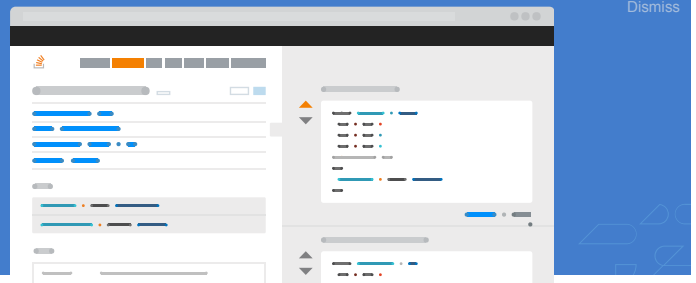


## Announcing Stack Overflow Documentation

We started with Q&A. Technical documentation is next, and we need your help.  
Whether you're a beginner or an experienced developer, you *can* contribute.

[Sign up and start helping →](#)[Learn more about Documentation →](#)

## Hidden features of Python [closed]

What are the lesser-known but useful features of the Python programming language?

- Try to limit answers to Python core.
- One feature per answer.
- Give an example and short description of the feature, not just a link to documentation.
- Label the feature using a title as the first line.

### Quick links to answers:

- [Argument Unpacking](#)
- [Braces](#)
- [Chaining Comparison Operators](#)
- [Decorators](#)
- [Default Argument Gotchas / Dangers of Mutable Default arguments](#)
- [Descriptors](#)
- [Dictionary default `.get` value](#)
- [Docstring Tests](#)
- [Ellipsis Slicing Syntax](#)
- [Enumeration](#)
- [For/else](#)
- [Function as `iter\(\)` argument](#)
- [Generator expressions](#)
- [`import this`](#)
- [In Place Value Swapping](#)
- [List stepping](#)
- [`\_\_missing\_\_` items](#)
- [Multi-line Regex](#)
- [Named string formatting](#)
- [Nested list/generator comprehensions](#)
- [New types at runtime](#)
- [`.pth` files](#)
- [ROT13 Encoding](#)
- [Regex Debugging](#)
- [Sending to Generators](#)
- [Tab Completion in Interactive Interpreter](#)
- [Ternary Expression](#)
- [`try/except/else`](#)
- [Unpacking+ `print\(\)` function](#)
- [`with` statement](#)

locked by [Bill the Lizard](#) Mar 9 '12 at 23:54

This question exists because it has historical significance, but **it is not considered a good, on-topic question for this site**, so please do not use it as evidence that you can ask similar questions here. This question and its answers are frozen and cannot be changed. More info: [help center](#).

closed as not constructive by [David](#), [Cody Gray](#), [casperOne](#) Feb 20 '12 at 20:09

As it currently stands, this question is not a good fit for our Q&A format. We expect answers to be supported by facts, references, or expertise, but this question will likely solicit debate, arguments, polling, or extended discussion. If you feel that this question can be improved and possibly reopened, [visit the help center](#) for guidance.

If this question can be reworded to fit the rules in the [help center](#), please [edit the question](#).

## 191 Answers

1 2 3 4 5 ... 7 next

### Chaining comparison operators:

```
>>> x = 5
>>> 1 < x < 10
True
>>> 10 < x < 20
False
>>> x < 10 < x*10 < 100
True
>>> 10 > x <= 9
True
>>> 5 == x > 4
True
```

In case you're thinking it's doing `1 < x`, which comes out as `True`, and then comparing `True < 10`, which is also `True`, then no, that's really not what happens (see the last example.) It's really translating into `1 < x` and `x < 10`, and `x < 10` and `10 < x * 10` and `x*10 < 100`, but with less typing and each term is only evaluated once.

edited Jan 18 '12 at 16:23

community wiki  
6 revs, 4 users 84%  
[Thomas Wouters](#)

- 121 That's very helpful. It should be standard for all languages. Sadly, it isn't. – [stalepretzel](#) Oct 17 '08 at 2:23
- 8 you should add some examples that return false aswell. such as >>> 10 < x < 20 False – [ShoeLace](#) Nov 21 '08 at 14:34
- 19 This applies to other comparison operators as well, which is why people are sometimes surprised why code like `(5 in [5])` is `True` is `False` (but it's unpythonic to explicitly test against booleans like that to begin with). – [Miles](#) Mar 2 '09 at 18:35
- 19 Good but watch out for equal precedence, like 'in' and '='. 'A in B == C in D' means '(A in B) and (B == C) and (C in D)' which might be unexpected. – [Charles Merriam](#) Feb 10 '10 at 10:28
- 15 Azafe: Lisp's comparisons naturally work this way. It's not a special case because there's no other (reasonable) way to interpret `(< 1 x 10)`. You can even apply them to single arguments, like `(= 10)`: [cs.cmu.edu/Groups/AI/html/hyperspec/HyperSpec/Body/...](#) – [Ken](#) May 26 '10 at 20:31



Did you find this question interesting? Try our newsletter

Sign up for our newsletter and get our top new questions delivered to your inbox ([see an example](#)).

### Get the python regex parse tree to debug your regex.

Regular expressions are a great feature of python, but debugging them can be a pain, and it's all too easy to get a regex wrong.

Fortunately, python can print the regex parse tree, by passing the undocumented, experimental, hidden flag `re.DEBUG` (actually, 128) to `re.compile`.

```
>>> re.compile("(^([font(?:=(?P<size>[+][0-9]{1,2}))?)?.*)[/font]",
re.DEBUG)
```

```

at at_beginning
literal 91
literal 102
literal 111
literal 110
literal 116
max_repeat 0 1
  subpattern None
    literal 61
    subpattern 1
      in
        literal 45
        literal 43
        max_repeat 1 2
        in
          range (48, 57)
literal 93
subpattern 2
  min_repeat 0 65535
  any None
in
  literal 47
  literal 102
  literal 111
  literal 110
  literal 116

```

Once you understand the syntax, you can spot your errors. There we can see that I forgot to escape the `[]` in `[/font]` .

Of course you can combine it with whatever flags you want, like commented regexes:

```

>>> re.compile("""
^           # start of a line
\[font     # the font tag
(?:=(?P<size> # optional [font+=size]
[~+][0-9]{1,2} # size specification
))?
\[        # end of tag
(.*?)    # text between the tags
\[font\]  # end of the tag
""", re.DEBUG|re.VERBOSE|re.DOTALL)

```

edited Mar 6 '12 at 14:28

community wiki  
6 revs, 4 users 90%  
BatchyX

- 
- 45 Instead of 128 you can also use `re.DEBUG`. Be aware that the comment in the source says this flag is experimental and you shouldn't rely on it. – [Andreas Thomas](#) Dec 28 '08 at 14:24
- 
- 27 If you can use `re.DEBUG`, then you should. It may be experimental, but it's still the symbolic name, and the actual 128 value is just as experimental, but less readable, and more subject to change. – [Lee B](#) Jun 18 '09 at 9:54
- 
- 11 The more idiomatic way to combine flags is using the OR operator, so it should probably be `"re.DEBUG | re.VERBOSE | re.DOTALL"` instead. They're equivalent in this case, but in other cases where you might want to set a flag in addition to a group of flags that *might* already have it, the OR operator is essential. – [rmmh](#) Jan 1 '10 at 1:22
- 
- 3 Except parsing HTML using regular expression is slow and painful. Even the built-in 'html' parser module doesn't use regexes to get the work done. And if the html module doesn't please you, there is plenty of XML/HTML parser modules that does the job without having to reinvent the wheel. – [BatchyX](#) Mar 26 '10 at 21:59
- 
- 1 This should be an official part of Python, not experimental... RegEx is always tricky and being able to trace what's happening is really helpful. – [Cahit](#) Jul 14 '10 at 23:27
- 

## enumerate

Wrap an iterable with `enumerate` and it will yield the item along with its index.

For example:

```

>>> a = ['a', 'b', 'c', 'd', 'e']
>>> for index, item in enumerate(a): print index, item
...
0 a
1 b
2 c
3 d
4 e
>>>


```

References:

- [Python tutorial—looping techniques](#)
- [Python docs—built-in functions—`enumerate`](#)
- [PEP 279](#)

edited Jul 14 '10 at 8:39

community wiki  
4 revs, 3 users 65%  
Dave

- 
- 56 I'm surprised this isn't covered routinely in tutorials talking about python lists. – [Draemon](#) Nov 27 '08 at 4:06
- 
- 45 And all this time I was coding this way: for i in range(len(a)): ... and then using a[i] to get the current item. – [Fernando Martin](#) Jun 22 '09 at 16:35
- 
- 4 @Berry Tsakala: To my knowledge, it has not been deprecated. – [JAB](#) Jul 27 '09 at 17:36
- 
- 23 Holy crap this is awesome. for i in xrange(len(a)): has always been my least favorite python idiom. – [Personman](#) Apr 15 '10 at 16:22
- 
- 15 enumerate can start from arbitrary index, not necessary 0. Example: 'for i, item in enumerate(list, start=1): print i, item' will start enumeration from 1, not 0. – [dmitry\\_romanov](#) Aug 22 '11 at 11:00 
- 

## Creating generators objects

If you write

```
x=(n for n in foo if bar(n))
```

you can get out the generator and assign it to x. Now it means you can do

```
for n in x:
```

The advantage of this is that you don't need intermediate storage, which you would need if you did

```
x = [n for n in foo if bar(n)]
```

In some cases this can lead to significant speed up.

You can append many if statements to the end of the generator, basically replicating nested for loops:

```
>>> n = ((a,b) for a in range(0,2) for b in range(4,6))
>>> for i in n:
...     print i

(0, 4)
(0, 5)
(1, 4)
(1, 5)
```


edited Nov 23 '09 at 14:36

community wiki  
4 revs, 3 users 64%  
freespace

---

You could also use a nested list comprehension for that, yes? – [shapr](#) May 5 '09 at 16:53

---

- 54 Of particular note is the memory overhead savings. Values are computed on-demand, so you never have the entire result of the list comprehension in memory. This is particularly desirable if you later iterate over only part of the list comprehension. – [saffsd](#) May 17 '09 at 2:41
- 
- 19 This is not particularly "hidden" imo, but also worth noting is the fact that you could not rewind a generator object, whereas you can reiterate over a list any number of times. – [susmits](#) Apr 15 '10 at 16:26
- 
- 13 The "no rewind" feature of generators can cause some confusion. Specifically, if you print a generator's contents for debugging, then use it later to process the data, it doesn't work. The data is produced, consumed by print(), then is not available for the normal processing. This doesn't apply to list comprehensions, since they're completely stored in memory. – [johntellsall](#) May 28 '10 at 16:01
- 
- 4 Similar (dup?) answer: [stackoverflow.com/questions/101268/hidden-features-of-python/...](http://stackoverflow.com/questions/101268/hidden-features-of-python/) Note, however, that the answer I linked here mentions a REALLY GOOD presentation about the power of generators. You really should check it out. – [Denilson Sá](#) Aug 4 '10 at 20:21 
- 

## iter() can take a callable argument

For instance:

```
def seek_next_line(f):
```

```
for c in iter(lambda: f.read(1), '\n'):
    pass
```

The `iter(callable, until_value)` function repeatedly calls `callable` and yields its result until `until_value` is returned.

edited May 28 '10 at 15:06

community wiki  
5 revs, 5 users 78%  
mbac32768

This is really cool, I didn't know iter could do that! – [Ryan](#) Sep 29 '08 at 1:13

33 You should also add the explanation: `iter(callable, sentinel)` -> iterator; the callable is called until it returns the sentinel. – [Cristian Ciupitu](#) Oct 5 '08 at 15:23

@Cristian Is this clearer? – [badp](#) May 28 '10 at 15:06

3 To be honest, either the generic description of `iter` from the [Python documentation](#) (`help(iter)`) or an explanation of what's going on here should be used. For example, something like this: *iter(...) creates an iterator that calls f.read(1) until it returns '\n'*. Anyway, since I already know what's going on, others (newbies?) should decide. – [Cristian Ciupitu](#) May 28 '10 at 18:37

As a newbie to python, can you please explain why the `lambda` keyword is necessary here? – [SiegeX](#) Aug 12 '11 at 20:29

## Be careful with mutable default arguments

```
>>> def foo(x=[]):
...     x.append(1)
...     print x
...
>>> foo()
[1]
>>> foo()
[1, 1]
>>> foo()
[1, 1, 1]
```

Instead, you should use a sentinel value denoting "not given" and replace with the mutable you'd like as default:

```
>>> def foo(x=None):
...     if x is None:
...         x = []
...     x.append(1)
...     print x
>>> foo()
[1]
>>> foo()
[1]
```

edited Jul 14 '10 at 2:12

community wiki  
2 revs, 2 users 67%  
Jason Baker

39 That's definitely one of the more nasty hidden features. I've run into it from time to time. – [Torsten Marek](#) Sep 23 '08 at 17:42

77 I found this a lot easier to understand when I learned that the default arguments live in a tuple that's an attribute of the function, e.g. `foo.func_defaults`. Which, being a tuple, is immutable. – [Robert Rossney](#) Nov 5 '09 at 20:43

2 @grayger: As the `def` statement is executed its arguments are evaluated by the interpreter. This creates (or rebinds) a name to a code object (the suite of the function). However, the default arguments are instantiated as objects at the time of definition. This is true of any time of defaulted object, but only significant (exposing visible semantics) when the object is mutable. There's no way of re-binding that default argument name in the function's closure although it can obviously be over-ridden for any call or the whole function can be re-defined). – [Jim Dennis](#) Jun 24 '10 at 4:20

3 @Robert of course the arguments tuple might be immutable, but the objects it point to are not necessarily immutable. – [poolie](#) Jul 22 '10 at 14:59

16 One quick hack to make your initialization a little shorter: `x = x or []`. You can use that instead of the 2 line if statement. – [dave mankoff](#) Aug 27 '10 at 2:58

**Sending values into generator functions.** For example having this function:

```
def mygen():
    """Yield 5 until something else is passed back via send()"""
    a = 5
    while True:
```

```
f = (yield a) #yield a and possibly get f in return
if f is not None:
    a = f #store the new value
```

You can:

```
>>> g = mygen()
>>> g.next()
5
>>> g.next()
5
>>> g.send(7) #we send this back to the generator
7
>>> g.next() #now it will yield 7 until we send something else
7
```

edited Apr 10 '11 at 21:54

community wiki  
3 revs, 3 users 85%  
Rafal Dowgird

89 In other languages, I believe this magical device is called a "variable". – [finnw](#) May 5 '09 at 17:49

5 coroutines should be coroutines and genenerator should be themselves too, without mixing. Mega-great link and talk and examples about this here: [dabeaz.com/coroutines](http://dabeaz.com/coroutines) – [u0b34a0f6ae](#) Aug 23 '09 at 21:39

4 This is a non-hidden feature – [Justin](#) May 28 '10 at 15:26

31 @finnw: the example implements something that's similar to a variable. However, the feature could be used in many other ways ... unlike a variable. It should also be obvious that similar semantics can be implemented using objects (a class implementing Python's `call` method, in particular). – [Jim Dennis](#) Jun 24 '10 at 4:07

4 This is too trivial an example for people who've never seen (and probably won't understand) co-routines. The example that implements the running average without risk of sum variable overflow is a good one. – [Prashant Kumar](#) Jun 11 '11 at 6:16

If you don't like using whitespace to denote scopes, you can use the C-style `{}` by issuing:

```
from __future__ import braces
```

answered Sep 21 '08 at 22:01

community wiki  
[eduffy](#)

122 That's evil. :) – [Jason Baker](#) Sep 22 '08 at 4:32

37 >>> from \_\_future\_\_ import braces File "<stdin>", line 1 SyntaxError: not a chance :P – [Benjamin W. Smith](#) Sep 22 '08 at 19:55

40 that's blasphemy! – [Berk D. Demir](#) Mar 22 '09 at 13:24

335 I think that we may have a syntactical mistake here, shouldn't that be "from `__past__` import braces"? – [Bill K](#) Apr 29 '09 at 21:26

47 from `__cruft__` import braces – [unpluggd](#) Jun 18 '09 at 7:16

The step argument in slice operators. For example:

```
a = [1,2,3,4,5]
>>> a[::2] # iterate over the whole list in 2-increments
[1,3,5]
```

The special case `x[::-1]` is a useful idiom for 'x reversed'.

```
>>> a[::-1]
[5,4,3,2,1]
```

answered Sep 19 '08 at 13:33

community wiki  
[Rafal Dowgird](#)

31 much clearer, in my opinion, is the `reversed()` function. >>> `list(reversed(range(4)))` [3, 2, 1, 0] – [Christian Oudard](#) Jan 2 '09 at 18:35

3 then how to write "this is a string"[::-1] in a better way? `reversed` doesn't seem to help – [Berry Tsakala](#) Jun 21 '09 at 20:56

24 The problem with `reversed()` is that it returns an iterator, so if you want to preserve the type of the reversed sequence (tuple, string, list, unicode, user types...), you need an additional step to convert it back. –

- 6 def reverse\_string(string): return string[::-1] – pi Feb 3 '10 at 9:42
- 4 @pi I think if one knows enough to define reverse\_string as you have then one can leave the[::-1] in your code and be comfortable with its meaning and the feeling it is appropriate. – physicsmichael May 28 '10 at 6:17

## Decorators

**Decorators** allow to wrap a function or method in another function that can add functionality, modify arguments or results, etc. You write decorators one line above the function definition, beginning with an "at" sign (@).

Example shows a `print_args` decorator that prints the decorated function's arguments before calling it:

```
>>> def print_args(function):
>>>     def wrapper(*args, **kwargs):
>>>         print 'Arguments:', args, kwargs
>>>         return function(*args, **kwargs)
>>>     return wrapper

>>> @print_args
>>> def write(text):
>>>     print text

>>> write('foo')
Arguments: ('foo',) {}
foo
```

edited Apr 15 '10 at 16:18

community wiki  
2 revs, 2 users 97%  
DzinX

- 54 When defining decorators, I'd recommend decorating the decorator with @decorator. It creates a decorator that preserves a functions signature when doing introspection on it. More info here: [pyhast.pitt.edu/~micheles/python/documentation.html](http://pyhast.pitt.edu/~micheles/python/documentation.html) – sirwart Sep 22 '08 at 15:53
- 45 How is this a hidden feature? – Vette Oct 2 '08 at 13:52
- 50 Well, it's not present in most simple Python tutorials, and I stumbled upon it a long time after I started using Python. That is what I would call a hidden feature, just about the same as other top posts here. – DzinX Oct 3 '08 at 6:51
- 16 vetler, the questions asks for "lesser-known but useful features of the Python programming language." How do you measure 'lesser-known but useful features'? I mean how are any of these responses hidden features? – Johnd May 23 '09 at 21:14
- 4 @vetler Most of the thing here are hardly "hidden". – Humphrey Bogart Apr 6 '10 at 15:33

The for...else syntax (see <http://docs.python.org/ref/for.html> )

```
for i in foo:
    if i == 0:
        break
else:
    print("i was never 0")
```

The "else" block will be normally executed at the end of the for loop, unless the break is called.

The above code could be emulated as follows:

```
found = False
for i in foo:
    if i == 0:
        found = True
        break
if not found:
    print("i was never 0")
```

edited Apr 10 '11 at 21:57

community wiki  
3 revs, 3 users 67%  
rlerallut

- 218 I think the for/else syntax is awkward. It "feels" as if the else clause should be executed if the body of the loop is never executed. – codeape Oct 10 '08 at 6:44
- 14 ah. Never saw that one! But I must say it is a bit of a misnomer. Who would expect the else block to

execute only if break never does? I agree with codeape: It looks like else is entered for empty foos. – [Daren Thomas](#) Oct 13 '08 at 17:31

52 seems like the keyword should be finally, not else – [Jiaaro](#) Jun 25 '09 at 16:31

21 Except finally is already used in a way where that suite is always executed. – Roger Pate Jun 27 '09 at 22:34

7 Should definately not be 'else'. Maybe 'then' or something, and then 'else' for when the loop was never executed. – [Tor Valamo](#) Jan 10 '10 at 16:26

From 2.5 onwards dicts have a special method `__missing__` that is invoked for missing items:

```
>>> class MyDict(dict):
...     def __missing__(self, key):
...         self[key] = rv = []
...         return rv
...
>>> m = MyDict()
>>> m["foo"].append(1)
>>> m["foo"].append(2)
>>> dict(m)
{'foo': [1, 2]}
```

There is also a dict subclass in `collections` called `defaultdict` that does pretty much the same but calls a function without arguments for not existing items:

```
>>> from collections import defaultdict
>>> m = defaultdict(list)
>>> m["foo"].append(1)
>>> m["foo"].append(2)
>>> dict(m)
{'foo': [1, 2]}
```

I recommend converting such dicts to regular dicts before passing them to functions that don't expect such subclasses. A lot of code uses `d[a_key]` and catches `KeyErrors` to check if an item exists which would add a new item to the dict.

answered Sep 21 '08 at 21:54

community wiki  
[Armin Ronacher](#)

10 I prefer using `setdefault`. `m={} ; m.setdefault('foo',1)` – [grayger](#) Dec 3 '09 at 3:03

22 @grayger meant this `m={}; m.setdefault('foo', []).append(1)` . – [Cristian Ciupitu](#) May 28 '10 at 18:53

1 There are however cases where passing the `defaultdict` is very handy. The function may for example iter over the value and it works for undefined keys without extra code, as the default is an empty list. – [Marian](#) May 29 '10 at 2:22

3 `defaultdict` is better in some circumstances than `setdefault`, since it doesn't create the default object **unless** the key is missing. `setdefault` creates it whether it's missing or not. If your default object is expensive to create this can be a performance hit - I got a decent speedup out of one program simply by changing all `setdefault` calls. – [Whatang](#) Feb 12 '11 at 17:56

2 `defaultdict` is also more powerful than the `setdefault` method in other cases. For example, for a counter— `dd = collections.defaultdict(int) ... dd[k] += 1` vs `d.setdefault(k, 0) += 1`. – [Mike Graham](#) Apr 10 '11 at 21:56

## In-place value swapping

```
>>> a = 10
>>> b = 5
>>> a, b
(10, 5)

>>> a, b = b, a
>>> a, b
(5, 10)
```

The right-hand side of the assignment is an expression that creates a new tuple. The left-hand side of the assignment immediately unpacks that (unreferenced) tuple to the names `a` and `b`.

After the assignment, the new tuple is unreferenced and marked for garbage collection, and the values bound to `a` and `b` have been swapped.

As noted in the [Python tutorial section on data structures](#),

Note that multiple assignment is really just a combination of tuple packing and sequence unpacking.



edited Apr 6 '10 at 1:40

community wiki  
4 revs, 3 users 59%  
Lucas S.

- 
- 1 Does this use more real memory than the traditional way? I would guess so since you are creating a tuple, instead of just one swap variable – [Nathan](#) Jun 14 '10 at 14:24
- 
- 75 It doesn't use more memory. It uses less.. I just wrote it both ways, and de-compiled the bytecode.. the compiler optimizes, as you'd hope it would. dis results showed it's setting up the vars, and then ROT\_TWOing. ROT\_TWO means 'swap the two top-most stack vars'... Pretty snazzy, actually. – [royal](#) Jul 13 '10 at 22:41
- 
- 5 You also inadvertently point out another nice feature of Python, which is that you can implicitly make a tuple of items just by separating them by commas. – [asmeurer](#) Dec 28 '10 at 5:12
- 
- 3 Dana the Sane: assignment in Python is a statement, not an expression, so that expression would be invalid if = had higher priority (i.e. it was interpreted as a, (b = b), a). – [hbn](#) Jan 23 '11 at 21:27
- 
- 5 This is the least hidden feature I've read here. Nice, but explicitly described in every Python tutorial. – [Thiago Chaves](#) May 25 '11 at 16:56
- 

## Readable regular expressions

In Python you can split a regular expression over multiple lines, name your matches and insert comments.

Example verbose syntax (from [Dive into Python](#)):

```
>>> pattern = """
... ^          # beginning of string
... M{0,4}     # thousands - 0 to 4 M's
... (CM|CD|D?C{0,3}) # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3 C's),
...           # or 500-800 (D, followed by 0 to 3 C's)
... (XC|XL|L?X{0,3}) # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 X's),
...           # or 50-80 (L, followed by 0 to 3 X's)
... (IX|IV|V?I{0,3}) # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 I's),
...           # or 5-8 (V, followed by 0 to 3 I's)
... $         # end of string
... """
>>> re.search(pattern, 'M', re.VERBOSE)
```

Example naming matches (from [Regular Expression HOWTO](#))

```
>>> p = re.compile(r'(?P<word>\b\w+\b)')
>>> m = p.search('(((( Lots of punctuation )))')
>>> m.group('word')
'Lots'
```

You can also verbosely write a regex without using `re.VERBOSE` thanks to string literal concatenation.

```
>>> pattern = (
...     """          # beginning of string
...     M{0,4}       # thousands - 0 to 4 M's
...     (CM|CD|D?C{0,3}) # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3 C's),
...                   # or 500-800 (D, followed by 0 to 3 C's)
...     (XC|XL|L?X{0,3}) # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 X's),
...                   # or 50-80 (L, followed by 0 to 3 X's)
...     (IX|IV|V?I{0,3}) # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 I's),
...                   # or 5-8 (V, followed by 0 to 3 I's)
...     "$"          # end of string
... )
>>> print pattern
"^M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$"
```

edited Aug 4 '12 at 1:54

community wiki  
3 revs, 3 users 67%  
user18044

- 
- 7 I don't know if I'd really consider that a Python feature, most RE engines have a verbose option. – [Jeremy Banks](#) Sep 21 '08 at 20:44
- 
- 18 Yes, but because you can't do it in grep or in most editors, a lot of people don't know it's there. The fact that other languages have an equivalent feature doesn't make it not a useful and little known feature of python – [Mark Baker](#) Oct 17 '08 at 9:08
- 
- 7 In a large project with lots of optimized regular expressions (read: optimized for machine but not human beings), I bit the bullet and converted all of them to verbose syntax. Now, introducing new developers to projects is much easier. From now on we enforce verbose REs on every project. – [Berk D. Demir](#) Mar 22 '09 at 13:18

I'd rather just say: hundreds = "(CM|CD|D?C{0,3})" # 900 (CM), 400 (CD), etc. The language already has a way to give things names, a way to add comments, and a way to combine strings. Why use special library

syntax here for things the language already does perfectly well? It seems to go directly against Perlis' Epigram 9. – [Ken](#) May 5 '09 at 18:19

- 3 @Ken: a regex may not always be directly in the source, it could be read from settings or a config file. Allowing comments or just additional whitespace (for readability) can be a great help. – [Roger Pate](#) Jun 27 '09 at 22:30

## Function argument unpacking

You can unpack a list or a dictionary as function arguments using `*` and `**`.

For example:

```
def draw_point(x, y):  
    # do some magic  
  
point_foo = (3, 4)  
point_bar = {'y': 3, 'x': 2}  
  
draw_point(*point_foo)  
draw_point(**point_bar)
```

Very useful shortcut since lists, tuples and dicts are widely used as containers.

edited Jun 23 '10 at 12:13

community wiki  
4 revs, 3 users 74%  
[e-satis](#)

- 27 `*` is also known as the splat operator – [Gabriel](#) Jul 15 '10 at 7:03

- 3 I like this feature, but pylint doesn't sadly. – [Stephen Paulger](#) Mar 25 '11 at 14:46

- 5 pylint's advice is not a law. The other way, `apply(callable, arg_seq, arg_map)`, is deprecated since 2.3. – [Yann Vernier](#) May 4 '11 at 11:50

- 1 pylint's advice may not be law, but it's damn good advice. Debugging code that over-indulges in stuff like this is pure hell. As the original poster notes, this is a useful *shortcut*. – [Andrew](#) Nov 21 '11 at 19:40

- 2 I saw this being used in code once and wondered what it did. Unfortunately it's hard to google for "Python \*\*\*" – [Fraser Graham](#) Jan 13 '12 at 23:33

ROT13 is a valid encoding for source code, when you use the right coding declaration at the top of the code file:

```
#!/usr/bin/env python  
# -*- coding: rot13 -*-  
  
cevag "Uryyb fgnpxbiresbj!".rapbqr("rot13")
```

edited Jun 21 '09 at 20:35

community wiki  
2 revs, 2 users 77%  
[André](#)

- 10 Great! Notice how byte strings are taken literally, but unicode strings are decoded: try `cevag h"Uryyb fgnpxbiresbj!"` – [u0b34a0f6ae](#) Oct 4 '09 at 1:12

- 12 unfortunately it is removed from py3k – [mykhal](#) Dec 19 '09 at 18:33

- 9 This is good for bypassing antivirus. – [Longpoke](#) May 13 '10 at 20:37

- 96 That has nothing to do with the encoding, it is just Python written in Welsh. :-P – [Olivier Verdier](#) Jul 14 '10 at 9:45

- 33 Ph'nglui mglw'nafh Cthulhu R'lyeh wgah'nagl fhtagn! – [Manuel Ferreria](#) Aug 26 '10 at 22:08

## Creating new types in a fully dynamic manner

```
>>> NewType = type("NewType", (object,), {"x": "hello"})  
>>> n = NewType()  
>>> n.x  
"hello"
```

which is exactly the same as

```
>>> class NewType(object):  
>>>     x = "hello"  
>>> n = NewType()
```

```
>>> n.x
"hello"
```

Probably not the most useful thing, but nice to know.

**Edit:** Fixed name of new type, should be `NewType` to be the exact same thing as with `class` statement.

**Edit:** Adjusted the title to more accurately describe the feature.

edited Jan 18 '12 at 16:23

community wiki  
4 revs, 3 users 87%  
Torsten Marek

---

8 This has a lot of potential for usefulness, e.g., JIT ORMs – [Mark Cidade](#) Sep 22 '08 at 18:44

---

8 I use it to generate HTML-Form classes based on a dynamic input. Very nice! – [pi](#) Mar 18 '09 at 16:00

---

15 Note: all classes are created at runtime. So you can use the 'class' statement within a conditional, or within a function (very useful for creating families of classes or classes that act as closures). The improvement that 'type' brings is the ability to neatly define a dynamically generated set of attributes (or bases). – [spookykey](#) Jan 1 '10 at 14:02

---

1 You can also create anonymous types with a blank string like: `type("", (object,), {'x': 'blah'})` – [bluehavana](#) Jun 16 '11 at 23:49

---

3 Could be very useful for code injections. – [Avihu Turzion](#) Jul 18 '11 at 8:49

---

## Context managers and the "with" Statement

Introduced in [PEP 343](#), a [context manager](#) is an object that acts as a run-time context for a suite of statements.

Since the feature makes use of new keywords, it is introduced gradually: it is available in Python 2.5 via the `__future__` directive. Python 2.6 and above (including Python 3) has it available by default.

I have used the ["with" statement](#) a lot because I think it's a very useful construct, here is a quick demo:

```
from __future__ import with_statement

with open('foo.txt', 'w') as f:
    f.write('hello!')
```

What's happening here behind the scenes, is that the ["with" statement](#) calls the special `__enter__` and `__exit__` methods on the file object. Exception details are also passed to `__exit__` if any exception was raised from the with statement body, allowing for exception handling to happen there.

What this does for you in this particular case is that it guarantees that the file is closed when execution falls out of scope of the `with` suite, regardless if that occurs normally or whether an exception was thrown. It is basically a way of abstracting away common exception-handling code.

Other common use cases for this include locking with threads and database transactions.

edited Apr 6 '10 at 2:02

community wiki  
5 revs, 5 users 71%  
Ycros

---

3 I wouldn't approve a code review which imported anything from `future`. The features are more cute than useful, and usually they just end up confusing Python newcomers. – [a paid nerd](#) May 11 '09 at 4:36


---

6 Yes, such "cute" features as nested scopes and generators are better left to those who know what they're doing. And anyone who wants to be compatible with future versions of Python. For nested scopes and generators, "future versions" of Python means 2.2 and 2.5, respectively. For the with statement, "future versions" of Python means 2.6. – [Chris B.](#) Jun 17 '09 at 18:33

---

10 This may go without saying, but with python v2.6+, you no longer need to import from `future`. `with` is now a first class keyword. – [fitzgeraldsteele](#) Nov 23 '09 at 14:28

---

25 In 2.7 you can have multiple `with`s: `with open('filea') as filea and open('fileb') as fileb: ...` – [Austin Richardson](#) Jul 14 '10 at 21:50 

---

5 @Austin i could not get that syntax to work on 2.7. this however did work: `with open('filea') as filea, open('fileb') as fileb: ...` – [wim](#) Jul 14 '11 at 5:29

---

## Dictionaries have a get() method

Dictionaries have a 'get()' method. If you do `d[key]` and key isn't there, you get an exception. If you do `d.get('key')`, you get back `None` if 'key' isn't there. You can add a second argument to get that item back instead of `None`, eg: `d.get('key', 0)`.

It's great for things like adding up numbers:

```
sum[value] = sum.get(value, 0) + 1
```

edited Jan 18 '12 at 16:22

community wiki  
3 revs, 2 users 67%  
Rory

---

39 also, checkout the `setdefault` method. – Daren Thomas Oct 13 '08 at 17:29

27 also, checkout `collections.defaultdict` class. – J.F. Sebastian Nov 23 '08 at 10:35

8 If you are using Python 2.7 or later, or 3.1 or later, check out the `Counter` class in the `collections` module. [docs.python.org/library/collections.html#collections.Counter](https://docs.python.org/library/collections.html#collections.Counter) – Elias Zamaria Oct 12 '10 at 1:33

Oh man, this whole time I've been doing `get(key, None)`. Had no idea that `None` was provided by default. – Jordan Reiter Oct 19 '11 at 16:41

## Descriptors

They're the magic behind a whole bunch of core Python features.

When you use dotted access to look up a member (eg, `x.y`), Python first looks for the member in the instance dictionary. If it's not found, it looks for it in the class dictionary. If it finds it in the class dictionary, and the object implements the descriptor protocol, instead of just returning it, Python executes it. A descriptor is any class that implements the `__get__`, `__set__`, or `__delete__` methods.

Here's how you'd implement your own (read-only) version of property using descriptors:

```
class Property(object):
    def __init__(self, fget):
        self.fget = fget

    def __get__(self, obj, type):
        if obj is None:
            return self
        return self.fget(obj)
```

and you'd use it just like the built-in `property()`:

```
class MyClass(object):
    @Property
    def foo(self):
        return "Foo!"
```

Descriptors are used in Python to implement properties, bound methods, static methods, class methods and slots, amongst other things. Understanding them makes it easy to see why a lot of things that previously looked like Python 'quirks' are the way they are.

Raymond Hettinger has [an excellent tutorial](#) that does a much better job of describing them than I do.

edited Jan 18 '12 at 16:22

community wiki  
7 revs, 7 users 84%  
Nick Johnson

---

This is a duplicate of decorators, isn't it!? ([stackoverflow.com/questions/101268/...](https://stackoverflow.com/questions/101268/)) – gecco Oct 20 '11 at 19:18

2 no, decorators and descriptors are totally different things, though in the example code, i'm creating a descriptor decorator. :) – Nick Johnson Oct 20 '11 at 23:11

1 The other way to do this is with a lambda: `foo = property(lambda self: self.__foo)` – Pete Peterson Nov 2 '11 at 2:50

1 @PetePeterson Yes, but `property` itself is implemented with descriptors, which was the point of my post. – Nick Johnson Nov 2 '11 at 3:40

## Conditional Assignment

```
x = 3 if (y == 1) else 2
```

It does exactly what it sounds like: "assign 3 to x if y is 1, otherwise assign 2 to x". Note that the parens are not necessary, but I like them for readability. You can also chain it if you have something more complicated:

```
x = 3 if (y == 1) else 2 if (y == -1) else 1
```

Though at a certain point, it goes a little too far.

Note that you can use if ... else in any expression. For example:

```
(func1 if y == 1 else func2)(arg1, arg2)
```

Here func1 will be called if y is 1 and func2, otherwise. In both cases the corresponding function will be called with arguments arg1 and arg2.

Analogously, the following is also valid:

```
x = (class1 if y == 1 else class2)(arg1, arg2)
```

where class1 and class2 are two classes.

edited Jan 12 '12 at 23:37

community wiki  
2 revs, 2 users 57%  
tghw

29 The assignment is not the special part. You could just as easily do something like: return 3 if (y == 1) else 2. – [Brian](#) Nov 9 '08 at 5:39

25 That alternate way is the first time I've seen obfuscated Python. – [Craig McQueen](#) Jun 9 '09 at 4:18

3 Kylebrooks: It doesn't in that case, boolean operators short circuit. It will only evaluate 2 if bool(3) == False. – [RoadieRich](#) Jul 12 '09 at 0:23

15 this backwards-style coding confusing me. something like `x = ((y == 1) ? 3 : 2)` makes more sense to me – [mpen](#) Oct 20 '09 at 7:12

13 I feel just the opposite of @Mark, C-style ternary operators have always confused me, is the right side or the middle what gets evaluated on a false condition? I much prefer Python's ternary syntax. – [Jeffrey Harris](#) Dec 3 '09 at 16:51

## Doctest: documentation and unit-testing at the same time.

Example extracted from the Python documentation:

```
def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    If the result is small enough to fit in an int, return an int.
    Else return a long.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(-1)
    Traceback (most recent call last):
      ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact integer:
    """

    import math
    if not n >= 0:
        raise ValueError("n must be >= 0")
    if math.floor(n) != n:
        raise ValueError("n must be exact integer")
    if n+1 == n: # catch a value like 1e300
        raise OverflowError("n too large")
    result = 1
    factor = 2
    while factor <= n:
        result *= factor
        factor += 1
    return result

def _test():
    import doctest
    doctest.testmod()

if __name__ == "__main__":
    _test()
```

edited Jan 18 '12 at 16:18

community wiki  
4 revs, 3 users 90%  
Pierre-Jean Coudert

- 
- 6 Doctests are certainly cool, but I really dislike all the cruft you have to type to test that something should raise an exception – [TM.](#) Mar 10 '09 at 22:55
- 
- 60 Doctests are overrated and pollute the documentation. How often do you test a standalone function without any setUp()? – [a paid nerd](#) May 11 '09 at 4:34
- 
- 2 who says you can't have setup in a doctest? write a function that generates the context and returns locals() then in your doctest do locals().update(setUp()) =D – [Jiaaro](#) Dec 2 '09 at 17:27
- 
- 12 If a standalone function requires a setUp, chances are high that it should be decoupled from some unrelated stuff or put into a class. Class doctest namespace can then be re-used in class method doctests, so it's a bit like setUp, only DRY and readable. – [Andy Mikhaylenko](#) Apr 17 '10 at 14:38
- 
- 4 "How often do you test a standalone function" - lots. I find doctests often emerge naturally from the design process when I am deciding on facades. – [Gregg Lind](#) Feb 11 '11 at 4:45
- 

## Named formatting

% -formatting takes a dictionary (also applies %i/%s etc. validation).

```
>>> print "The %(foo)s is %(bar)i." % {'foo': 'answer', 'bar':42}
The answer is 42.

>>> foo, bar = 'question', 123

>>> print "The %(foo)s is %(bar)i." % locals()
The question is 123.
```

And since locals() is also a dictionary, you can simply pass that as a dict and have % - substitutions from your local variables. I think this is frowned upon, but simplifies things..

## New Style Formatting

```
>>> print("The {foo} is {bar}".format(foo='answer', bar=42))
```

edited Jan 18 '12 at 16:20

community wiki  
5 revs, 5 users 73%  
Pasi Savolainen

- 
- 60 Will be phased out and eventually replaced with string's format() method. – [Constantin](#) Oct 5 '08 at 9:41
- 
- 3 Named formatting is very useful for translators as they tend to just see the format string without the variable names for context – [pixelbeat](#) Oct 12 '08 at 22:45
- 
- 2 Appears to work in python 3.0.1 (needed to add parentheses around print call). – [Pasi Savolainen](#) Jul 1 '09 at 11:32
- 
- 9 a hash, huh? I see where you came from. – [shylent](#) Nov 14 '09 at 13:45
- 
- 11 %s formatting will not be phased out. str.format() is certainly more pythonic, however is actually 10x's slower for simple string replacement. My belief is %s formatting is still best practice. – [Kenneth Reitz](#) Jul 14 '10 at 12:34
- 

To add more python modules (especially 3rd party ones), most people seem to use PYTHONPATH environment variables or they add symlinks or directories in their site-packages directories. Another way, is to use \*.pth files. Here's the official python doc's explanation:

"The most convenient way [to modify python's search path] is to add a path configuration file to a directory that's already on Python's path, usually to the ../site-packages/ directory. Path configuration files have an extension of .pth, and each line must contain a single path that will be appended to sys.path. (Because the new paths are appended to sys.path, modules in the added directories will not override standard modules. This means you can't use this mechanism for installing fixed versions of standard modules.)"

answered Sep 22 '08 at 8:43

community wiki  
dgrant

- 
- 1 I never made the connection between that .pth file in the site-packages directory from setuptools and this idea. awesome. – [dave paola](#) Jul 14 '10 at 2:07
-

Exception **else** clause:

```
try:
    put_4000000000_volts_through_it(parrot)
except Voom:
    print "'E's pining!"
else:
    print "This parrot is no more!"
finally:
    end_sketch()
```

The use of the else clause is better than adding additional code to the try clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the try ... except statement.

See <http://docs.python.org/tut/node10.html>

edited Sep 22 '10 at 5:55

community wiki  
2 revs, 2 users 90%  
Constantin

---

8 +1 this is awesome. If the try block executes without entering any exception blocks, then the else block is entered. And then of course, the finally block is executed – [inspectorG4dget](#) Sep 22 '10 at 4:49

---

I finally get why the 'else' is there! Thanks. – [taynaron](#) Dec 7 '10 at 18:26

---

It would make more sense to use continue, but I guess it's already taken ;) – [Pawel Prazak](#) Dec 16 '10 at 17:58

---

Note that on older versions of Python2 you can't have both else: and finally: clauses for the same try: block – [Kevin Horn](#) May 26 '11 at 22:49

---

1 @Pawel Prazak, as Kevin Horn mentioned, this syntax was introduced after the initial release of Python and adding new reserved keywords to existing language is always problematic. That's why an existing keyword is usually reused (c.f. "auto" in recent C++ standard). – [Constantin](#) Oct 7 '11 at 10:28

---

## Re-raising exceptions:

```
# Python 2 syntax
try:
    some_operation()
except SomeError, e:
    if is_fatal(e):
        raise
    handle_nonfatal(e)

# Python 3 syntax
try:
    some_operation()
except SomeError as e:
    if is_fatal(e):
        raise
    handle_nonfatal(e)
```

The 'raise' statement with no arguments inside an error handler tells Python to re-raise the exception *with the original traceback intact*, allowing you to say "oh, sorry, sorry, I didn't mean to catch that, sorry, sorry."

If you wish to print, store or fiddle with the original traceback, you can get it with `sys.exc_info()`, and printing it like Python would is done with the 'traceback' module.

edited Apr 6 '10 at 2:08

community wiki  
2 revs, 2 users 71%  
Thomas Wouters

---

Sorry but this is a well known and common feature of almost all languages. – [Lucas S.](#) Sep 19 '08 at 14:04

---

I agree with Lucas S. – [Cristian Ciupitu](#) Oct 5 '08 at 15:27

---

6 Note the italicized text. Some people will do `raise e` instead, which doesn't preserve the original traceback. – [habnabit](#) Jan 27 '09 at 3:14

---

12 Maybe more magical, `exc_info = sys.exc_info(); raise exc_info[0], exc_info[1], exc_info[2]` is equivalent to this, but you can change those values around (e.g., change the exception type or message) – [ianb](#) May 5 '09 at 20:27

---

3 @Lucas S. Well, I didn't know it, and I'm glad it's written here. – [e-satis](#) Jun 23 '10 at 12:32

---

Main messages :)

```
import this
# btw look at this module's source :)
```

De-cyphered:

The Zen of Python, by Tim Peters

Beautiful is better than ugly.  
Explicit is better than implicit.  
Simple is better than complex.  
Complex is better than complicated.  
Flat is better than nested.  
Sparse is better than dense.  
Readability counts.  
Special cases aren't special enough to break the rules.  
Although practicality beats purity.  
Errors should never pass silently.  
Unless explicitly silenced.  
In the face of ambiguity, refuse the temptation to guess. There should be one-- and preferably only one --obvious way to do it.  
Although that way may not be obvious at first unless you're Dutch.  
Now is better than never.  
Although never is often better than *right* now.  
If the implementation is hard to explain, it's a bad idea.  
If the implementation is easy to explain, it may be a good idea.  
Namespaces are one honking great idea -- let's do more of those!

edited Oct 20 '09 at 7:08

community wiki  
2 revs, 2 users 83%  
Mark

1 Any idea why the source was cyphered that way? Was it just for fun, or was there some other reason? – MiniQuark Dec 14 '08 at 20:32

42 the way the source is written goes against the zen! – hasen Jan 1 '09 at 5:39

8 [svn.python.org/view/python/trunk/Lib/this.py?view=markup](http://svn.python.org/view/python/trunk/Lib/this.py?view=markup) – erikprice Jun 24 '09 at 19:41

2 I've updated my /usr/lib/python2.6/this.py replacing the old code with this `print s.translate("".join(chr(64<i<91 and 65+(i-52)%26 or 96<i<123 and 97+(i-84)%26 or i) for i in range(256)))` and it looks much better now!! :-D – fortran Jun 25 '09 at 13:12

2 @MiniQuark: quick history lesson: [wefearchange.org/2010/06/import-this-and-zen-of-python.html](http://wefearchange.org/2010/06/import-this-and-zen-of-python.html) – Dan Jul 14 '10 at 0:51

## Interactive Interpreter Tab Completion

```
try:
    import readline
except ImportError:
    print "Unable to load readline module."
else:
    import rlcompleter
    readline.parse_and_bind("tab: complete")

>>> class myclass:
...     def function(self):
...         print "my function"
...
>>> class_instance = myclass()
>>> class_instance.<TAB>
class_instance.__class__      class_instance.__module__
class_instance.__doc__       class_instance.function
>>> class_instance.f<TAB>unction()
```

You will also have to set a PYTHONSTARTUP environment variable.

answered Oct 3 '08 at 18:38

community wiki  
mjard

2 This is a very useful feature. So much so I've a simple script to enable it (plus a couple of other introspection enhancements): [pixelbeat.org/scripts/inpy](http://pixelbeat.org/scripts/inpy) – pixelbeat Oct 12 '08 at 22:49

43 IPython gives you this plus tons of other neat stuff – akaihola Jan 10 '09 at 3:47

4 @akaihola read the main qn. – Sriram Nov 3 '09 at 17:43



- 2 @haridsv -- easy\_install ipdb -- then you can use `import ipdb; ipdb.set_trace()` – Doug Harris May 20 '10 at 22:03
- 
- 1 On osx [and i imagine other systems which use libedit] you have to do `readline.parse_and_bind ("bind ^I rl_complete")` – Foo Bah Feb 10 '11 at 15:20
- 

Nested list comprehensions and generator expressions:

```
[(i,j) for i in range(3) for j in range(i) ]  
((i,j) for i in range(4) for j in range(i) )
```

These can replace huge chunks of nested-loop code.

answered Sep 19 '08 at 12:45 community wiki  
Rafal Dowgird

- "for j in range(i)" - is this a typo? Normally you'd want fixed ranges for i and j. If you're accessing a 2d array, you'd miss out on half your elements. – Peter Gibson Mar 29 '09 at 2:27
- I'm not accessing any arrays in this example. The only purpose of this code is to show that the expressions from the inner ranges can access those from the outer ones. The by-product is a list of pairs (x,y) such that  $4 > x > y > 0$ . – Rafal Dowgird Mar 30 '09 at 8:23
- 2 sorta like double integration in calculus, or double summation. – Yoo Sep 29 '09 at 20:57
- 22 Key point to remember here (which took me a long time to realize) is that the order of the `for` statements are to be written in the order you'd expect them to be written in a standard for-loop, from the outside inwards. – sykora Jan 26 '10 at 11:17
- 2 To add on to sykora's comment: imagine you're starting with a stack of `for` s and `if` s with `yield x` inside. To convert that to a generator expression, move `x` first, delete all the colons (and the `yield` ), and surround the whole thing in parentheses. To make a list comprehension instead, replace the outer parens with square brackets. – Ken Arnold Jun 14 '11 at 1:30
- 

Operator overloading for the `set` builtin:

```
>>> a = set([1,2,3,4])  
>>> b = set([3,4,5,6])  
>>> a | b # Union  
{1, 2, 3, 4, 5, 6}  
>>> a & b # Intersection  
{3, 4}  
>>> a < b # Subset  
False  
>>> a - b # Difference  
{1, 2}  
>>> a ^ b # Symmetric Difference  
{1, 2, 5, 6}
```

More detail from the standard library reference: [Set Types](#)

edited Feb 3 '12 at 18:46 community wiki  
2 revs, 2 users 97%  
Kiv

In the tutorial, partly [docs.python.org/tutorial/datastructures.html#sets](https://docs.python.org/tutorial/datastructures.html#sets) – XTL Feb 16 '12 at 7:58

---