

[Home Page](#)[Research](#)[Writing...](#)[Python](#)[Instant Hacking](#)[Instant Python](#)[Praise](#)[Contact...](#)[Miscellaneous...](#)[Printable version](#)

# Instant Python

[If you like this tutorial, please check out my book [Beginning Python](#).]

This is a *minimal* crash-course in the programming language [Python](#). To learn more, take a look at the documentation at the Python web site, [www.python.org](http://www.python.org); especially the [tutorial](#). If you wonder why you should be interested, check out the [comparison](#) page where Python is compared to other languages.

This introduction has been translated into several languages, among them [Portuguese](#), [Italian](#), [Spanish](#), [Russian](#), [French](#), [Lithuanian](#), [Japanese](#), [German](#) and [Greek](#), and is currently being translated into Norwegian, Polish, and Korean. Since this document still might undergo changes, these translations may not always be up to date.

**Note:** To get the examples working properly, write the programs in a text file and then run that with the interpreter; do *not* try to run them directly in the interactive interpreter - not all of them will work. (Please do *not* ask me for details on this! I get swamped with emails on the subject... Check the [documentation](#), or send an email to [help@python.org](mailto:help@python.org)).

## The Basics

To begin with, think of Python as pseudo-code. It's almost true. Variables don't have types, so you don't have to declare them. They appear when you assign to them, and disappear when you don't use them anymore. Assignment is done by the `=` operator. Equality is tested by the `==` operator. You can assign several variables at once:

```
x,y,z = 1,2,3
first, second = second, first
a = b = 123
```

Blocks are indicated through indentation, and *only* through indentation. (No `BEGIN/END` or braces.) Some common control structures are:

```
if x < 5 or (x > 10 and x < 20):
    print "The value is OK."

if x < 5 or 10 < x < 20:
    print "The value is OK."

for i in [1,2,3,4,5]:
    print "This is iteration number", i

x = 10
while x >= 0:
    print "x is still not negative."
    x = x-1
```

The first two examples are equivalent.

The index variable given in the `for` loop iterates through the elements of a *list* (written as in the example). To make an "ordinary" `for` loop (that is, a counting loop), use the built-in function `range()`.

```
# Print out the values from 0 to 99 inclusive.
for value in range(100):
    print value
```

(The line beginning with `"#"` is a comment, and is ignored by the interpreter.)

Okay; now you know enough to (in theory) implement any algorithm in Python. Let's add some *basic* user interaction. To get input from the user (from a text prompt), use the builtin function

```
input.
```

```
x = input("Please enter a number: ")
print "The square of that number is", x*x
```

The `input` function displays the prompt given (which may be empty) and lets the user enter any valid Python value. In this case we were expecting a number – if something else (like a string) is entered, the program would crash. To avoid that we would need some error checking. I won't go into that here; suffice it to say that if you want the user input stored *verbatim* as a string (so that *anything* can be entered), use the function `raw_input` instead. If you wanted to convert the input string `s` to an integer, you could then use `int(s)`.

**Note:** If you want to input a string with `input`, the user has to write the quotes explicitly. In Python, strings can be enclosed in either single or double quotes.

So, we have control structures, input and output covered – now we need some snazzy data structures. The most important ones are *lists* and *dictionaries*. Lists are written with brackets, and can (naturally) be nested:

```
name = ["Cleese", "John"]

x = [[1,2,3],[y,z],[[]]]
```

One of the nice things about lists is that you can access their elements separately or in groups, through *indexing* and *slicing*. Indexing is done (as in many other languages) by appending the index in brackets to the list. (Note that the first element has index 0).

```
print name[1], name[0] # Prints "John Cleese"

name[0] = "Smith"
```

Slicing is almost like indexing, except that you indicate both the start and stop index of the result, with a colon (":") separating them:

```
x = ["spam","spam","spam","spam","spam","eggs","and","spam"]

print x[5:7] # Prints the list ["eggs","and"]
```

Notice that the end is non-inclusive. If one of the indices is dropped, it is assumed that you want everything in that direction. I.e. `list[:3]` means "every element from the beginning of `list` up to element 3, non-inclusive." (It could be argued that it would actually mean element 4, since the counting starts at 0... Oh, well.) `list[3:]` would, on the other hand, mean "every element from `list`, starting at element 3 (inclusive) up to, and including, the last one." For really interesting results, you can use negative numbers too: `list[-3]` is the third element from the end of the list...

While on the subject of indexing, you might be interested to know that the built-in function `len` gives you the length of a list.

Now, then – what about dictionaries? To put it simply, they are like lists, only their contents are not ordered. How do you index them then? Well, every element has a *key*, or a "name" which is used to look up the element just like in a real dictionary. A couple of example dictionaries:

```
{ "Alice" : 23452532, "Boris" : 252336,
  "Clarice" : 2352525, "Doris" : 23624643}

person = { 'first name': "Robin", 'last name': "Hood",
           'occupation': "Scoundrel" }
```

Now, to get `person`'s occupation, we use the expression `person["occupation"]`. If we wanted to change his last name, we could write:

```
person['last name'] = "of Locksley"
```

Simple, isn't it? Like lists, dictionaries can hold other dictionaries. Or lists, for that matter. And naturally lists can hold dictionaries too. That way, you can easily make some quite advanced data structures.

## Functions

Next step: Abstraction. We want to give a name to a piece of code, and call it with a couple of

parameters. In other words – we want to define a function (or "procedure"). That's easy. Use the keyword `def` like this:

```
def square(x):
    return x*x

print square(2) # Prints out 4
```

For those of you who understand it: When you pass a parameter to a function, you bind the parameter to the value, thus creating a new reference. If you change the "contents" of this parameter name (i.e. rebind it) that won't affect the original. This works just like in Java, for instance. Let's take a look at an example:

```
def change(some_list):
    some_list[1] = 4

x = [1,2,3]
change(x)
print x # Prints out [1,4,3]
```

As you can see, it is the original list that is passed in, and if the function changes it, these changes carry over to the place where the function was called. Note, however the behaviour in the following example:

```
def nochange(x):
    x = 0

y = 1
nochange(y)
print y # Prints out 1
```

Why is there no change now? Because we *don't change the value!* The value that is passed in is the number 1 — we can't change a number in the same way that we change a list. The number 1 is (and will always be) the number 1. What we *did* do is change the contents of the local variable (parameter) `x`, and this does *not* carry over to the environment.

For those of you who didn't understand this: Don't worry — it's pretty easy if you don't think too much about it :)

Python has all kinds of nifty things like *named arguments* and *default arguments* and can handle a variable number of arguments to a single function. For more info on this, see the Python tutorial's [section 4.7](#).

If you know how to use functions in general, this is basically what you need to know about them in Python. (Oh, yes... The `return` keyword halts the execution of the function and returns the value given.)

One thing that might be useful to know, however, is that functions are *values* in Python. So if you have a function like `square`, you could do something like:

```
queueble = square
print queueble(2) # Prints out 4
```

To call a function without arguments you must remember to write `doit()` and not `doit`. The latter, as shown, only returns the function itself, as a value. (This goes for methods in objects too... See below.)

## Objects and Stuff...

I assume you know how object-oriented programming works. (Otherwise, this section might not make much sense. No problem... Start playing without the objects :).) In Python you define classes with the (surprise!) `class` keyword, like this:

```
class Basket:

    # Always remember the *self* argument
    def __init__(self, contents=None):
        self.contents = contents or []

    def add(self, element):
        self.contents.append(element)

    def print_me(self):
```

```

result = ""
for element in self.contents:
    result = result + " " + `element`
print "Contains:"+result

```

New things here:

1. All methods (functions in an object) receive an additional argument at the start of the argument list, containing the object itself. (Called `self` in this example, which is customary.)
2. Methods are called like this: `object.method(arg1,arg2)`.
3. Some method names, like `__init__` (with two underscores before and after), are pre-defined, and mean special things. `__init__` is the name of the *constructor* of the class, i.e. it is the function that is called when you create an instance.
4. Some arguments can be *optional* and given a default value (as mentioned before, under the section on functions). This is done by writing the definition like:

```
def spam(age=32): ...
```

Here, `spam` can be called with one or zero parameters. If none is used, then the parameter `age` will have the value 32.

5. "Short circuit logic." This is a clever bit... See below.
6. Backquotes convert an object to its string representation. (So if `element` contains the number 1, then ``element`` is the same as "1" whereas `'element'` is a literal string.)
7. The addition sign `+` is used also for concatenating lists, and strings are really just lists of characters (which means that you can use indexing and slicing and the `len` function on them. Cool, huh?)

No methods or member variables are protected (or private or the like) in Python. Encapsulation is pretty much a matter of programming style. (If you *really* need it, there are naming-conventions that will allow some privacy :)).

Now, about that short-circuit logic...

All values in Python can be used as logic values. Some of the more "empty" ones, like `[]`, `0`, `"` and `None` represent logical falsity, while most other values (like `[0]`, `1` or `"Hello, world"`) represent logical truth.

Now, logical expressions like `a and b` are evaluated like this: First, check if `a` is true. If it is *not*, then simply return it. If it *is*, then simply return `b` (which will represent the truth value of the expression.) The corresponding logic for `a or b` is: If `a` is true, then return it. If it isn't, then return `b`.

This mechanism makes `and` and `or` behave like the boolean operators they are supposed to implement, but they also let you write short and sweet little conditional expressions. For instance, the statement

```

if a:
    print a
else:
    print b

```

Could instead be written:

```
print a or b
```

Actually, this is somewhat of a Python idiom, so you might as well get used to it. This is what we do in the method `Basket.__init__`. The argument `contents` has a default value of `None` (which is, among other things, false). Therefore, to check if it had a value, we could write:

```

if contents:
    self.contents = contents
else:
    self.contents = []

```

Of course, now you know there is a better way. And why don't we give it the default value of `[]` in the first place? Because of the way Python works, this would give all the Baskets the same empty list as default contents. As soon as one of them started to fill up, they all would contain the same elements, and the default would not be empty anymore... To learn more about this, you should read the documentation and look for the difference between *identity* and *equality*.

Another way of doing the above is:

```
def __init__(self, contents=[]):
    self.contents = contents[:]
```

Can you guess how this works? Instead of using the same empty list everywhere, we use the expression `contents[:]` to make a copy. (We simply slice the entire thing.)

So, to actually make a `Basket` and to use it (i.e. to call some methods on it) we would do something like this:

```
b = Basket(['apple', 'orange'])
b.add("lemon")
b.print_me()
```

There are other magic methods than `__init__`. One such method is `__str__` which defines how the object wants to look if it is treated like a string. We could use this in our basket instead of `print_me`:

```
def __str__(self):
    result = ""
    for element in self.contents:
        result = result + " " + `element`
    return "Contains:" + result
```

Now, if we wanted to print the basket `b`, we could just use:

```
print b
```

Cool, huh?

Subclassing is done like this:

```
class SpamBasket(Basket):
    # ...
```

Python allows multiple inheritance, so you can have several superclasses in the parentheses, separated by commas. Classes are instantiated like this: `x = Basket()`. Constructors are, as I said, made by defining the special member function `__init__`. Let's say that `SpamBasket` had a constructor `__init__(self, type)`. Then you could make a spam basket thus: `y = SpamBasket("apples")`.

If you, in the constructor of `SpamBasket`, needed to call the constructor of one or more superclasses, you could call it like this: `Basket.__init__(self)`. Note, that in addition to supplying the ordinary parameters, you have to explicitly supply `self`, since the superclass `__init__` doesn't know which instance it is dealing with.

For more about the wonders of object-oriented programming in Python, see the tutorial's [section 9](#).

## A Jedi Mind Trick

(This section is only here because I think it is cool. It is definitely *not* necessary to read it to start learning Python. See the end of the section for a note about changes for Python 2.1.)

Do you like mind boggling concepts? Then, if you're really daring, you might want to check out Guido van Rossum's essay on [metaclasses](#). If, however, you would rather *not* have your brain explode, you might be satisfied with this little trick.

Python uses dynamic as opposed to lexical namespaces. That means that if you have a function like this:

```
def orange_juice():
    return x*2
```

... where a variable (in this case `x`) is not bound to an argument and is not given a value inside the function, Python will use the value it has where and when the function is called. In this case:

```
x = 3
y = orange_juice()
# y is now 6
x = 1
```

```
y = orange_juice()
# y is now 2
```

Usually, this is the sort of behaviour you want (though the example above is a bit contrived – you seldom access variables like this.) *However*, sometimes it can be nice to have something like a static namespace, that is, storing some values from the environment in the function when it is created. The way of doing this in Python is by means of default arguments.

```
x = 4
def apple_juice(x=x):
    return x*2
```

Here, the argument `x` is given a default value which is the same as the *value* of the variable `x` at the time when the function is defined. So, as long as nobody supplies an argument for the function, it will work like this:

```
x = 3
y = apple_juice()
# y is now 8
x = 1
y = apple_juice()
# y is now 8
```

So – the value of `x` is not changed. If this was all we wanted, we might just as well have written

```
def tomato_juice():
    x = 4
    return x*2
```

or even

```
def carrot_juice():
    return 8
```

However, the *point* is that the value of `x` is picked up from the *environment* at the time when the function is defined. How is this useful? Let's take an example – a function which composes two other functions.

We want a function that works like this:

```
from math import sin, cos
sincos = compose(sin,cos)
x = sincos(3)
```

Where `compose` is the function we want to make, and `x` has the value `-0.836021861538`, which is the same as `sin(cos(3))`. Now, how do we do this?

(Note that we are using functions as arguments here... That's a pretty neat trick in itself.)

Clearly, `compose` takes two functions as parameters, and returns a function which again takes one parameter. So, a skeleton solution might be:

```
def compose(fun1, fun2):
    def inner(x):
        pass # ...
    return inner
```

We might be tempted to put `return fun1(fun2(x))` inside the function `inner` and leave it at that. No, no, no. That would behave very strangely. Imagine the following scenario:

```
from math import sin, cos

# Wrong version
def compose(fun1, fun2):
    def inner(x):
        return fun1(fun2(x))
    return inner

def fun1(x):
    return x + " world!"

def fun2(x):
    return "Hello,"

sincos = compose(sin,cos) # Using the wrong version

x = sincos(3)
```

Now, what value would `x` have? That's right: `"Hello, world"`. Why is that? Because when it's called, it picks up the values of `fun1` and `fun2` from the environment, not the ones that were around when it was created. To get a working solution, all we have to do is use the technique I described earlier:

```
def compose(fun1, fun2):
    def inner(x, fun1=fun1, fun2=fun2):
        return fun1(fun2(x))
    return inner
```

Now we only have to hope that nobody supplies the resulting function with more than one argument, as that would wreck it :). And by the way, since we have no need for the name `inner`, and it contains only an expression, we might as well use an *anonymous* function, using the keyword `lambda`:

```
def compose(f1, f2):
    return lambda x, f1=f1, f2=f2: f1(f2(x))
```

Terse, yet clear. You've gotta love it :)

(And if you didn't understand any of that, don't worry. At least I hope it has convinced you that Python is more than "just a scripting language"... :))

## A Note About Python 2.1 and Nested Scopes

With the advent of Python 2.1, the language now (optionally) has statically nested scopes or namespaces. That means that you can do the things described in this section without some of the trickiness. Now you can simply write the following:

```
# This magic won't be necessary in Python 2.2:
from __future__ import nested_scopes

def compose(fun1, fun2):
    def inner(x):
        return fun1(fun2(x))
    return inner
```

... and it will work as it should.

## And Now...

Just a few things near the end. Most useful functions and classes are put in *modules*, which are really text-files containing Python code. You can import these and use them in your own programs. For instance, to use the method `split` from the standard module `string`, you can do either:

```
import string

x = string.split(y)
```

Or...

```
from string import split

x = split(y)
```

**Note:** The `string` module isn't used much anymore; rather than the code below, you *should* use `x=y.split()`.

For more information on the standard library modules, take a look at [www.python.org/doc/lib](http://www.python.org/doc/lib). They contain a lot of useful stuff.

All the code in the module/script is run when it is imported. If you want your program to be both an importable module and a runnable program, you might want to add something like this at the end of it:

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

This is a magic way of saying that if this module is run as an executable script (that is, it is not being imported into another script), then the function `go` should be called. Of course, you could do

anything after the colon there... :)

And for those of you who want to make an executable script on UN\*X, use the following first line to make it run by itself:

```
#!/usr/bin/env python
```

Finally, a brief mention of an important concept: Exceptions. Some operations (like dividing something by zero or reading from a non-existent file) produce an error condition or *exception*. You can even make your own and raise them at the appropriate times.

If nothing is done about the exception, your program ends and prints out an error message. You can avoid this with a `try/except`-statement. For instance:

```
def safe_division(a,b):
    try:
        return a/b
    except ZeroDivisionError:
        return None
```

`ZeroDivisionError` is a standard exception. In this case, you *could* have checked if `b` was zero, but in many cases, that strategy is not feasible. And besides, if we didn't have the `try`-clause in `safe_division`, thereby making it a risky function to call, we could still do something like:

```
try:
    unsafe_division(a,b)
except ZeroDivisionError:
    print "Something was divided by zero in unsafe_division"
```

In cases where you *normally* would not have a specific problem, but it *might* occur, using exceptions lets you avoid costly testing etc.

Well – that's it. Hope you learned something. Now go and [play](#). And remember the Python motto of learning: "Use the source, Luke." (Translation: Read all the code you can get your hands on :)) To get you started, here is an [example](#). It is Hoare's well known *QuickSort* algorithm. A syntax-coloured version is [here](#).

One thing might be worth mentioning about this example. The `done` variable controls whether or not the `partition` has finished moving about the elements. So when one of the two inner loops wants to end the entire swapping sequence, it sets `done` to 1 and then exits with `break`. Why do the inner loops use `done`? Because, when the first inner loop ends with a `break`, whether or not the next loop is to be started depends on whether the main loop is finished, that is, whether or not `done` has been set to 1:

```
while not done:
    while not done:
        # Iterates until a break

    while not done:
        # Only executed if the first didn't set "done" to 1
```

An equivalent, possibly clearer, but in my opinion less pretty version would be:

```
while not done:
    while 1:
        # Iterates until a break

    if not done:
        while 1:
            # Only executed if the first didn't set "done" to 1
```

The only reason I used the `done` variable in the first loop was that I liked preserving the symmetry between the two. That way one could reverse their order and the algorithm would still work.

Some more examples can be found on Joe Strout's [tidbit](#) page.