

# Instant Hacking

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

This is a *short* introduction to the art of programming, with examples written in the programming language [Python](#). (If you already know how to program, but want a short intro to Python, you may want to check out my article [Instant Python](#).) This article has been translated into [Italian](#), [Polish](#), [Japanese](#), [Serbian](#), [Brazilian Portuguese](#), and [Dutch](#), and is in the process of being translated into Korean.

This page is not about breaking into other people's computer systems etc. I'm not into that sort of thing, so please *don't* email me about it.

*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 don't ask me on details on this. Check the [documentation](#) or send an email to [help@python.org](mailto:help@python.org)).

## The Environment

To program in Python, you must have an interpreter installed. It exists for most platforms (including Macintosh, Unix and Windows). More information about this can be found on the [Python web site](#). You also should have a text editor (like *emacs*, *notepad* or something similar).

## What is Programming?

Programming a computer means giving it a set of instructions telling it what to do. A computer program in many ways resembles recipes, like the ones we use for cooking. For example [1]:

Fiesta SPAM Salad

Ingredients:

Marinade:

1/4 cup lime juice  
1/4 cup low-sodium soy sauce  
1/4 cup water  
1 tablespoon vegetable oil  
3/4 teaspoon cumin  
1/2 teaspoon oregano  
1/4 teaspoon hot pepper sauce  
2 cloves garlic, minced

Salad:

1 (12-ounce) can SPAM Less Sodium luncheon meat,  
    cut into strips  
1 onion, sliced  
1 bell pepper, cut in strips  
Lettuce  
12 cherry tomatoes, halved

**Instructions:**

In jar with tight-fitting lid, combine all marinade ingredients; shake well. Place SPAM strips in plastic bag. Pour marinade over SPAM. Seal bag; marinate 30 minutes in refrigerator. Remove SPAM from bag; reserve 2 tablespoons marinade. Heat reserved marinade in large skillet. Add SPAM, onion, and green pepper. Cook 3 to 4 minutes or until SPAM is heated. Line 4 individual salad plates with lettuce. Spoon hot salad mixture over lettuce. Garnish with tomato halves. Serves 4.

Of course, no computer would understand this... And most computers wouldn't be able to make a salad even if they *did* understand the recipe. So what do we have to do to make this more computer-friendly? Well — basically two things. We have to (1) talk in a way that the computer can understand, and (2) talk about things that it can do something with.

The first point means that we have to use a language — a *programming language* that we have an interpreter program for, and the second point means that we can't expect the computer to make a salad — but we *can* expect it to add numbers, write things to the screen etc.

## Hello...

There's a tradition in programming tutorials to always begin with a program that prints "Hello, world!" to the screen. In Python, this is quite simple:

```
print "Hello, world!"
```

This is basically like the recipe above (although it is much shorter!). It tells the computer what to do: To print "Hello, world!". Piece of cake. What if we would want it to do more stuff?

```
print "Hello, world!"
print "Goodbye, world!"
```

Not much harder, was it? And not really very interesting... We want to be able to *do something* with the *ingredients*, just like in the spam salad. Well — what ingredients do we have? For one thing, we have strings of text, like "Hello, world!", but we also have *numbers*. Say we wanted the computer to calculate the area of a rectangle for us. Then we could give it the following little recipe:

```
# The Area of a Rectangle

# Ingredients:

width = 20
height = 30

# Instructions:

area = width*height
print area
```

You can probably see the similarity (albeit slight) to the spam salad recipe. But how does it work? First of all, the lines beginning with # are called comments and are actually ignored by the computer. However, inserting small explanations like this can be important in making your programs more readable to humans.

Now, the lines that look like `foo = bar` are called *assignments*. In the case of `width = 20` we tell the computer that the width should be 20 from this point on. What does it mean that “the width is 20”? It means that a *variable* by the name “width” is created (or if it already exists, it is reused) and given the value 20. So, when we use the variable later, the computer knows its value. Thus,

```
width*height
```

is essentially the same as

```
20*30
```

which is calculated to be 600, which is then assigned to the variable by the name “area”. The final statement of the program prints out the value of the variable “area”, so what you see when you run this program is simply

```
600
```

*Note:* In some languages you have to tell the computer which variables you need at the beginning of the program (like the ingredients of the salad) — Python is smart enough to figure this out as it goes along.

## Feedback

OK. Now you can perform simple, and even quite advanced calculations. For instance, you might want to make a program to calculate the area of a circle instead of a rectangle:

```
radius = 30
```

```
print radius*radius*3.14
```

However, this is not significantly more interesting than the rectangle program. At least not in my opinion. It is somewhat inflexible. What if the circle we were looking at had a radius of 31? How would the computer know? It’s a bit like the part of the salad recipe that says: “Cook 3 to 4 minutes or until SPAM is heated.” To know when it is cooked, we have to *check*. We need *feedback*, or *input*. How does the computer know the radius of our circle? It too needs input... What we can do is to tell it to check the radius:

```
radius = input("What is the radius?")
```

```
print radius*radius*3.14
```

Now things are getting snazzy... `input` is something called a *function*. (You’ll learn to create your own in a while. `input` is a function that is built into the Python language.) Simply writing

```
input
```

won’t do much... You have to put a pair of parentheses at the end of it. So `input()` would work — it would simply wait for the user to enter the radius. The version above is perhaps a bit more user-friendly, though, since it prints out a question first. When we put something like the question-string “What is the radius?” between the parentheses of a *function call* it is called *passing a parameter* to the function. The thing (or things) in the parentheses is (or are) the *parameter(s)*. In this case we pass a question as a parameter so that `input` knows what to print out before getting the answer from the user.

But how does the answer get to the `radius` variable? The function `input`, when called, *returns a value*

(like many other functions). You don't *have* to use this value, but in our case, we want to. So, the following two statements have very different meanings:

```
foo = input
bar = input()
```

`foo` now contains the input function *itself* (so it can actually be used like `foo("What is your age?")`; this is called a *dynamic function call*) while `bar` contains whatever is typed in by the user.

## Flow

Now we can write programs that perform simple actions (arithmetic and printing) and that can receive input from the user. This is useful, but we are still limited to so-called *sequential execution* of the commands, that is — they have to be executed in a fixed order. Most of the spam salad recipe is sequential or linear like that. But what if we wanted to tell the computer how to check on the cooked spam? If it is heated, then it should be removed from the oven — otherwise, it should be cooked for another minute or so. How do we express that?

What we want to do, is to control the *flow* of the program. It can go in two directions — either take out the spam, or leave it in the oven. We can choose, and the *condition* is whether or not it is properly heated. This is called *conditional execution*. We can do it like this:

```
temperature = input("What is the temperature of the spam?")

if temperature > 50:
    print "The salad is properly cooked."
else:
    print "Cook the salad some more."
```

The meaning of this should be obvious: If the temperature is higher than 50 (centigrades), then print out a message telling the user that it is properly cooked, otherwise, tell the user to cook the salad some more.

*Note:* The indentation is important in Python. *Blocks* in conditional execution (and *loops* and *function definitions* — see below) must be indented (and indented by the same amount of whitespace; a tab counts as 8 spaces) so that the interpreter can tell where they begin and end. It also makes the program more readable to humans.

Let's return to our area calculations. Can you see what this program does?

```
# Area calculation program

print "Welcome to the Area calculation program"
print "_____ "
print

# Print out the menu:
print "Please select a shape:"
print "1  Rectangle"
print "2  Circle"

# Get the user's choice:
shape = input("> ")

# Calculate the area:
```

```

if shape == 1:
    height = input("Please enter the height: ")
    width = input("Please enter the width: ")
    area = height*width
    print "The area is", area
else:
    radius = input("Please enter the radius: ")
    area = 3.14*(radius**2)
    print "The area is", area

```

*New things* in this example:

1. `print` used all by itself prints out an empty line
2. `==` checks whether two things are equal, as opposed to `=`, which assigns the value on the right side to the variable on the left. This is an *important distinction!*
3. `**` is Python's *power* operator — thus the squared radius is written `radius**2`.
4. `print` can print out more than one thing. Just separate them with commas. (They will be separated by single spaces in the output.)

The program is quite simple: It asks for a number, which tells it whether the user wants to calculate the area of a rectangle or a circle. Then, it uses an `if`-statement (conditional execution) to decide which block it should use for the area calculation. These two blocks are essentially the same as those used in the previous area examples. Notice how the comments make the code more readable. It has been said that the first commandment of programming is: "Thou shalt comment!" Anyway — it's a nice habit to acquire.

## Exercise 1

Extend the program above to include area calculations on squares, where the user only has to enter the length of one side. There is one thing you need to know to do this: If you have more than two choices, you can write something like:

```

if foo == 1:
    # Do something...
elif foo == 2:
    # Do something else...
elif foo == 3:
    # Do something completely different...
else:
    # If all else fails...

```

Here `elif` is a mysterious code which means "else if" :). So; if `foo` is one, then do something; otherwise, if `foo` is two, then do something else, etc. You might want to add other options to the programs too — like triangles or arbitrary polygons. It's up to you.

## Loops

Sequential execution and conditionals are only *two* of the *three* fundamental building blocks of programming. The third is the *loop*. In the previous section I proposed a solution for checking if the spam was heated, but it was quite clearly inadequate. What if the spam wasn't finished the next time we checked either? How could we know how many times we needed to check it? The truth is, we couldn't. And we shouldn't have to. We should be able to ask the computer to keep checking until it was done. How do we do that? You guessed it — we use a loop, or *repeated execution*.

Python has two loop types: *while*-loops and *for*-loops. *For*-loops are perhaps the simplest. For instance:

```
for food in "spam", "eggs", "tomatoes":
    print "I love", food
```

This means: For every element in the list "spam", "eggs", "tomatoes", print that you love it. The block inside the loop is executed once for every element, and each time, the current element is assigned to the variable `food` (in this case). Another example:

```
for number in range(1,100):
    print "Hello, world!"
    print "Just", 100 - number, "more to go..."

print "Hello, world"
print "That was the last one.. Phew!"
```

The function `range` returns a list of numbers in the range given (including the first, excluding the last... In this case, [1..99]). So, to paraphrase this:

The contents of the loop is executed *for each number in the range* of numbers from (and including) *1* up to (and excluding) *100*. (What the loop body and the following statements actually *do* is left as an exercise.)

But this doesn't really help us with our cooking problem. If we want to check the spam a hundred times, then it would be quite a nice solution; but we don't know if that's enough — or if it's too much. We just want to keep checking it *while it is not hot enough* (or, until it *is* hot enough — a matter of point-of-view). So, we use *while*:

```
# Spam-cooking program

# Fetch the function *sleep*
from time import sleep

print "Please start cooking the spam. (I'll be back in 3 minutes.)"

# Wait for 3 minutes (that is, 3*60 seconds)...
sleep(180)

print "I'm baaack :)"

# How hot is hot enough?
hot_enough = 50

temperature = input("How hot is the spam? ")
while temperature < hot_enough:
    print "Not hot enough... Cook it a bit more..."
    sleep(30)
    temperature = input("OK. How hot is it now? ")

print "It's hot enough - You're done!"
```

*New things* in this example...

1. Some useful functions are stored in *modules* and can be *imported*. In this case we import the function `sleep` (which sleeps for a given number of seconds) from the module `time` which comes with Python. (It is possible to make your own modules too...)

## Exercise 2

Write a program that continually reads in numbers from the user and adds them together until the sum reaches 100. Write another program that reads 100 numbers from the user and prints out the sum.

## Bigger Programs — Abstraction

If you want an overview of the contents of a book, you don't plow through all pages — you take a look at the table of contents, right? It simply lists the main topics of the book. Now — imagine writing a cookbook. Many of the recipes, like “Creamy Spam and Macaroni” and “Spam Swiss Pie” may contain similar things, like spam, in this case - yet you wouldn't want to repeat how to make spam in every recipe. (OK... So you don't actually *make* spam... But bear with me for the sake of example :)). You'd put the recipe for spam in a separate chapter, and simply refer to it in the other recipes. So — instead of writing the entire recipe every time, you only had to use the name of a chapter. In computer programming this is called *abstraction*.

Have we run into something like this already? Yup. Instead of telling the computer exactly how to get an answer from the user (OK - so we couldn't really do this... But we couldn't really make spam either, so there... :)) we simply used `input` - a function. We can actually make our own functions, to use for this kind of abstraction.

Let's say we want to find the largest integer that is less than a given positive number. For instance, given the number 2.7, this would be 2. This is often called the “floor” of the given number. (This could actually be done with built-in Python function `int`, but again, bear with me...) How would we do this? A simple solution would be to try all possibilities from zero:

```
number = input("What is the number? ")

floor = 0
while floor <= number:
    floor = floor+1
floor = floor-1

print "The floor of", number, "is", floor
```

Notice that the loop ends when `floor` is *no longer* less than (or equal to) the number; we add one *too much* to it. Therefore we have to subtract one afterwards. What if we want to use this “floor”-thing in a complex mathematical expression? We would have to write the entire loop for every number that needed “floor”-ing. Not very nice... You have probably guessed what we will do instead: Put it all in a *function* of our own, called “floor”:

```
def floor(number):
    result = 0
    while result <= number:
        result = result+1
    result = result-1
    return result
```

*New things* in this example...

1. Functions are defined with the keyword `def`, followed by their name and the expected parameters in parentheses.
2. If the function is to return a value, this is done with the keyword `return` (which also automatically

ends the function.

Now that we have defined it, we can use it like this:

```
x = 2.7
y = floor(2.7)
```

After this, `y` should have the value 2. It is also possible to make functions with more than one parameter:

```
def sum(x,y):
    return x+y
```

## Exercise 3

Write a function that implements Euclid’s method for finding a common factor of two numbers. It works like this:

1. You have two numbers, `a` and `b`, where `a` is larger than `b`
2. You repeat the following until `b` becomes zero:
3. `a` is changed to the value of `b`
4. `b` is changed to the remainder when `a` (before the change) is divided by `b` (before the change)
5. You then return the last value of `a`

*Hints:*

1. Use `a` and `b` as parameters to the function
2. Simply assume that `a` is greater than `b`
3. The *remainder* when `x` is divided by `z` is calculated by the expression `x % z`
4. Two variables can be assigned to simultaneously like this: `x, y = y, y+1`. Here `x` is given the value of `y` (that is, the value `y` had before the assignment) and `y` is incremented by one

## More About Functions

How did the exercise go? Was it difficult? Still a bit confused about functions? Don’t worry — I haven’t left the topic quite yet.

The sort of abstraction we have used when building functions is often called *procedural abstraction*, and many languages use the word *procedure* along with the word *function*. Actually, the two concepts are different, but both are called functions in *Python* (since they are defined and used in the same way, more or less.)

What is the difference (in other languages) between functions and procedures? Well — as you saw in the previous section, functions can *return* a value. The difference lies in that procedures *do not* return such a value. In many ways, this way of dividing functions into two types — those who *do* and those who *don’t* return values — can be quite useful.

A function that *doesn’t* return a value (a “procedure”) is used as a “sub-program” or subroutine. We call the function, and the program does some stuff, like making whipped cream or whatever. We can use this function in many places without rewriting the code. (This is called *code reuse* — more on that later.)

The usefulness of such a function (or procedure) lies in its *side effects* — it changes its environment (by mixing the sugar and cream and whipping it, for instance...) Let’s look at an example:

```
def hello(who):
    print "Hello,", who

hello("world")
# Prints out "Hello, world"
```

Printing out stuff is considered a *side effect*, and since that is all this function does, it is quite typical for a so-called procedure. But... It doesn't really change its environment does it? How could it do that?

Let's try:

```
# The *wrong* way of doing it
age = 0

def setAge(a):
    age = a

setAge(100)
print age
# Prints "0"
```

What's wrong here? The problem is that the function `setAge` creates its own *local* variable, also named `age` which is only seen inside `setAge`. How can we avoid that? We can use something called `global` variables.

*Note:* Global variables are not used much in Python. They easily lead to bad structure, or what is called spaghetti code. I use them here to lead up to more complex techniques - please avoid them if you can.

By telling the interpreter that a variable is global (done with a statement like `global age`) we effectively tell it to use the variable *outside* the function instead of creating a new local one. (So, it is *global* as opposed to *local*.) The program can then be rewritten like this:

```
# The correct, but not-so-good way of doing it
age = 0

def setAge(a):
    global age
    age = a

setAge(100)
print age
# Prints "100"
```

When you learn about objects (below), you'll see that a more appropriate way of doing this would be to use an object with an `age` property and a `setAge` method. In the section on data structures, you will also see some better examples of functions that change their environment.

Well — what about *real functions*, then? What is a function, really? Mathematical functions are like a kind of “machine” that gets some input and calculates a result. It will return the same result every time, when presented with the same input. For instance:

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

This is the same as the mathematical function  $f(x)=x^2$ . It behaves like a nice function, in that it *only* relies on its input, and it *does not* change its environment in any way.

So — I have outlined two ways of making functions: One type is more like a procedure, and doesn't

return a result; the other is more like a mathematical function and doesn't do anything *but* returning a result (almost). Of course, it is possible to do something in between the two extremes, although when a function changes things, it should be clear that it does. You could signal this through its name, for instance by using only a noun for “pure” functions like `square` and an imperative for procedure-like functions like `setAge`.

## More Ingredients — data structures

Well — you know a lot already: How to get input and give output, how to structure complicated algorithms (programs) and to perform arithmetic; and yet the best is still to come.

What ingredients have we been using in our programs up until now? Numbers and strings. Right? Kinda boring... No let's introduce a couple of other ingredients to make things a bit more exciting.

Data structures are ingredients that structure data. (Surprise, surprise...) A single number doesn't really have much structure, does it? But let's say we want more numbers put together to a single ingredient — that would have some structure. For instance, we might want a list of numbers. That's easy:

```
[3,6,78,93]
```

I mentioned lists in the section on loops, but didn't really say much about them. Well — this is how you make them. Just list the elements, separated by commas and enclosed in brackets.

Let us jump into an example that calculates primes (numbers divisible only by themselves or 1):

```
# Calculate all the primes below 1000
# (Not the best way to do it, but...)

result = [1]
candidates = range(3,1000)
base = 2
product = base

while candidates:
    while product < 1000:
        if product in candidates:
            candidates.remove(product)
        product = product+base
    result.append(base)
    base = candidates[0]
    product = base
    del candidates[0]

result.append(base)
print result
```

*New things* in this example...

1. The built-in function `range` actually returns a list that can be used like all other lists. (It includes the first index, but not the last.)
2. A list can be used as a logic variable. If it is not empty, then it is *true* — if it *is* empty, then it is *false*. Thus, `while candidates` means “while the list named `candidates` is not empty” or simply “while there are still candidates”.
3. You can write `if someElement in someList` to check if an element is in a list.

4. You can write `someList.remove(someElement)` to remove `someElement` from `someList`.
5. You can append an element to a list by using `someList.append(something)`. Actually, you can use `+` too (as in `someList = someList+[something]`) but it is not as efficient.
6. You can get at an element of a list by giving its position as a number (where the first element, strangely, is element 0) in brackets after the name of the list. Thus `someList[3]` is the fourth element of the list `someList`. (More on this below.)
7. You can delete variables by using the keyword `del`. It can also be used (as here) to delete elements from a list. Thus `del someList[0]` deletes the first element of `someList`. If the list was `[1,2,3]` before the deletion, it would be `[2,3]` afterwards.

Before going on to explaining the mysteries of indexing list elements, I will give a brief explanation of the example.

This is a version of the ancient algorithm called “The Sieve of Eratosthenes” (or something close to that). It considers a set (or in this case, a list) of candidate numbers, and then systematically removes the numbers *known* not to be primes. How do we know? Because they are products of two other numbers.

We start with a list of candidates containing numbers `[2..999]` — we know that 1 is a prime (actually, it may or may not be, depending on who you ask), and we wanted all primes *below* 1000. (Actually, our list of candidates is `[3..999]`, but 2 is also a candidate, since it is our first `base`). We also have a list called `result` which at all times contains the updated results so far. To begin with, this list contains only the number 1. We also have a variable called `base`. For each iteration (“round”) of the algorithm, we remove all numbers that are some multiple of this base number (which is always the smallest of the candidates). After each iteration, we know that the smallest number left is a prime (since all the numbers that were products of the smaller ones are removed — get it?). Therefore, we add it to the result, set the new base to this number, and remove it from the candidate list (so we won’t process it again.) When the candidate list is empty, the result list will contain all the primes. Clever, huh?

*Things to think about:* What is special with the first iteration? Here the base is 2, yet that too is removed in the “sieving”? Why? Why doesn’t that happen to the other bases? Can we be sure that `product` is always in the candidate list when we want to remove it? Why?

Now — what next? Ah, yes... Indexing. And slicing. These are the ways to get at the individual elements of Python lists. You have already seen ordinary indexing in action. It is pretty straightforward. Actually, I have told you all you need to know about it, except for one thing: Negative indices count from the *end* of the list. So, `someList[-1]` is the *last* element of `someList`, `someList[-2]` is the element before that, and so on.

Slicing, however, should be new to you. It is similar to indexing, except with slicing you can target an entire *slice* of the list, and not just a *single element*. How is it done? Like this:

```
food = ["spam", "spam", "eggs", "sausages", "spam"]

print food[2:4]
# Prints "['eggs', 'sausages']"
```

## More Abstraction — Objects and Object-Oriented Programming

Now there’s a buzz-word if ever there was one: “Object-oriented programming.”

As the section title suggests, object-oriented programming is just another way of abstracting away

details. Procedures abstract simple statements into more complex operations by giving them a name. In OOP, we don't just treat the operations this way, but *objects*. (Now, *that* must have been a big surprise, huh?) For instance, if we were to make a spam-cooking-program, instead of writing lots of procedures that dealt with the temperature, the time, the ingredients etc., we could lump it together into a *spam-object*. Or, perhaps we could have an oven-object and a clock-object too... Now, things like temperature would just be attributes of the spam-object, while the time could be read from the clock-object. And to make our program *do* something, we could teach our object some *methods*; for instance, the oven might know how to cook the spam etc.

So — how do we do this in Python? Well we can't just make an object directly. Instead of just making an oven, we make a *recipe* describing *how ovens are*. This recipe then describes a *class* of objects that we call ovens. A very simple oven class might be:

```
class Oven:
    def insertSpam(self, spam):
        self.spam = spam

    def getSpam(self):
        return self.spam
```

Now, does this look weird, or what?

*New things* in this example...

1. Classes of objects are defined with the keyword `class`.
2. Class names usually start with capital letters, whereas functions and variables (as well as methods and attributes) start with lowercase letters.
3. Methods (i.e. the functions or operations that the objects know how to do) are defined in the normal way, but *inside* the class block.
4. All object methods should have a first parameter called `self` (or something similar...) The reason will (hopefully) become clear in a moment.
5. Attributes and methods of an object are accessed like this: `mySpam.temperature = 2`, or `dilbert.be_nice()`.

I would guess that some things are still a bit unclear about the example. For instance, what is this `self` thing? And, now that we have an object recipe (i.e. *class*), how do we actually make an object?

Let's tackle the last point first. An object is created by calling the classname as if it were a function:

```
myOven = Oven()
```

`myOven` now contains an `Oven` object, usually called an *instance* of the class `oven`. Let's assume that we have made a class `spam` as well; then we could do something like:

```
mySpam = Spam()
myOven.insertSpam(mySpam)
```

`myOven.spam` would now contain `mySpam`. How come? Because, when we call one of the methods of an object, the first parameter, usually called `self`, always contains the object itself. (Clever, huh?) Thus, the line `self.spam = spam` sets the *attribute* `spam` of the current `oven` object to the value of the *parameter* `spam`. Note that these are two different things, even though they are both called `spam` in this example.

## Answer to Exercise 3

Here is a very concise version of the algorithm:

```
def euclid(a,b):  
    while b:  
        a,b = b,a % b  
    return a
```

## References

[1] Recipe for Fiesta Spam Salad taken from the [Hormel Foods](#) Digital Recipe Book