric`, `sage.rings.padics.generic_nodes.CappedAbsoluteGeneric`

Initialization.

INPUTS:

```
- 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
```

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

Bases: sage.rings.padics.generic_nodes.pAdicFieldGeneric, sage.rings.padics.generic_nodes.CappedRelativeFieldGeneric

Initialization.

INPUTS:

```
- 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
```

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

Bases: sage.rings.padics.generic_nodes.pAdicRingGeneric, sage.rings.padics.generic_nodes.CappedRelativeRingGeneric

Initialization.

INPUTS:

```
- 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
```

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

Bases: sage.rings.padics.padic_base_generic.pAdicBaseGeneric, sage.rings.padics.generic_nodes.pAdicFieldGeneric

Initialization

TESTS:

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

composite (*subfield1*, *subfield2*)

Returns 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 ()

Returns 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).

EXAMPLE:

```
sage: K = Qp(17, 8, print_mode='val-unit', print_sep='&')
sage: c, L = K.construction(); L
Rational Field
sage: c(L)
17-adic Field with capped relative precision 8
sage: K == c(L)
True
```

subfield (*list*)

Returns 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)

Returns 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: sage.rings.padics.padic_generic.pAdicGeneric, sage.rings.ring.Field

Initialization.

INPUTS:

```
- 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
```

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

Bases: sage.rings.padics.generic_nodes.pAdicRingGeneric, sage.rings.padics.generic_nodes.FixedModGeneric

Initialization.

INPUTS:

```
- 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
```

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

Bases: sage.rings.padics.padic_base_generic.pAdicBaseGeneric, sage.rings.padics.generic_nodes.pAdicRingGeneric

Initialization

TESTS:

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

construction()

Returns 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).

EXAMPLE:

```
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'*)

Returns a random element of self, optionally using the algorithm argument to decide how it generates the element. Algorithms currently implemented:

- default: Choose a_i, $i >= 0$, 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()
3 + 3*5 + 2*5^2 + 3*5^3 + 2*5^4 + 5^5 + O(5^6)
sage: ZpCA(5,6).random_element()
4*5^2 + 5^3 + O(5^6)
sage: ZpFM(5,6).random_element()
2 + 4*5^2 + 2*5^4 + 5^5 + O(5^6)
```

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

Bases: sage.rings.padics.padic_generic.pAdicGeneric, sage.rings.ring.EuclideanDomain

Initialization.

INPUTS:

```
- 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
```

is_field (*proof=True*)

Returns whether this ring is actually a field, ie False.

EXAMPLES:

```
sage: Zp(5).is_field()
False
```

krull_dimension ()

Returns 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 $\mathbb{Z}_p$ and $\mathbb{Q}_p$.

AUTHORS:

- David Roe

class `sage.rings.padics.padic_base_generic.`**`pAdicBaseGeneric`**(*p*, *prec*, *print_mode*, *names*, *element_class*)

Bases: `sage.rings.padics.padic_generic.pAdicGeneric`

Initialization

TESTS:

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

`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
```

`fraction_field`(*print_mode=None*)

Returns the fraction field of `self`.

INPUT:

- `print_mode` - a dictionary containing print options. Defaults to the same options as this ring.

OUTPUT:

•the fraction field of `self`.

EXAMPLES:

```
sage: R = Zp(5, print_mode='digits')
sage: K = R.fraction_field(); repr(K(1/3))[3:]
'31313131313131313132'
sage: L = R.fraction_field({'max_ram_terms':4}); repr(L(1/3))[3:]
'3132'
```

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 $\mathbb{Z}_p$ 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 $\mathbb{Z}_p$ 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
```

integer_ring (*print_mode=None*)

Returns the integer ring of `self`, possibly with `print_mode` changed.

INPUT:

•`print_mode` - a dictionary containing print options. Defaults to the same options as this ring.

OUTPUT:

•The ring of integral elements in `self`.

EXAMPLES:

```
sage: K = Qp(5, print_mode='digits')
sage: R = K.integer_ring(); repr(R(1/3))[3:]
'31313131313131313132'
```

```
sage: S = K.integer_ring({'max_ram_terms':4}); repr(S(1/3))[3:]
'3132'
```

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 $\mathbb{Z}_p$ for the same prime as `self`.

INPUT:

- `self` – a p-adic ring
- `ring` – a ring

OUTPUT:

- `boolean` – whether `ring` is an implementation of mathbb{Z}_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(
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
```

plot(*max_points=2500*, ***args*)

Creates a visualization of this p-adic ring as a fractal similar as a generalization of the the Sierpi'nski triangle. The resulting image attempts to capture the algebraic and topological characteristics of $\mathbb{Z}_p$.

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()
Graphics object consisting of 1 graphics primitive
sage: Zp(5).plot(max_points=625)
Graphics object consisting of 1 graphics primitive
sage: Zp(23).plot(rgbcolor=(1,0,0))
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 + O(3^5)
```

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.`**`pAdicExtensionGeneric`**(*poly*, *prec*, *print_mode*, *names*, *element_class*)

Bases: `sage.rings.padics.padic_generic.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
```

`defining_polynomial`()

Returns the polynomial defining this 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.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^
```

`degree`()

Returns the degree of this extension.

EXAMPLES:

```
sage: R.<a> = Zq(125); R.degree()
3
sage: R = Zp(5); S.<x> = ZZ[]; f = x^5 - 25*x^3 + 5; W.<w> = R.ext(f)
sage: W.degree()
5
```

`fraction_field`(*print_mode=None*)

Returns the fraction field of this extension, which is just the extension of base.fraction_field() determined by the same polynomial.

INPUT:

- print_mode – a dictionary containing print options. Defaults to the same options as this ring.

OUTPUT:

- the fraction field of self.

EXAMPLES:

```
sage: U.<a> = Zq(17^4, 6, print_mode='val-unit', print_max_terse_terms=3)
sage: U.fraction_field()
Unramified Extension of 17-adic Field with capped relative precision 6 in a defined by (1 +
sage: U.fraction_field({"pos":False}) == U.fraction_field()
False
```

ground_ring()

Returns the ring of which this ring is an extension.

EXAMPLE:

```
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
```

integer_ring(*print_mode=None*)

Returns the ring of integers of self, which is just the extension of base.integer_ring() determined by the same polynomial.

INPUT:

- print_mode – a dictionary containing print options. Defaults to the same options as this ring.

OUTPUT:

- the ring of elements of self with nonnegative valuation.

EXAMPLES:

```
sage: U.<a> = Qq(17^4, 6, print_mode='val-unit', print_max_terse_terms=3)
sage: U.integer_ring()
Unramified Extension of 17-adic Ring with capped relative precision 6 in a defined by (1 + O
sage: U.fraction_field({"pos":False}) == U.fraction_field()
False
```

modulus()

Returns the polynomial defining this 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.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^
```

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()

Returns 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); R.random_element()
3*a + (2*a + 1)*5 + (4*a^2 + 3*a + 4)*5^2 + (a^2 + 2*a)*5^3 + (a + 2)*5^4 + O(5^5)
sage: R = Zp(5,3); S.<x> = ZZ[]; f = x^5 + 25*x^2 - 5; W.<w> = R.ext(f)
sage: W.random_element()
3 + 4*w + 3*w^2 + w^3 + 4*w^4 + w^5 + w^6 + 3*w^7 + w^8 + 2*w^10 + 4*w^11 + w^12 + 2*w^13 +
```

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: sage.rings.padics.padic_extension_generic.pAdicExtensionGeneric

Initializes self.

EXAMPLES:

```
sage: A = Zp(7,10)
sage: S.<x> = A[]
sage: B.<t> = A.ext(x^2+7) #indirect doctest
```

gen(*n=0*)

Returns a generator for self as an extension of its ground ring.

EXAMPLES:

```
sage: A = Zp(7,10)
sage: S.<x> = A[]
sage: B.<t> = A.ext(x^2+7)
sage: B.gen()
t + O(t^21)
```

inertia_degree(*K=None*)

Returns the inertia degree of self over K, or the ground ring if K is None.

The inertia degree is the degree of the extension of residue fields induced by this extensions. Since Eisenstein extensions are totally ramified, this will be 1 for K=None.

INPUTS:

- self – an Eisenstein extension
- K – a subring of self (default None -> self.ground_ring())

OUTPUTS:

- The degree of the induced extensions of residue fields.

EXAMPLES:

```
sage: A = Zp(7,10)
sage: S.<x> = A[]
sage: B.<t> = A.ext(x^2+7)
sage: B.inertia_degree()
1
```

inertia_subring()

Returns 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[]
sage: B.<t> = A.ext(x^2+7)
sage: B.inertia_subring()
7-adic Ring with capped relative precision 10
```

ramification_index(*K=None*)

Returns the ramification index of self over K, or over the ground ring if K is None.

The ramification index is the index of the image of the valuation map on K in the image of the valuation map on self (both normalized so that the valuation of p is 1).

INPUTS:

- self – an Eisenstein extension
- K – a subring of self (default None -> self.ground_ring())

OUTPUTS:

- The ramification index of the extension self/K

EXAMPLES:

```
sage: A = Zp(7,10)
sage: S.<x> = A[]
sage: B.<t> = A.ext(x^2+7)
sage: B.ramification_index()
2
```

residue_class_field()

Returns the residue class field.

INPUT:

- self – a p-adic ring

OUTPUT:

- the residue field

EXAMPLES:

```
sage: A = Zp(7,10)
sage: S.<x> = A[]
sage: B.<t> = A.ext(x^2+7)
sage: B.residue_class_field()
Finite Field of size 7
```

uniformizer()

Returns the uniformizer of self, ie a generator for the unique maximal ideal.

EXAMPLES:

```
sage: A = Zp(7,10)
sage: S.<x> = A[]
sage: B.<t> = A.ext(x^2+7)
sage: B.uniformizer()
t + O(t^21)
```

uniformizer_pow(*n*)

Returns the nth power of the uniformizer of self (as an element of self).

EXAMPLES:

```
sage: A = Zp(7,10)
sage: S.<x> = A[]
sage: B.<t> = A.ext(x^2+7)
sage: B.uniformizer_pow(5)
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: `sage.rings.padics.padic_extension_generic.pAdicExtensionGeneric`

An unramified extension of Qp or Zp.

`discriminant`(*K=None*)

Returns the discriminant of self over the subring K.

INPUTS:

```
- K -- a subring/subfield (defaults to the base ring).
```

EXAMPLES:

```
sage: R.<a> = Zq(125)
sage: R.discriminant()
Traceback (most recent call last):
...
NotImplementedError
```

`gen`(*n=0*)

Returns 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()
a + O(5^20)
```

`has_pth_root`()

Returns 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()
True
sage: R.<a> = Zq(17^5); R.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.<a> = Zq(37^8)
sage: R.has_root_of_unity(144)
True
sage: R.has_root_of_unity(89)
True
sage: R.has_root_of_unity(11)
False
```

inertia_degree(*K=None*)

Returns the inertia degree of self over the subring K.

INPUTS:

```
- K -- a subring (or subfield) of self.  Defaults to the
  base.
```

EXAMPLES:

```
sage: R.<a> = Zq(125); R.inertia_degree()
3
```

is_galois(*K=None*)

Returns True if this extension is Galois.

Every unramified extension is Galois.

INPUTS:

```
- K -- a subring/subfield (defaults to the base ring).
```

EXAMPLES:

```
sage: R.<a> = Zq(125); R.is_galois()
True
```

ramification_index(*K=None*)

Returns the ramification index of self over the subring K.

INPUTS:

```
- K -- a subring (or subfield) of self.  Defaults to the
  base.
```

EXAMPLES:

```
sage: R.<a> = Zq(125); R.ramification_index()
1
```

residue_class_field()

Returns the residue class field.

EXAMPLES:

```
sage: R.<a> = Zq(125); R.residue_class_field()
Finite Field in a0 of size 5^3
```

uniformizer()

Returns 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)
sage: R.uniformizer()
5 + O(5^21)
```

uniformizer_pow(*n*)

Returns the nth power of the uniformizer of self (as an element of self).

EXAMPLES:

```
sage: R.<a> = ZqCR(125)
sage: R.uniformizer_pow(5)
5^5 + O(5^25)
```

CHAPTER

TEN

P-ADIC BASE LEAVES

Implementations of $\mathbb{Z}_p$ and $\mathbb{Q}_p$

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 functions `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)
<type '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)
<type '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 $\mathbb{Z}/p^n\mathbb{Z}$ 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 + O(5^5)
sage: a // 5
1 + 2*5^2 + 5^3 + O(5^5)
```

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 $\mathbb{Z}_p$ and $\mathbb{Q}_p$ 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
```

```
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 $\mathbb{Z}/p^n\mathbb{Z}$ 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 + O(5^5)
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.

TESTS:

```
sage: R = Qp(5, 15, print_mode='bars', print_sep='&')
sage: repr(R(2777))[3:]
'4&2&1&0&2'
sage: TestSuite(R).run()

sage: R = Zp(5, 15, print_mode='bars', print_sep='&')
sage: repr(R(2777))[3:]
'4&2&1&0&2'
sage: TestSuite(R).run()

sage: R = ZpCA(5, 15, print_mode='bars', print_sep='&')
sage: repr(R(2777))[3:]
'4&2&1&0&2'
sage: TestSuite(R).run()
```

class `sage.rings.padics.padic_base_leaves.`**`pAdicFieldCappedRelative`**(*p*, *prec*, *print_mode*, *names*)

Bases: `sage.rings.padics.generic_nodes.pAdicFieldBaseGeneric`, `sage.rings.padics.generic_nodes.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'*)

Returns 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()
15*17^-8 + 10*17^-7 + 3*17^-6 + 2*17^-5 + 11*17^-4 + 6*17^-3 + O(17^-2)
```

class `sage.rings.padics.padic_base_leaves.`**pAdicRingCappedAbsolute** (*p*, *prec*, *print_mode*, *names*)

Bases: `sage.rings.padics.generic_nodes.pAdicRingBaseGeneric`, `sage.rings.padics.generic_nodes.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: `sage.rings.padics.generic_nodes.pAdicRingBaseGeneric`, `sage.rings.padics.generic_nodes.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: `sage.rings.padics.generic_nodes.pAdicRingBaseGeneric`, `sage.rings.padics.generic_nodes.pAdicFixedModRingGeneric`

An implementation of the p-adic integers using fixed modulus.

fraction_field (*print_mode=None*)

Would normally return $\mathbb{Q}_p$, but there is no implementation of $\mathbb{Q}_p$ matching this ring so this raises an error

If you want to be able to divide with elements of a fixed modulus p-adic ring, you must cast explicitly.

EXAMPLES:

```
sage: ZpFM(5).fraction_field()
Traceback (most recent call last):
...
TypeError: This implementation of the p-adic ring does not support fields of fractions.

sage: a = ZpFM(5)(4); b = ZpFM(5)(5)
```

CHAPTER ELEVEN

P-ADIC EXTENSION LEAVES

The final classes for extensions of Zp and Qp (ie classes that are not just designed to be inherited from).

AUTHORS:

- David Roe

class sage.rings.padics.padic_extension_leaves.**EisensteinExtensionFieldCappedRelative**(*prepoly, poly, prec, halt, print_mode, shift_seed, names*)

Bases: sage.rings.padics.eisenstein_extension_generic.EisensteinExtensionGeneric, sage.rings.padics.generic_nodes.pAdicCappedRelativeFieldGeneric

TESTS:

```
sage: R = Qp(3, 10000, print_pos=False); S.<x> = ZZ[]; f = x^3 + 9*x - 3
sage: W.<w> = R.ext(f); W == loads(dumps(W))
True
```

class sage.rings.padics.padic_extension_leaves.**EisensteinExtensionRingCappedAbsolute**(*prepoly, poly, prec, halt, print_mode, shift_seed, names*)

Bases: sage.rings.padics.eisenstein_extension_generic.EisensteinExtensionGeneric, sage.rings.padics.generic_nodes.pAdicCappedAbsoluteRingGeneric

TESTS:

```
sage: R = ZpCA(3, 10000, print_pos=False); S.<x> = ZZ[]; f = x^3 + 9*x - 3
sage: W.<w> = R.ext(f); W == loads(dumps(W))
True
```

class sage.rings.padics.padic_extension_leaves.**EisensteinExtensionRingCappedRelative**(*prepoly, poly, prec, halt, print_mode, shift_seed, names*)

Bases: sage.rings.padics.eisenstein_extension_generic.EisensteinExtensionGeneric, sage.rings.padics.generic_nodes.pAdicCappedRelativeRingGeneric

TESTS:

```
sage: R = Zp(3, 10000, print_pos=False); S.<x> = ZZ[]; f = x^3 + 9*x - 3
sage: W.<w> = R.ext(f); W == loads(dumps(W))
True
```

class sage.rings.padics.padic_extension_leaves.**EisensteinExtensionRingFixedMod**(*prepoly*, *poly*, *prec*, *halt*, *print_mode*, *shift_seed*, *names*)

Bases: sage.rings.padics.eisenstein_extension_generic.EisensteinExtensionGeneric, sage.rings.padics.generic_nodes.pAdicFixedModRingGeneric

TESTS:

```
sage: R = ZpFM(3, 10000, print_pos=False); S.<x> = ZZ[]; f = x^3 + 9*x - 3
sage: W.<w> = R.ext(f); W == loads(dumps(W))
True
```

class sage.rings.padics.padic_extension_leaves.**UnramifiedExtensionFieldCappedRelative**(*prepoly*, *poly*, *prec*, *halt*, *print_mod*, *shift_seed*, *names*)

Bases: sage.rings.padics.unramified_extension_generic.UnramifiedExtensionGeneric, sage.rings.padics.generic_nodes.pAdicCappedRelativeFieldGeneric

TESTS:

```
sage: R.<a> = QqCR(27,10000); R == loads(dumps(R))
True
```

class sage.rings.padics.padic_extension_leaves.**UnramifiedExtensionRingCappedAbsolute**(*prepoly*, *poly*, *prec*, *halt*, *print_mode*, *shift_seed*, *names*)

Bases: sage.rings.padics.unramified_extension_generic.UnramifiedExtensionGeneric, sage.rings.padics.generic_nodes.pAdicCappedAbsoluteRingGeneric

TESTS:

```
sage: R.<a> = ZqCA(27,10000); R == loads(dumps(R))
True
```

class sage.rings.padics.padic_extension_leaves.**UnramifiedExtensionRingCappedRelative**(*prepoly*, *poly*, *prec*, *halt*, *print_mode*, *shift_seed*, *names*)

Bases: `sage.rings.padics.unramified_extension_generic.UnramifiedExtensionGeneric`, `sage.rings.padics.generic_nodes.pAdicCappedRelativeRingGeneric`

TESTS:

```
sage: R.<a> = ZqCR(27,10000); R == loads(dumps(R))
True
```

class `sage.rings.padics.padic_extension_leaves.`**`UnramifiedExtensionRingFixedMod`**(*prepoly*, *poly*, *prec*, *halt*, *print_mode*, *shift_seed*, *names*)

Bases: `sage.rings.padics.unramified_extension_generic.UnramifiedExtensionGeneric`, `sage.rings.padics.generic_nodes.pAdicFixedModRingGeneric`

TESTS:

```
sage: R.<a> = ZqFM(27,10000); R == loads(dumps(R))
True
```

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): added inverse_of_unit()

class `sage.rings.padics.local_generic_element.`**`LocalGenericElement`**

Bases: `sage.structure.element.CommutativeRingElement`

`add_bigoh`(*prec*)

Returns self to reduced precision `prec`.

EXAMPLES:: sage: K = Qp(11, 5) sage: L.<a> = K.extension(x^20 - 11) sage: b = a^3 + 3*a^5; b a^3 + 3*a^5 + O(a^103) sage: b.add_bigoh(17) a^3 + 3*a^5 + O(a^17) sage: b.add_bigoh(150) a^3 + 3*a^5 + O(a^103)

`euclidean_degree`()

Return the degree of this element as an element of a 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)
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 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
```

TESTS:

Test that this works for all kinds of p-adic base elements:

```
sage: ZpCA(3,5)(2).inverse_of_unit()
2 + 3 + 3^2 + 3^3 + 3^4 + O(3^5)
sage: ZpCR(3,5)(2).inverse_of_unit()
2 + 3 + 3^2 + 3^3 + 3^4 + O(3^5)
sage: ZpFM(3,5)(2).inverse_of_unit()
2 + 3 + 3^2 + 3^3 + 3^4 + O(3^5)
sage: QpCR(3,5)(2).inverse_of_unit()
2 + 3 + 3^2 + 3^3 + 3^4 + O(3^5)
```

Over unramified extensions:

```
sage: R = ZpCA(3,5); S.<t> = R[]; W.<t> = R.extension( t^2 + 1 )
sage: t.inverse_of_unit()
2*t + 2*t*3 + 2*t*3^2 + 2*t*3^3 + 2*t*3^4 + O(3^5)

sage: R = ZpCR(3,5); S.<t> = R[]; W.<t> = R.extension( t^2 + 1 )
sage: t.inverse_of_unit()
2*t + 2*t*3 + 2*t*3^2 + 2*t*3^3 + 2*t*3^4 + O(3^5)

sage: R = ZpFM(3,5); S.<t> = R[]; W.<t> = R.extension( t^2 + 1 )
sage: t.inverse_of_unit()
2*t + 2*t*3 + 2*t*3^2 + 2*t*3^3 + 2*t*3^4 + O(3^5)

sage: R = QpCR(3,5); S.<t> = R[]; W.<t> = R.extension( t^2 + 1 )
sage: t.inverse_of_unit()
2*t + 2*t*3 + 2*t*3^2 + 2*t*3^3 + 2*t*3^4 + O(3^5)
```

Over Eisenstein extensions:

```
sage: R = ZpCA(3,5); S.<t> = R[]; W.<t> = R.extension( t^2 - 3 )
sage: (t - 1).inverse_of_unit()
2 + 2*t + t^2 + t^3 + t^4 + t^5 + t^6 + t^7 + O(t^8)

sage: R = ZpCR(3,5); S.<t> = R[]; W.<t> = R.extension( t^2 - 3 )
sage: (t - 1).inverse_of_unit()
2 + 2*t + t^2 + t^3 + t^4 + t^5 + t^6 + t^7 + t^8 + t^9 + O(t^10)

sage: R = ZpFM(3,5); S.<t> = R[]; W.<t> = R.extension( t^2 - 3 )
sage: (t - 1).inverse_of_unit()
2 + 2*t + t^2 + t^3 + t^4 + t^5 + t^6 + t^7 + t^8 + t^9 + O(t^10)

sage: R = QpCR(3,5); S.<t> = R[]; W.<t> = R.extension( t^2 - 3 )
sage: (t - 1).inverse_of_unit()
2 + 2*t + t^2 + t^3 + t^4 + t^5 + t^6 + t^7 + t^8 + t^9 + O(t^10)
```

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
```

TESTS:

```
sage: R(4).is_padic_unit()
True
sage: R(6).is_padic_unit()
False
sage: R(9).is_padic_unit()
False
sage: K(0).is_padic_unit()
False
sage: K(1).is_padic_unit()
True
sage: K(2).is_padic_unit()
True
sage: K(3).is_padic_unit()
False
sage: K(4).is_padic_unit()
True
sage: K(6).is_padic_unit()
False
sage: K(9).is_padic_unit()
False
sage: K(1/3).is_padic_unit()
False
sage: K(1/9).is_padic_unit()
False
sage: Qq(3^2,5,names='a')(3).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

NOTES:

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 va
True
sage: Qp(5,5)(5).is_padic_unit()
False
```

TESTS:

```
sage: R(4).is_unit()
True
sage: R(6).is_unit()
False
sage: R(9).is_unit()
False
sage: K(0).is_unit()
False
sage: K(1).is_unit()
True
sage: K(2).is_unit()
True
sage: K(3).is_unit()
True
sage: K(4).is_unit()
True
sage: K(6).is_unit()
True
sage: K(9).is_unit()
True
sage: K(1/3).is_unit()
True
sage: K(1/9).is_unit()
True
sage: Qq(3^2,5,names='a')(3).is_unit()
True
sage: R(0,0).is_unit()
False
sage: K(0,0).is_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[]
sage: F.<z> = Q7.ext(x^3+7*x+7)
sage: z.normalized_valuation()
1/3
```

quo_rem(*other*)

Return the quotient with remainder of the division of this element by `other`.

INPUT:

- `other` – an element in the same ring

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))
(0, 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)
```

slice(*i*, *j*, *k=1*)

Returns the sum of the $p^{i+l\cdot k}$ terms of the series expansion of this element, 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 can not 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]
5^-2 + 5 + O(5^3)
```

TESTS:

Test that slices also work over fields:

```
sage: a = K(1/25); a
5^-2 + O(5^4)
sage: b = K(25); b
5^2 + O(5^8)

sage: a.slice(2, 4)
O(5^4)
sage: b.slice(2, 4)
5^2 + O(5^4)
sage: a.slice(-3, -1)
5^-2 + O(5^-1)
sage: b.slice(-1, 1)
O(5)
sage: b.slice(-3, -1)
O(5^-1)
sage: b.slice(101, 100)
O(5^8)
sage: b.slice(0,7,2)
5^2 + O(5^7)
sage: b.slice(0,8,2)
5^2 + O(5^8)
sage: b.slice(0,9,2)
5^2 + O(5^8)
```

Verify that trac ticket #14106 has been fixed:

```
sage: R = Zp(5,7)
sage: a = R(300)
sage: a
```

```
2*5^2 + 2*5^3 + O(5^9)
sage: a[:5]
2*5^2 + 2*5^3 + O(5^5)
sage: a.slice(None, 5, None)
2*5^2 + 2*5^3 + O(5^5)
```

sqrt (*extend=True*, *all=False*)

TODO: document what "extend" and "all" do

INPUT:

- `self` – a local ring element

OUTPUT:

- local ring element – the square root of `self`

EXAMPLES:

```
sage: R = Zp(13, 10, 'capped-rel', 'series')
sage: a = sqrt(R(-1)); a * a
12 + 12*13 + 12*13^2 + 12*13^3 + 12*13^4 + 12*13^5 + 12*13^6 + 12*13^7 + 12*13^8 + 12*13^9 +
sage: sqrt(R(4))
2 + O(13^10)
sage: sqrt(R(4/9)) * 3
2 + O(13^10)
```

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

class `sage.rings.padics.padic_generic_element.`**`pAdicGenericElement`**

Bases: `sage.rings.padics.local_generic_element.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)
0.200000000000000
sage: Qp(7)(0).abs()
0
sage: Qp(7)(0).abs(prec=20)
0.00000
```

An unramified extension:

```
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: 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*)

Returns the additive order of self, where self is considered to be zero if it is zero modulo p^{prec}.

INPUT:

- `self` – a p-adic element
- `prec` – an integer

OUTPUT:

integer – the additive order of self

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 `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
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
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 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
sage: a.algebraic_dependency(1)
19*x - 7
sage: K2 = Qp(7,20,'capped-rel')
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
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
```

dwork_expansion(*bd=20*)

Return the value of a function defined by Dwork.

Used to compute the p-adic Gamma function, see `gamma()`.

INPUT:

- `bd` – integer. Is a bound for precision, defaults to 20

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 [RV] section 6.2.

REFERENCES:

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*)

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: If self has a `lift` method (which should happen for elements of $\mathbf{Q}_p$ and $\mathbf{Z}_p$), then one uses the rule: $\exp(x) = \exp(p)^{x/p}$ modulo the precision. The value of $\exp(p)$ is precomputed. Otherwise, use the power series expansion of $\exp$, evaluating a certain number of terms which does about $O(\text{prec})$ multiplications.

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);
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);
```

```
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: 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: 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)
```

TESTS:

Check that results are consistent over a range of precision:

```
sage: max_prec = 40
sage: p = 3
sage: K = Zp(p, max_prec)
sage: full_exp = (K(p)).exp()
sage: for prec in range(2, max_prec):
...       ll = (K(p).add_bigoh(prec)).exp()
...       assert ll == full_exp
...       assert ll.precision_absolute() == prec
sage: K = Qp(p, max_prec)
sage: full_exp = (K(p)).exp()
sage: for prec in range(2, max_prec):
...       ll = (K(p).add_bigoh(prec)).exp()
...       assert ll == full_exp
...       assert ll.precision_absolute() == prec
```

Check that this also works for capped-absolute implementations:

```
sage: Z13 = ZpCA(13, 10)
sage: a = Z13(14); a
1 + 13 + O(13^10)
sage: a.log().exp()
1 + 13 + O(13^10)

sage: R = ZpCA(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^16)
sage: z.log().exp()
```

```
1 + w^2 + 4*w^7 + O(w^16)
```

Check that this also works for fixed-mod implementations:

```
sage: Z13 = ZpFM(13, 10)
sage: a = Z13(14); a
1 + 13 + O(13^10)
sage: a.log().exp()
1 + 13 + O(13^10)

sage: R = ZpFM(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)
```

Some corner cases:

```
sage: Z2 = Zp(2, 5)
sage: Z2(2).exp()
Traceback (most recent call last):
...
ValueError: Exponential does not converge for that input.

sage: S.<x> = Z2[]
sage: W.<w> = Z2.ext(x^3-2)
sage: (w^2).exp()
Traceback (most recent call last):
...
ValueError: Exponential does not converge for that input.
sage: (w^3).exp()
Traceback (most recent call last):
...
ValueError: Exponential does not converge for that input.
sage: (w^4).exp()
1 + w^4 + w^5 + w^7 + w^9 + w^10 + w^14 + O(w^15)
```

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

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. set to `'pari'` by default, 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 [RV] 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)
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)
```

TESTS:

The implementation also works over extensions:

```
sage: K = Qp(3)
sage: R.<a> = K[]
sage: L.<a> = K.extension(a^3-3)
sage: (a+3).gcd(3)
1 + O(a^60)

sage: R = Zp(3)
sage: S.<a> = R[]
sage: S.<a> = R.extension(a^3-3)
sage: (a+3).gcd(3)
a + O(a^61)

sage: K = Qp(3)
sage: R.<a> = K[]
sage: L.<a> = K.extension(a^2-2)
sage: (a+3).gcd(3)
1 + O(3^20)

sage: R = Zp(3)
sage: S.<a> = R[]
sage: S.<a> = R.extension(a^2-2)
sage: (a+3).gcd(3)
1 + O(3^20)
```

For elements with a fixed modulus:

```
sage: R = ZpFM(3)
sage: R(3).gcd(9)
3 + O(3^20)
```

And elements with a capped absolute precision:

```
sage: R = ZpCA(3)
sage: R(3).gcd(9)
3 + O(3^20)
```

is_square()

Returns whether self is a square

INPUT:

- `self` – a p-adic element

OUTPUT:

boolean – whether self is a square

EXAMPLES:

```
sage: R = Zp(3,20,'capped-rel')
sage: R(0).is_square()
True
sage: R(1).is_square()
True
```

```
sage: R(2).is_square()
False
```

TESTS:

```
sage: R(3).is_square()
False
sage: R(4).is_square()
True
sage: R(6).is_square()
False
sage: R(9).is_square()
True

sage: R2 = Zp(2,20,'capped-rel')
sage: R2(0).is_square()
True
sage: R2(1).is_square()
True
sage: R2(2).is_square()
False
sage: R2(3).is_square()
False
sage: R2(4).is_square()
True
sage: R2(5).is_square()
False
sage: R2(6).is_square()
False
sage: R2(7).is_square()
False
sage: R2(8).is_square()
False
sage: R2(9).is_square()
True

sage: K = Qp(3,20,'capped-rel')
sage: K(0).is_square()
True
sage: K(1).is_square()
True
sage: K(2).is_square()
False
sage: K(3).is_square()
False
sage: K(4).is_square()
True
sage: K(6).is_square()
False
sage: K(9).is_square()
True
sage: K(1/3).is_square()
False
sage: K(1/9).is_square()
True

sage: K2 = Qp(2,20,'capped-rel')
sage: K2(0).is_square()
True
```

```
sage: K2(1).is_square()
True
sage: K2(2).is_square()
False
sage: K2(3).is_square()
False
sage: K2(4).is_square()
True
sage: K2(5).is_square()
False
sage: K2(6).is_square()
False
sage: K2(7).is_square()
False
sage: K2(8).is_square()
False
sage: K2(9).is_square()
True
sage: K2(1/2).is_square()
False
sage: K2(1/4).is_square()
True
```

log(*p_branch=None*, *pi_branch=None*, *aprec=None*, *change_frac=False*)

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.

INPUTS:

- `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.

NOTES:

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 (as of version 2.19-2)

Todo

There is a soft-linear time algorith for logarithm described by Dan Berstein at http://cr.yp.to/lineartime/multapps-20041007.pdf

ALGORITHM:

1.Take the unit part u of the input.

2. Raise u to $q-1$ where q is the inertia degree of the ring extension, to obtain a 1-unit.

3.Use the series expansion

$$\log(1-x) = -x - 1/2x^2 - 1/3x^3 - 1/4x^4 - 1/5x^5 - \cdots$$

to compute the logarithm $\log(u)$.

4.Divide the result by `q-1` and multiply by `self.valuation()*log(pi)`

EXAMPLES:

```
sage: Z13 = Zp(13, 10)
sage: a = Z13(14); a
1 + 13 + O(13^10)
```

Note that the relative precision decreases when we take log – it is the absolute precision that is preserved:

```
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)
```

The next few examples illustrate precision when computing p-adic logarithms:

```
sage: R = Zp(5,10)
sage: e = R(389); e
4 + 2*5 + 3*5^3 + O(5^10)
sage: e.log()
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: K = Qp(5,10)
sage: e = K(389); e
4 + 2*5 + 3*5^3 + O(5^10)
sage: e.log()
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)
```

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)
sage: z = 1 + w^2 + 4*w^7; z
1 + w^2 + 4*w^7 + O(w^20)
sage: z.log()
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
```

In an extension, there will usually be a difference between specifying `p_branch` and `pi_branch`:

```
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)
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
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
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
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 +
```

Check that log is multiplicative:

```
sage: y.log(p_branch=0) + z.log() - (y*z).log(p_branch=0)
O(w^20)
```

Now an unramified example:

```
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: 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)
sage: w.log(p_branch=2)
Traceback (most recent call last):
...
ValueError: logarithm is not integral, use change_frac=True to obtain a result in the fracti
sage: w.log(p_branch=2, change_frac=True)
2*w^-3 + O(w^21)
```

TESTS:

Check that results are consistent over a range of precision:

```
sage: max_prec = 40
sage: p = 3
sage: K = Zp(p, max_prec)
sage: full_log = (K(1 + p)).log()
sage: for prec in range(2, max_prec):
...       ll1 = (K(1+p).add_bigoh(prec)).log()
...       ll2 = K(1+p).log(prec)
...       assert ll1 == full_log
...       assert ll2 == full_log
...       assert ll1.precision_absolute() == prec
```

Check that `aprec` works for fixed-mod elements:

```
sage: R = ZpFM(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(7^10)
sage: x.log(aprec = 7)
7 + 3*7^2 + 4*7^3 + 3*7^4 + 7^5 + 3*7^6 + O(7^10)
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)
```

Check that precision is computed correctly in highly ramified extensions:

```
sage: S.<x> = ZZ[]
sage: K = Qp(5,5)
sage: f = x^625 - 5*x - 5
sage: W.<w> = K.extension(f)
sage: z = 1 - w^2 + O(w^11)
sage: x = 1 - z
sage: z.log().precision_absolute()
-975
sage: (x^5/5).precision_absolute()
-570
sage: (x^25/25).precision_absolute()
-975
sage: (x^125/125).precision_absolute()
-775

sage: z = 1 - w + O(w^2)
sage: x = 1 - z
sage: z.log().precision_absolute()
-1625
sage: (x^5/5).precision_absolute()
-615
sage: (x^25/25).precision_absolute()
-1200
sage: (x^125/125).precision_absolute()
-1625
sage: (x^625/625).precision_absolute()
-1250

sage: z.log().precision_relative()
250
```

AUTHORS:

- William Stein: initial version
- David Harvey (2006-09-13): corrected subtle precision bug (need to take denominators into account! – see trac ticket #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 `_shifted_log()`. This should really fix the issue with divisions.
- Julian Rueth (2013-02-14): Added doctests, some changes for capped-absolute implementations.

minimal_polynomial(*name*)

Returns a minimal polynomial of this p-adic element, i.e., `x - self`

INPUT:

- `self` – a p-adic element
- `name` – string: the name of the variable

EXAMPLES:

```
sage: Zp(5,5)(1/3).minimal_polynomial('x')
(1 + O(5^5))*x + (3 + 5 + 3*5^2 + 5^3 + 3*5^4 + O(5^5))
```

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(3).multiplicative_order(20)
+Infinity
sage: K(4).multiplicative_order(20)
+Infinity
sage: K(5).multiplicative_order(20)
+Infinity
sage: K(25).multiplicative_order(20)
+Infinity
sage: K(1/5).multiplicative_order(20)
+Infinity
sage: K(1/25).multiplicative_order(20)
+Infinity
sage: K.zeta().multiplicative_order(20)
4

sage: R = Zp(5,20,'capped-rel')
sage: R(-1).multiplicative_order(20)
2
sage: R(1).multiplicative_order(20)
1
sage: R(2).multiplicative_order(20)
+Infinity
sage: R(3).multiplicative_order(20)
+Infinity
sage: R(4).multiplicative_order(20)
+Infinity
sage: R(5).multiplicative_order(20)
```

```
+Infinity
sage: R(25).multiplicative_order(20)
+Infinity
sage: R.zeta().multiplicative_order(20)
4
```

norm(*ground=None*)

Returns the norm of this p-adic element over the ground ring.

> **Warning:** 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.

INPUT:

- `ground` – a subring of the parent (default: base ring)

EXAMPLES:

```
sage: Zp(5)(5).norm()
5 + O(5^21)
```

ordp(*p=None*)

Returns 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 integer and rational.

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
```

rational_reconstruction()

Returns a rational approximation to this p-adic number

INPUT:

- `self` – a p-adic element

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*)

Returns the square root of this p-adic number

INPUT:

- `self` – a p-adic element
- `extend` – bool (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` – bool (default: False); if True, return a list of all square roots

OUTPUT:

p-adic element – the square root of this p-adic number

If `all=False`, the square root chosen is the one whose reduction mod p is in the range $[0, p/2)$.

EXAMPLES:

```
sage: R = Zp(3,20,'capped-rel', 'val-unit')
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() == R(-2)
True
sage: R(9).square_root()
3 * 1 + O(3^21)
```

When p = 2, the precision of the square root is one less than the input:

```
sage: R2 = Zp(2,20,'capped-rel')
sage: R2(0).square_root()
0
sage: R2(1).square_root()
1 + O(2^19)
sage: R2(4).square_root()
2 + O(2^20)

sage: R2(9).square_root() == R2(3, 19) or R2(9).square_root() == R2(-3, 19)
True

sage: R2(17).square_root()
1 + 2^3 + 2^5 + 2^6 + 2^7 + 2^9 + 2^10 + 2^13 + 2^16 + 2^17 + O(2^19)
```

```
sage: R3 = Zp(5,20,'capped-rel')
sage: R3(0).square_root()
0
sage: R3(1).square_root()
1 + O(5^20)
sage: R3(-1).square_root() == R3.teichmuller(2) or R3(-1).square_root() == R3.teichmuller(3)
True
```

TESTS:

```
sage: R = Qp(3,20,'capped-rel')
sage: R(0).square_root()
0
sage: R(1).square_root()
1 + O(3^20)
sage: R(4).square_root() == R(-2)
True
sage: R(9).square_root()
3 + O(3^21)
sage: R(1/9).square_root()
3^-1 + O(3^19)

sage: R2 = Qp(2,20,'capped-rel')
sage: R2(0).square_root()
0
sage: R2(1).square_root()
1 + O(2^19)
sage: R2(4).square_root()
2 + O(2^20)
sage: R2(9).square_root() == R2(3,19) or R2(9).square_root() == R2(-3,19)
True
sage: R2(17).square_root()
1 + 2^3 + 2^5 + 2^6 + 2^7 + 2^9 + 2^10 + 2^13 + 2^16 + 2^17 + O(2^19)

sage: R3 = Qp(5,20,'capped-rel')
sage: R3(0).square_root()
0
sage: R3(1).square_root()
1 + O(5^20)
sage: R3(-1).square_root() == R3.teichmuller(2) or R3(-1).square_root() == R3.teichmuller(3)
True

sage: R = Zp(3,20,'capped-abs')
sage: R(1).square_root()
1 + O(3^20)
sage: R(4).square_root() == R(-2)
True
sage: R(9).square_root()
3 + O(3^19)
sage: R2 = Zp(2,20,'capped-abs')
sage: R2(1).square_root()
1 + O(2^19)
sage: R2(4).square_root()
2 + O(2^18)
sage: R2(9).square_root() == R2(3) or R2(9).square_root() == R2(-3)
True
sage: R2(17).square_root()
1 + 2^3 + 2^5 + 2^6 + 2^7 + 2^9 + 2^10 + 2^13 + 2^16 + 2^17 + O(2^19)
sage: R3 = Zp(5,20,'capped-abs')
```

```
sage: R3(1).square_root()
1 + O(5^20)
sage: R3(-1).square_root() == R3.teichmuller(2) or R3(-1).square_root() == R3.teichmuller(3)
True
```

str(*mode=None*)

Returns a string representation of self.

EXAMPLES:

```
sage: Zp(5,5,print_mode='bars')(1/3).str()[3:]
'1|3|1|3|2'
```

trace(*ground=None*)

Returns the trace of this p-adic element over the ground ring

INPUT:

- `ground` – a subring of the ground ring (default: base ring)

OUTPUT:

- `element` – the trace of this p-adic element over the ground ring

EXAMPLES:

```
sage: Zp(5,5)(5).trace()
5 + O(5^6)
```

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*)

Returns 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 integer and rational.

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
```

TESTS:

```
sage: R(1).valuation()
0
sage: R(2).valuation()
0
sage: R(5).valuation()
1
sage: R(10).valuation()
1
sage: R(25).valuation()
2
sage: R(50).valuation()
2
sage: R = Qp(17, 4)
sage: a = R(2*17^2)
sage: a.valuation()
2
sage: R = Qp(5, 4)
sage: R(0).valuation()
+Infinity
sage: R(1).valuation()
0
sage: R(2).valuation()
0
sage: R(5).valuation()
1
sage: R(10).valuation()
1
sage: R(25).valuation()
2
sage: R(50).valuation()
2
sage: R(1/2).valuation()
0
sage: R(1/5).valuation()
-1
sage: R(1/10).valuation()
-1
sage: R(1/25).valuation()
-2
sage: R(1/50).valuation()
-2

sage: K.<a> = Qq(25)
sage: K(0).valuation()
+Infinity

sage: R(1/50).valuation(5)
-2
sage: R(1/50).valuation(3)
Traceback (most recent call last):
...
ValueError: Ring (5-adic Field with capped relative precision 4) residue field of the wrong
```

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 divisior 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 paramter:

```
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: 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)
(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)
```

TESTS:

The implementation also works over extensions:

```
sage: K = Qp(3)
sage: R.<a> = K[]
sage: L.<a> = K.extension(a^3-3)
sage: (a+3).xgcd(3)
(1 + O(a^60),
 a^-1 + 2*a + a^3 + 2*a^4 + 2*a^5 + 2*a^8 + 2*a^9
  + 2*a^12 + 2*a^13 + 2*a^16 + 2*a^17 + 2*a^20 + 2*a^21 + 2*a^24
  + 2*a^25 + 2*a^28 + 2*a^29 + 2*a^32 + 2*a^33 + 2*a^36 + 2*a^37
  + 2*a^40 + 2*a^41 + 2*a^44 + 2*a^45 + 2*a^48 + 2*a^49 + 2*a^52
  + 2*a^53 + 2*a^56 + 2*a^57 + O(a^59),
 0)

sage: R = Zp(3)
sage: S.<a> = R[]
sage: S.<a> = R.extension(a^3-3)
sage: (a+3).xgcd(3)
(a + O(a^61),
 1 + 2*a^2 + a^4 + 2*a^5 + 2*a^6 + 2*a^9 + 2*a^10
  + 2*a^13 + 2*a^14 + 2*a^17 + 2*a^18 + 2*a^21 + 2*a^22 + 2*a^25
  + 2*a^26 + 2*a^29 + 2*a^30 + 2*a^33 + 2*a^34 + 2*a^37 + 2*a^38
  + 2*a^41 + 2*a^42 + 2*a^45 + 2*a^46 + 2*a^49 + 2*a^50 + 2*a^53
  + 2*a^54 + 2*a^57 + 2*a^58 + O(a^60),
 0)

sage: K = Qp(3)
sage: R.<a> = K[]
sage: L.<a> = K.extension(a^2-2)
sage: (a+3).xgcd(3)
(1 + O(3^20),
 2*a + (a + 1)*3 + (2*a + 1)*3^2 + (a + 2)*3^4 + 3^5
  + (2*a + 2)*3^6 + a*3^7 + (2*a + 1)*3^8 + (a + 2)*3^10 + 3^11
  + (2*a + 2)*3^12 + a*3^13 + (2*a + 1)*3^14 + (a + 2)*3^16
  + 3^17 + (2*a + 2)*3^18 + a*3^19 + O(3^20),
 0)

sage: R = Zp(3)
sage: S.<a> = R[]
sage: S.<a> = R.extension(a^2-2)
sage: (a+3).xgcd(3)
(1 + O(3^20),
 2*a + (a + 1)*3 + (2*a + 1)*3^2 + (a + 2)*3^4 + 3^5
  + (2*a + 2)*3^6 + a*3^7 + (2*a + 1)*3^8 + (a + 2)*3^10 + 3^11
  + (2*a + 2)*3^12 + a*3^13 + (2*a + 1)*3^14 + (a + 2)*3^16 + 3^17
  + (2*a + 2)*3^18 + a*3^19 + O(3^20),
 0)
```

For elements with a fixed modulus:

```
sage: R = ZpFM(3)
sage: R(3).xgcd(9)
(3 + O(3^20), 1 + O(3^20), O(3^20))
```

And elements with a capped absolute precision:

```
sage: R = ZpCA(3)
sage: R(3).xgcd(9)
(3 + O(3^20), 1 + O(3^19), O(3^20))
```

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: sage.rings.padics.padic_capped_relative_element.pAdicTemplateElement

add_bigoh(*absprec*)

Returns 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

EXAMPLE:

```
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)
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*)

Returns 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

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)
```

TESTS:

```
sage: aa.is_equal_to(a), aa.is_equal_to(a, 4), aa.is_equal_to(a, 5)
(False, True, False)
sage: aa.is_equal_to(a, 15)
Traceback (most recent call last):
...
PrecisionError: Elements not known to enough precision

sage: b.is_equal_to(a), b.is_equal_to(a, 2)
(True, True)
sage: b.is_equal_to(a, 5)
Traceback (most recent call last):
...
PrecisionError: Elements not known to enough precision

sage: bb.is_equal_to(b, 3)
True
sage: bb.is_equal_to(b, 4)
Traceback (most recent call last):
...
PrecisionError: Elements not known to enough precision

sage: b.is_equal_to(c, 2), b.is_equal_to(c, 3)
(True, False)
sage: b.is_equal_to(c, 4)
Traceback (most recent call last):
...
PrecisionError: Elements not known to enough precision

sage: cc.is_equal_to(c, 2), cc.is_equal_to(c, 4), cc.is_equal_to(c, 5)
(True, True, False)
```

is_zero(*absprec=None*)

Determines 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
```

list(*lift_mode='simple'*, *start_val=None*)

Returns a list of coefficients in a power series expansion of self in terms of π. If self is a field element, they start at $\pi^{\text{valuation}}$, if a ring element at π^0.

For each lift mode, this funciton 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 $\mathbb{F}_p$ 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 also be p-adic elements.

INPUT:

- `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). If `start_val` is larger than the valuation of this element a `ValueError` is raised.

OUTPUT:

- the list of coefficients of this element. For base elements these will be integers if `lift_mode` is `'simple'` or `'smallest'`, and elements of `self.parent()` if `lift_mode` is `'teichmuller'`.

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: L = a.list(); L
[3, 4, 4, 0, 4]
sage: sum([L[i] * 7^i for i in range(len(L))]) == a
True
sage: L = a.list('smallest'); L
[3, -3, -2, 1, -3, 1]
sage: sum([L[i] * 7^i for i in range(len(L))]) == a
True
sage: L = a.list('teichmuller'); L
[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([L[i] * 7^i for i in range(len(L))])
3 + 4*7 + 4*7^2 + 4*7^4 + O(7^6)

sage: R(0, 7).list()
[]

sage: R = Qp(7,4); a = R(6*7+7**2); a.list()
```

```
[6, 1]
sage: a.list('smallest')
[-1, 2]
sage: a.list('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)]
```

TESTS:

Check to see that trac ticket #10292 is resolved:

```
sage: E = EllipticCurve('37a')
sage: R = E.padic_regulator(7)
sage: len(R.list())
19
```

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()

Returns 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
```

teichmuller_list()

Returns a list [a_0, a_1,..., a_n] such that

- $a_i^q = a_i$, where q is the cardinality of the residue field,
- `self.unit_part()` $= \sum_{i=0}^n a_i p^i$, and
- if $a_i \neq 0$, the absolute precision of a_i is self.precision_relative() - i

EXAMPLES:

```
sage: R = Qp(5,5); R(70).list('teichmuller') #indirect doctest
[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)]
```

unit_part()

Returns 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)
<type '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)
<type 'sage.rings.padics.padic_capped_relative_element.pAdicCappedRelativeElement'>
sage: a = R(2*17^2); a
2*17^2 + O(17^6)
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*)

Returns 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 integer and rational.

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))
```

sage.rings.padics.padic_capped_relative_element.**base_p_list**(*n*, *pos*, *prime_pow*)

Returns 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]
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: sage.rings.padics.padic_capped_relative_element.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))
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: cannot coerce from the given integer mod ring (not a power of the same prime)

sage: R(Integers(48)(3))
Traceback (most recent call last):
...
TypeError: cannot coerce from the given integer mod ring (not a power of the same prime)
```

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)
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))
4 + 4*5 + 4*5^2 + 4*5^3 + 4*5^4 + O(5^5)
sage: pari(R(0))
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
sage: R = Qp(7,4); a = R(8/7); a.lift()
8/7
```

residue (*absprec=1*)

Reduces this element modulo p^{absprec}.

INPUT:

- `absprec` - a non-negative integer (default: `1`)

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
sage: a.residue(2)
8

sage: K = Qp(7,4)
sage: a = K(8)
sage: a.residue(1)
1
sage: a.residue(2)
8
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.
```

TESTS:

```
sage: R = Zp(7,4)
sage: a = R(8)
sage: a.residue(0)
0
sage: a.residue(-1)
Traceback (most recent call last):
...
ValueError: cannot reduce modulo a negative power of p.
sage: a.residue(5)
Traceback (most recent call last):
...
PrecisionError: not enough precision known in order to compute residue.
```

class `sage.rings.padics.padic_capped_relative_element.`**`pAdicCoercion_QQ_CR`**

Bases: `sage.rings.morphism.RingHomomorphism_coercion`

The canonical inclusion from the rationals to a capped relative field.

EXAMPLES:

```
sage: f = Qp(5).coerce_map_from(QQ); f
Ring Coercion 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: sage.rings.morphism.RingHomomorphism_coercion

The canonical inclusion from the integer ring to a capped relative ring.

EXAMPLES:

```
sage: f = Zp(5).coerce_map_from(ZZ); f
Ring Coercion 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: sage.rings.morphism.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: sage.rings.morphism.RingMap

The map from a capped relative ring back to the ring of integers that returns the 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_QQ_CR`**

Bases: `sage.categories.morphism.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`()

Returns 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: `sage.rings.padics.padic_generic_element.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)
```

`lift_to_precision`(*absprec=None*)

Returns 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 + O(5^20)
sage: a.lift_to_precision(10000)
5 + O(5^20)
```

padded_list (*n*, *lift_mode='simple'*)

Returns a list of coefficients of the uniformizer π starting with π^0 up to π^n exclusive (padded with zeros if needed).

For a field element of valuation v, starts at π^v instead.

INPUT:

- `n` - an integer
- `lift_mode` - 'simple', 'smallest' or 'teichmuller'

EXAMPLES:

```
sage: R = Zp(7,4,'capped-abs'); a = R(2*7+7**2); a.padded_list(5)
[0, 2, 1, 0, 0]
sage: R = Zp(7,4,'fixed-mod'); a = R(2*7+7**2); a.padded_list(5)
[0, 2, 1, 0, 0]
```

For elements with positive valuation, this function will return a list with leading 0s if the parent is not a field:

```
sage: R = Zp(7,3,'capped-rel'); a = R(2*7+7**2); a.padded_list(5)
[0, 2, 1, 0, 0]
sage: R = Qp(7,3); a = R(2*7+7**2); a.padded_list(5)
[2, 1, 0, 0]
sage: a.padded_list(3)
[2, 1]
```

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.

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: `sage.rings.padics.padic_capped_absolute_element.pAdicTemplateElement`

`add_bigoh`(*absprec*)

Returns a new element with absolute precision decreased to `absprec`. The precision never increases.

INPUT:

- `absprec` – an integer

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*)

Determines 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*)

Determines 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
```

list(*lift_mode='simple'*, *start_val=None*)

Returns a list of coefficients of p starting with p^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 $\mathbb{F}_p$ 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 also be p-adic elements.

INPUT:

- `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). If `start_val` is larger than the valuation of this element a `ValueError` is raised.

Note: Use slice operators to get a particular range.

EXAMPLES:

```
sage: R = ZpCA(7,6); a = R(12837162817); a
3 + 4*7 + 4*7^2 + 4*7^4 + O(7^6)
sage: L = a.list(); L
[3, 4, 4, 0, 4]
sage: sum([L[i] * 7^i for i in range(len(L))]) == a
True
sage: L = a.list('smallest'); L
[3, -3, -2, 1, -3, 1]
sage: sum([L[i] * 7^i for i in range(len(L))]) == a
True
sage: L = a.list('teichmuller'); L
[3 + 4*7 + 6*7^2 + 3*7^3 + 2*7^5 + O(7^6),
O(7^5),
5 + 2*7 + 3*7^3 + O(7^4),
1 + O(7^3),
3 + 4*7 + O(7^2),
5 + O(7)]
sage: sum([L[i] * 7^i for i in range(len(L))])
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: a.list()
[0, 0, 0, 3, 2]
```

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
```

teichmuller_list()

Returns a list $[a_0, a_1, \ldots, a_n]$ such that

- $a_i^q = a_i$, where q is the cardinality of the residue field,
- `self` equals $\sum_{i=0}^n a_i \pi^i$, and
- if $a_i \neq 0$, the absolute precision of a_i is `self.precision_relative() - i`

EXAMPLES:

```
sage: R = ZpCA(5,5); R(14).list('teichmuller') #indirect doctest
[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)]
```

unit_part()

Returns 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)
<type 'sage.rings.padics.padic_capped_absolute_element.pAdicCappedAbsoluteElement'>
sage: R(0).unit_part()
O(17^0)
```

val_unit()

Returns 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))
```

`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_pAdicCappedAbsoluteElemen
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: `sage.rings.padics.padic_capped_absolute_element.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()

sage: R = ZpCA(3) sage: R(10).lift() 10 sage: R(-1).lift() 3486784400

multiplicative_order()

Returns 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*)

Reduces `self` modulo p^{absprec}.

INPUT:

- `absprec` - a non-negative integer (default: 1)

OUTPUT:

This element reduced modulo p^{absprec} as an element of $\mathbf{Z}/p^{\mathrm{absprec}}\mathbf{Z}$

EXAMPLES:

```
sage: R = Zp(7,4,'capped-abs')
sage: a = R(8)
sage: a.residue(1)
 1
sage: a.residue(2)
8
```

TESTS:

```
sage: a.residue(0)
0
sage: a.residue(-1)
Traceback (most recent call last):
...
ValueError: cannot reduce modulo a negative power of p.
sage: a.residue(5)
Traceback (most recent call last):
...
PrecisionError: not enough precision known in order to compute residue.
```

class `sage.rings.padics.padic_capped_absolute_element.`**`pAdicCoercion_ZZ_CA`**

Bases: `sage.rings.morphism.RingHomomorphism_coercion`

The canonical inclusion from the ring of integers to a capped absolute ring.

EXAMPLES:

```
sage: f = ZpCA(5).coerce_map_from(ZZ); f
Ring Coercion morphism:
  From: Integer Ring
  To:   5-adic Ring with capped absolute precision 20
```

`section``()`

Returns 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: `sage.rings.morphism.RingMap`

The map from a capped absolute ring back to the ring of integers that returns the 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_QQ_CA`**

Bases: `sage.categories.morphism.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: sage.rings.padics.padic_generic_element.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)
```

lift_to_precision(*absprec=None*)

Returns 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 + O(5^20)
sage: a.lift_to_precision(10000)
5 + O(5^20)
```

padded_list (*n*, *lift_mode='simple'*)

Returns a list of coefficients of the uniformizer π starting with π^0 up to π^n exclusive (padded with zeros if needed).

For a field element of valuation v, starts at π^v instead.

INPUT:

- `n` - an integer
- `lift_mode` - 'simple', 'smallest' or 'teichmuller'

EXAMPLES:

```
sage: R = Zp(7,4,'capped-abs'); a = R(2*7+7**2); a.padded_list(5)
[0, 2, 1, 0, 0]
sage: R = Zp(7,4,'fixed-mod'); a = R(2*7+7**2); a.padded_list(5)
[0, 2, 1, 0, 0]
```

For elements with positive valuation, this function will return a list with leading 0s if the parent is not a field:

```
sage: R = Zp(7,3,'capped-rel'); a = R(2*7+7**2); a.padded_list(5)
[0, 2, 1, 0, 0]
sage: R = Qp(7,3); a = R(2*7+7**2); a.padded_list(5)
[2, 1, 0, 0]
sage: a.padded_list(3)
[2, 1]
```

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.

`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` – the 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, pAdicCappedAb
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.`**`FMElement`**

Bases: `sage.rings.padics.padic_fixed_mod_element.pAdicTemplateElement`

`add_bigoh`(*absprec*)

Returns a new element truncated modulo π^{absprec}.

INPUT:

- `absprec` – an integer

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 + O(7^4)
```

`is_equal_to`(*_right*, *absprec=None*)

Returns 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
```

list(*lift_mode='simple'*)

Returns a list of coefficients of π^i starting with π^0.

INPUT:

- `lift_mode` – `'simple'`, `'smallest'` or `'teichmuller'` (default: `'simple'`:)

OUTPUT:

The list of coefficients of this element.

Note:

- Returns a list $[a_0, a_1, \ldots, a_n]$ so that each a_i is an integer and $\sum_{i=0}^{n} a_i \cdot p^i$ is equal to this element modulo the precision cap.
- If `lift_mode` is `'simple'`, $0 \leq a_i < p$.
- If `lift_mode` is `'smallest'`, $-p/2 < a_i \leq p/2$.
- If `lift_mode` is `'teichmuller'`, $a_i^q = a_i$, modulo the precision cap.

EXAMPLES:

```
sage: R = ZpFM(7,6); a = R(12837162817); a
3 + 4*7 + 4*7^2 + 4*7^4 + O(7^6)
sage: L = a.list(); L
[3, 4, 4, 0, 4]
sage: sum([L[i] * 7^i for i in range(len(L))]) == a
True
sage: L = a.list('smallest'); L
[3, -3, -2, 1, -3, 1]
sage: sum([L[i] * 7^i for i in range(len(L))]) == a
True
sage: L = a.list('teichmuller'); L
[3 + 4*7 + 6*7^2 + 3*7^3 + 2*7^5 + O(7^6),
O(7^6),
5 + 2*7 + 3*7^3 + 6*7^4 + 4*7^5 + O(7^6),
1 + O(7^6),
3 + 4*7 + 6*7^2 + 3*7^3 + 2*7^5 + O(7^6),
5 + 2*7 + 3*7^3 + 6*7^4 + 4*7^5 + O(7^6)]
sage: sum([L[i] * 7^i for i in range(len(L))])
3 + 4*7 + 4*7^2 + 4*7^4 + O(7^6)
```

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
```

teichmuller_list()

Returns a list [a_0, a_1,..., a_n] such that

- $a_i^q = a_i$
- self.unit_part() = $\sum_{i=0}^{n} a_i \pi^i$

EXAMPLES:

```
sage: R = ZpFM(5,5); R(14).list('teichmuller') #indirect doctest
[4 + 4*5 + 4*5^2 + 4*5^3 + 4*5^4 + O(5^5),
3 + 3*5 + 2*5^2 + 3*5^3 + 5^4 + O(5^5),
2 + 5 + 2*5^2 + 5^3 + 3*5^4 + O(5^5),
1 + O(5^5),
4 + 4*5 + 4*5^2 + 4*5^3 + 4*5^4 + O(5^5)]
```

unit_part()

Returns 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 + O(17^4)
sage: R(18*17).unit_part()
1 + 17 + O(17^4)
sage: R(0).unit_part()
O(17^4)
sage: type(R(5).unit_part())
<type 'sage.rings.padics.padic_fixed_mod_element.pAdicFixedModElement'>
sage: R = ZpFM(5, 5); a = R(75); a.unit_part()
3 + O(5^5)
```

val_unit()

Returns 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 + O(5^5))
```

```
sage: b.val_unit()
(5, O(5^5))
```

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 + O(5^20)
```

class sage.rings.padics.padic_fixed_mod_element.**pAdicCoercion_ZZ_FM**

Bases: sage.rings.morphism.RingHomomorphism_coercion

The canonical inclusion from ZZ to a fixed modulus ring.

EXAMPLES:

```
sage: f = ZpFM(5).coerce_map_from(ZZ); f
Ring Coercion morphism:
  From: Integer Ring
  To:   5-adic Ring of fixed modulus 5^20
```

section()

Returns a map back to ZZ 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: sage.rings.morphism.RingMap

The map from a fixed modulus ring back to ZZ that returns the 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 ZZ, 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_QQ_FM**

Bases: sage.categories.morphism.Morphism

The inclusion map from QQ 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: sage.rings.padics.padic_fixed_mod_element.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 + O(5^20)
sage: R(75)
75 + O(5^20)
sage: R(0)
0 + O(5^20)

sage: R(-1)
95367431640624 + O(5^20)
sage: R(-5)
95367431640620 + O(5^20)
```

Construct from rationals:

```
sage: R(1/2)
47683715820313 + O(5^20)
sage: R(-7875/874)
9493096742250 + O(5^20)
sage: R(15/425)
Traceback (most recent call last):
...
ValueError: p divides denominator
```

Construct from IntegerMod:

```
sage: R(Integers(125)(3))
3 + O(5^20)
```

```
sage: R(Integers(5)(3))
3 + O(5^20)
sage: R(Integers(5^30)(3))
3 + O(5^20)
sage: R(Integers(5^30)(1+5^23))
1 + O(5^20)
sage: R(Integers(49)(3))
Traceback (most recent call last):
...
TypeError: cannot coerce from the given integer mod ring (not a power of the same prime)

sage: R(Integers(48)(3))
Traceback (most recent call last):
...
TypeError: cannot coerce from the given integer mod ring (not a power of the same prime)
```

Some other conversions:

```
sage: R(R(5))
5 + O(5^20)
```

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 $i^{\mathrm{prec}_c\mathrm{ap}}$ 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())
<type '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 + 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*)

Reduce `self` modulo p^{absprec}.

INPUT:

- `absprec` – an integer (default: `1`)

OUTPUT:

This element reduced modulo p^{absprec} as an element of $\mathbf{Z}/p^{\text{absprec}}\mathbf{Z}$.

EXAMPLES:

```
sage: R = Zp(7,4,'fixed-mod')
sage: a = R(8)
sage: a.residue(1)
1
sage: a.residue(2)
8
```

TESTS:

```
sage: R = Zp(7,4,'fixed-mod')
sage: a = R(8)
sage: a.residue(0)
0
sage: a.residue(-1)
Traceback (most recent call last):
...
ValueError: Cannot reduce modulo a negative power of p.
sage: a.residue(5)
Traceback (most recent call last):
...
PrecisionError: Not enough precision known in order to compute residue.
```

class `sage.rings.padics.padic_fixed_mod_element.`**`pAdicTemplateElement`**

Bases: `sage.rings.padics.padic_generic_element.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)
```

lift_to_precision (*absprec=None*)

Returns 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 + O(5^20)
sage: a.lift_to_precision(10000)
5 + O(5^20)
```

padded_list (*n*, *lift_mode='simple'*)

Returns a list of coefficients of the uniformizer π starting with π^0 up to π^n exclusive (padded with zeros if needed).

For a field element of valuation v, starts at π^v instead.

INPUT:

- n - an integer
- lift_mode - 'simple', 'smallest' or 'teichmuller'

EXAMPLES:

```
sage: R = Zp(7,4,'capped-abs'); a = R(2*7+7**2); a.padded_list(5)
[0, 2, 1, 0, 0]
sage: R = Zp(7,4,'fixed-mod'); a = R(2*7+7**2); a.padded_list(5)
[0, 2, 1, 0, 0]
```

For elements with positive valuation, this function will return a list with leading 0s if the parent is not a field:

```
sage: R = Zp(7,3,'capped-rel'); a = R(2*7+7**2); a.padded_list(5)
[0, 2, 1, 0, 0]
sage: R = Qp(7,3); a = R(2*7+7**2); a.padded_list(5)
[2, 1, 0, 0]
sage: a.padded_list(3)
[2, 1]
```

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.

sage.rings.padics.padic_fixed_mod_element.**unpickle_fme_v2**(*cls*, *parent*, *value*)

Unpickles a capped relative element.

EXAMPLES:

```
sage: from sage.rings.padics.padic_fixed_mod_element import pAdicFixedModElement, unpickle_fme_v
sage: R = ZpFM(5)
sage: a = unpickle_fme_v2(pAdicFixedModElement, R, 17*25); a
2*5^2 + 3*5^3 + O(5^20)
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: `sage.rings.padics.padic_generic_element.pAdicGenericElement`

`frobenius`(*arithmetic=True*)

Returns the image of this element under the Frobenius automorphism applied to its parent.

INPUT:

- `self` – an element of an unramified extension.
- `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
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: 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*)

Reduces this element modulo π^{absprec}.

INPUT:

- `absprec` - a non-negative integer (default: `1`)

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): intial version

EXAMPLES:

Unramified case:

```
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: residue() not implemented in extensions for absprec larger than one.
```

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
sage: a.residue(2)
Traceback (most recent call last):
...
NotImplementedError: residue() not implemented in extensions for absprec larger than one.
```

TESTS:

sage: K = Qp(3,5) sage: S.<a> = R[] sage: W.<a> = R.extension(a^2 + 9*a + 1) sage: (a/3).residue(0) Traceback (most recent call last): ... ValueError: element must have non-negative valuation in order to compute residue.

sage: R = ZpFM(3,5) sage: S.<a> = R[] sage: W.<a> = R.extension(a^2 + 9*a + 1) sage: W.one().residue(0) 0 sage: a.residue(-1) Traceback (most recent call last): ... ValueError: cannot reduce modulo a negative power of the uniformizer. sage: a.residue(16) Traceback (most recent call last): ... PrecisionError: not enough precision known in order to compute residue.

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: sage.rings.padics.padic_ext_element.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 $\mathbb{Q}_p$ or $\mathbb{Z}_p$.

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)
```

TESTS:

```
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: ((1+2*w)^5).norm()
1 + 5^2 + O(5^5)
sage: ((1+2*w)).norm()^5
```

```
1 + 5^2 + O(5^5)
sage: R = ZpFM(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: ((1+2*w)^5).norm()
1 + 5^2 + O(5^5)
sage: ((1+2*w)).norm()^5
1 + 5^2 + O(5^5)
```

TESTS:

Check that #11586 has been resolved:

```
sage: R.<x> = QQ[]
sage: f = x^2 + 3*x + 1
sage: M.<a> = Qp(7).extension(f)
sage: M(7).norm()
7^2 + O(7^22)
sage: b = 7*a + 35
sage: b.norm()
4*7^2 + 7^3 + O(7^22)
sage: b*b.frobenius()
4*7^2 + 7^3 + O(7^22)
```

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 $\mathbb{Q}_p$ or $\mathbb{Z}_p$.

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)
```

TESTS:

```
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 = (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)
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 = (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 $\mathbb{Z}_p$ and $\mathbb{Q}_p$ with capped relative 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:

- `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 $\mathbb{Q}_p$ or $\mathbb{Z}_p$. 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` (see `sage/libs/ntl/decl.pxi` and `c_lib/src/ntl_wrap.cpp`)
- `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^{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
Eisenstein Extension of 5-adic Ring with capped relative precision 5 in w defined by (1 + O(5^5))*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 +
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
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()
Eisenstein Extension of 5-adic Field with capped relative precision 5 in w defined by (1 + O(5^5))*x^
```

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 +
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)
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.ex
sage: z = (1+w)^5; repr(z)
'...4110403113210310442221311242000111011201102002023303214332011214403232013144001400444441030421100
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,
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
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
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)
```

NOTES:

```
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: `sage.rings.padics.padic_ZZ_pX_element.pAdicZZpXElement`

Creates an element of a capped relative precision, unramified or Eisenstein extension of $\mathbb{Z}_p$ or $\mathbb{Q}_p$.

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
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)
```

TESTS:

Check that trac ticket #3865 is fixed:

```
sage: W(gp('3 + O(5^10)'))
3 + O(w^3125)
```

Check that trac ticket #13612 has been fixed:

```
sage: R = Zp(3)
sage: S.<a> = R[]
sage: W.<a> = R.extension(a^2+1)
sage: W(W.residue_field().zero())
O(3)

sage: K = Qp(3)
sage: S.<a> = K[]
sage: L.<a> = K.extension(a^2+1)
sage: L(L.residue_field().zero())
O(3)
```

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 = 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*)

Returns 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 = 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*)

Returns a `pAdicZZpXCRElement` 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. 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
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(
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*
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
```

list (*lift_mode='simple'*)

Returns 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`.

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).list()
[0, 1, 0, 4, 0, 2, 1, 2, 4, 1]
sage: (y>>9).list('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: y.list()
[[], [0, 4], [3, 1, 3], [0, 0, 4], [0, 0, 1]]
sage: y.list('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: W(0).list()
[]
sage: W(0,4).list()
[0]
sage: A(0,4).list()
[[]]
```

matrix_mod_pn ()

Returns 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]
```

TESTS:

Check that trac ticket #13617 has been fixed:

```
sage: W.zero().matrix_mod_pn()
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
```

precision_absolute()

Returns the absolute precision of `self`, ie 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()

Returns the relative precision of `self`, ie 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_list()

Returns a list [a_0, a_1,..., 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`

EXAMPLES:

```
sage: R.<a> = ZqCR(5^4,4)
sage: L = a.teichmuller_list(); L
[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
sage: sum([5^i*L[i] for i in range(4)])
a + O(5^4)
sage: all([L[i]^625 == L[i] for i in range(4)])
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_list(); 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
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_list(); 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
sage: sum([w^i*L[i] for i in range(len(L))])
3 + O(w^9)
```

unit_part()

Returns the unit part of `self`, 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)
```

TESTS:

We check that trac ticket #13616 is resolved:

```
sage: z = (1+w)^5
sage: y = z - 1
sage: t=y-y
sage: t.unit_part()
O(w^0)
```

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 $\mathbb{Q}_p$ or $\mathbb{Z}_p$. 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 (see `sage/libs/ntl/decl.pxi` and `c_lib/src/ntl_wrap.cpp`)
- `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^{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
Eisenstein Extension of 5-adic Ring with capped absolute precision 5 in w defined by (1 + O(5^5))*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 +
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
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()
Eisenstein Extension of 5-adic Field with capped relative precision 5 in w defined by (1 + O(5^5))*x^
```

An unramified extension:

```
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 +
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: R = ZpCA(5, print_mode='digits'); S.<x> = ZZ[]; f = x^5 + 75*x^3 - 15*x^2 + 125*x -5; W.<w> = R
sage: z = (1+w)^5; repr(z)
'...41104031132103104422213112420001110112011020020233032143320112144032320131440014004444410304211000
sage: R = ZpCA(5, print_mode='bars'); S.<x> = ZZ[]; 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,
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
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> =
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: 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)
```

NOTES:

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: `sage.rings.padics.padic_ZZ_pX_element.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
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)
```

TESTS:

Check that trac ticket #13600 is fixed:

```
sage: K = W.fraction_field()
sage: W(K.zero())
O(w^25)
sage: W(K.one())
1 + O(w^25)
sage: W(K.zero().add_bigoh(3))
O(w^3)
```

Check that trac ticket #3865 is fixed:

sage: W(gp('5 + O(5^2)')) w^5 + 2*w^7 + 4*w^9 + O(w^10)

Check that trac ticket #13612 has been fixed:

```
sage: R = ZpCA(3)
sage: S.<a> = R[]
sage: W.<a> = R.extension(a^2+1)
sage: W(W.residue_field().zero())
O(3)
```

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*)

Returns 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(
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*
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
```

list (*lift_mode='simple'*)

Returns 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.

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).list()
[0, 1, 0, 4, 0, 2, 1, 2, 4, 1]
sage: (y>>9).list('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(5^5)
sage: y.list()
[[], [0, 4], [3, 1, 3], [0, 0, 4], [0, 0, 1]]
sage: y.list('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).list()
[0]
sage: A(0,4).list()
[[]]
```

matrix_mod_pn()

Returns 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()
[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]
```

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_list()

Returns a list [a_0, a_1,..., 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`

EXAMPLES:

```
sage: R.<a> = Zq(5^4,4)
sage: L = a.teichmuller_list(); L
[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
sage: sum([5^i*L[i] for i in range(4)])
a + O(5^4)
sage: all([L[i]^625 == L[i] for i in range(4)])
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_list(); 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
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_list(); 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
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 +
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 +
sage: y.parent()
Eisenstein Extension of 5-adic Field with capped relative precision 5 in w defined by (1 + O
```

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 $\mathbb{Z}_p$ 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^{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
```

```
Eisenstein Extension of 5-adic Ring of fixed modulus 5^5 in w defined by (1 + O(5^5))*x^5 + (O(5^5))
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 
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^2
sage: y.valuation()
4
sage: y.precision_relative()
21
sage: y.precision_absolute()
25
sage: z - (y << 1)
1 + O(w^25)
```

An unramified extension:

```
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 +
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(5^5)
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)
```

Different printing modes:

```
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
sage: z = (1+w)^5; repr(z)
'...41104031132103104422213112420001110112011020020233032143320112144032320131440014004444103042110
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,
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.e
sage: z = (1+w)^5; z
6 + 95367431640505*w + 25*w^2 + 95367431640560*w^3 + 5*w^4 + O(w^100)
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> =
sage: y = (1+w)^5 - 1; y
w^5 * (2090041 + 95367431439401*w + 76293946571402*w^2 + 57220458985049*w^3 + 57220459001160*w^4) + 
```

AUTHORS:

- David Roe (2008-01-01) initial version

`sage.rings.padics.padic_ZZ_pX_FM_element.`**`make_ZZpXFMElement`**(*parent*, *f*)

Creates 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: sage.rings.padics.padic_ZZ_pX_element.pAdicZZpXElement

Creates an element of a fixed modulus, unramified or eisenstein extension of $\mathbb{Z}_p$ or $\mathbb{Q}_p$.

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
```

TESTS:

Check that trac ticket #3865 is fixed:

```
sage: W(gp('2 + O(5^2)'))
2 + O(w^25)
```

Check that trac ticket #13612 has been fixed:

```
sage: R = ZpFM(3)
sage: S.<a> = R[]
sage: W.<a> = R.extension(a^2+1)
sage: W(W.residue_field().zero())
O(3^20)
```

add_bigoh(*absprec*)

Returns a new element truncated modulo pi^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 + O(7^4)
```

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 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*)

Returns whether the valuation of `self` is at least `absprec`. If `absprec` is `None`, returns 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
sage: a.is_zero(21)
False
```

lift_to_precision (*absprec=None*)

Returns `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 + O(w^25)
```

list (*lift_mode='simple'*)

Returns 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
  - 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 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 `self` is the sum of that list times powers of the uniformizer.

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^2
sage: (y>>9).list()
[0, 1, 0, 4, 0, 2, 1, 2, 4, 1, 0, 1, 2, 3, 1, 1, 4, 1, 2, 4, 1, 0, 4, 3]
sage: (y>>9).list('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
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^2
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(5^5)
sage: y.list()
[[], [0, 4], [3, 1, 3], [0, 0, 4], [0, 0, 1]]
sage: y.list('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).list()
[0]
sage: A(0,4).list()
[[]]
```

matrix_mod_pn()

Returns 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 emph{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()
[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 $\mathbb{Q}_p$ or $\mathbb{Z}_p$.

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)
```

precision_absolute()

Returns 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 + O(w^25)
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
```

precision_relative()

Returns 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^23 + O(w^25)
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
```

teichmuller_list()

Returns a list [a_0, a_1,..., 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()

EXAMPLES:

```
sage: R.<a> = ZqFM(5^4,4)
sage: L = a.teichmuller_list(); L
[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
sage: sum([5^i*L[i] for i in range(4)])
a + O(5^4)
sage: all([L[i]^625 == L[i] for i in range(4)])
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_list(); L
[1 + O(w^9), 5 + 5*w^3 + w^6 + 4*w^7 + O(w^9), 3 + 3*w^3 + w^7 + O(w^9), 3 + 3*w^3 + w^7 + O
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_list(); L
[3 + 3*w^3 + w^7 + O(w^9), O(w^9), O(w^9), 4 + 5*w^3 + w^6 + 4*w^7 + O(w^9), O(w^9), O(w^9),
sage: sum([w^i*L[i] for i in range(len(L))])
3 + O(w^9)
```

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 $\mathbb{Q}_p$ or $\mathbb{Z}_p$.

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()

Returns 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
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 + O(w^25)
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
```

The unit part inserts nonsense digits if this element has positive valuation:

```
sage: (a-a).unit_part()
O(w^25)
```

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*)

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: `sage.rings.padics.pow_computer.PowComputer_class`

class `sage.rings.padics.pow_computer.`**`PowComputer_class`**

Bases: `sage.structure.sage_object.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: PC = PowComputer_ext_maker(3, 5, 10, 20, False, ntl.ZZ_pX([-3,0,1], 3^10), 'big','e',n
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: `sage.rings.padics.pow_computer_ext.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',nt
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',
sage: PC.speed_test(10, 10^6) # random
0.0090679999999991878
```

class sage.rings.padics.pow_computer_ext.**PowComputer_ZZ_pX_FM**

Bases: sage.rings.padics.pow_computer_ext.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: sage.rings.padics.pow_computer_ext.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: sage.rings.padics.pow_computer_ext.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
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: sage.rings.padics.pow_computer_ext.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: sage.rings.padics.pow_computer_ext.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: sage.rings.padics.pow_computer_ext.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: `sage.rings.padics.pow_computer.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',
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',
sage: ZZ_pX_eis_shift_test(A, [0, 1], 1, 5)
[1]
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={})`

Creates 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: `sage.structure.sage_object.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)
-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','
```

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
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
(3*a^4 + ... + 3) + (a^5 + ... + a)*5 + (3*a^3 + ... + 2)*5^2 + (3*a^5 + ... + 2)*5^3 + (4*a

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))
'...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'

sage: padic_printing.sep('|')
sage: padic_printing.mode('series')
```

class sage.rings.padics.padic_printing.**pAdicPrinter_class**

Bases: sage.structure.sage_object.SageObject

This class stores the printing options for a specific p-adic ring or field, and uses these to compute the representations of elements.

cmp_modes (*other*)

Returns a comparison of the printing modes of self and other.

Returns 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). 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.cmp_modes(S._printer)
0
sage: R = Qp(7)
sage: S = Qp(7,print_mode='val-unit')
sage: R == S
False
sage: R._printer.cmp_modes(S._printer)
-1
```

dict ()

Returns 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)'
```

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: `exceptions.ArithmeticError`

CHAPTER

TWENTYSIX

MISCELLANEOUS FUNCTIONS

This file contains some miscellaneous functions used by p-adics.

- `min` – a version of `min` that returns ∞ on empty input.
- `max` – a version of `max` that returns $-\infty$ on empty input.

AUTHORS:

- David Roe

`sage.rings.padics.misc.max(*L)`

Returns the maximum of the inputs, where the maximum of the empty list is `-infinity`.

EXAMPLES:

```
sage: from sage.rings.padics.misc import max
sage: max()
-Infinity
sage: max(2,3)
3
```

`sage.rings.padics.misc.min(*L)`

Returns the minimum of the inputs, where the minimum of the empty list is `infinity`.

EXAMPLES:

```
sage: from sage.rings.padics.misc import min
sage: min()
+Infinity
sage: min(2,3)
2
```

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

VALUE GROUPS OF DISCRETE VALUATIONS

This file defines additive subgroups of QQ generated by a rational number.

AUTHORS:

- Julian Rueth (2013-09-06): initial version

class `sage.rings.padics.discrete_value_group.`**`DiscreteValueGroup`**(*generator*, *category*)

Bases: `sage.structure.unique_representation.UniqueRepresentation`, `sage.structure.parent.Parent`

The value group of a discrete valuation, an additive subgroup of QQ generated by `generator`.

INPUT:

- `generator` – a rational number

EXAMPLES:

```
sage: D1 = DiscreteValueGroup(0); D1
DiscreteValueGroup(0)
sage: D2 = DiscreteValueGroup(4/3); D2
DiscreteValueGroup(4/3)
sage: D3 = DiscreteValueGroup(-1/3); D3
DiscreteValueGroup(1/3)
```

TESTS:

```
sage: TestSuite(D1).run()
sage: TestSuite(D2).run()
sage: TestSuite(D3).run()
```

`index`(*other*)

Return the index of `other` in this group.

INPUT:

- `other` – a subgroup of this group

EXAMPLES:

```
sage: DiscreteValueGroup(3/8).index(DiscreteValueGroup(3))
8
sage: DiscreteValueGroup(3).index(DiscreteValueGroup(3/8))
Traceback (most recent call last):
...
ValueError: `other` must be a subgroup of this group
sage: DiscreteValueGroup(3).index(DiscreteValueGroup(0))
+Infinity
```

```
sage: DiscreteValueGroup(0).index(DiscreteValueGroup(0))
1
sage: DiscreteValueGroup(0).index(DiscreteValueGroup(3))
Traceback (most recent call last):
...
ValueError: `other` must be a subgroup of this group
```

CHAPTER

TWENTYNINE

FROBENIUS ENDOMORPHISMS ON PADIC FIELDS

class sage.rings.padics.morphism.**FrobeniusEndomorphism_padics**

Bases: sage.rings.morphism.RingHomomorphism

A class implementing Frobenius endomorphisms on padic 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 padic 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 padic 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 endormorphism 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

THIRTY

INDICES AND TABLES

- Index
- Module Index
- Search Page

[RV] Rodriguez Villegas, Fernando. Experimental Number Theory. Oxford Graduate Texts in Mathematics 13, 2007.

r

INDEX

A

B

C

D

I

K

L

M

N

O

P

Q

R

S

T

U

V

X

Z