Think Python

How to Think Like a Computer Scientist

Version 1.1.24+Kart [Python 3.2]

Think Python

How to Think Like a Computer Scientist

Version 1.1.24+Kart [Python 3.2]

Allen Downey

Green Tea Press

Needham, Massachusetts

Printing history:

April 2002: First edition of *How to Think Like a Computer Scientist*.

August 2007: Major revision, changed title to *How to Think Like a (Python) Programmer*.

June 2008: Major revision, changed title to *Think Python: How to Think Like a Computer Scientist*.

Green Tea Press
9 Washburn Ave
Needham MA 02492

The original form of this book is LaTeX source code. Compiling this LaTeX source has the effect of generating a device-independent representation of a textbook, which can be converted to other formats and printed.

The LaTeX source for this book is available from `http://www.thinkpython.com`

Chapter 1

More Functions

1.1 Math functions

Python has a math module that provides most of the familiar mathematical functions. A **module** is a file that contains a collection of related functions.

Before we can use the module, we have to import it:

```
>>> import math
```

This statement creates a **module object** named math. If you print the module object, you get some information about it:

```
>>> print(math)
<module 'math' (built-in)>
```

The module object contains the functions and variables defined in the module. To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period). This format is called **dot notation**.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

The first example computes the logarithm base 10 of the signal-to-noise ratio. The math module also provides a function called `log` that computes logarithms base `e`. (We're not interested in the math here; we're just using math because it provides an easy way to call built-in Python functions. Focus on syntax here, not log and radians and all of that.)

The second example finds the sine of `radians`. The name of the variable is a hint that `sin` and the other trigonometric functions (`cos`, `tan`, etc.) take arguments in radians. To convert from degrees to radians, divide by 360 and multiply by 2π:

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.7071067811865475
```

The expression `math.pi` gets the variable `pi` from the math module. The value of this variable is an approximation of π, accurate to about 15 digits.

If you know your trigonometry, you can check the previous result by comparing it to the square root of two divided by two:

```
>>> math.sqrt(2) / 2.0
0.7071067811865476
```

1.2 Composition

One of the most useful features of programming languages is their ability to take small building blocks and **compose** them. For example, the argument of a function can be any kind of expression, including arithmetic operators:

```
x = math.sin(degrees / 360.0 * 2 * math.pi)
```

And even function calls:

```
x = math.exp(math.log(x+1))
```

Almost anywhere you can put a value, you can put an arbitrary expression, with one exception: the left side of an assignment statement has to be a variable name. Any other expression on the left side is a syntax error.

```
>>> minutes = hours * 60                # right
>>> hours * 60 = minutes                # wrong!
SyntaxError: can't assign to operator
```

1.3 Flow of execution

The **flow of execution** refers to the order in which Python code is executed.

Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom.

Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.

A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, executes all the statements there, and then comes back to pick up where it left off.

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to execute the statements in another function. But while executing that new function, the program might have to execute yet another function!

Fortunately, Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

What's the moral of this sordid tale? When you read a program, you don't always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.

Let's consider an example. Here are three functions being defined at the Python shell:

```
>>> def first(x):
...   return x + 5
...
>>> def second(x):
...   x = x + first(9)
...   return x
...
>>> def third(x):
...   print(second(2))
...   print('Hi!')
...   print(first(4))
...   return x
...
>>>
```

To begin, type the expression `first(8)`. The output is `13`. To arrive at this output, Python calls `first` with the `x` parameter of `first` set to `8`. `first` then returns its parameter plus 5, which is printed by the Python shell.

Next, let's try `second(3)`. This calls `second` with the `x` parameter of `second` having value 3. Inside `second`, the parameter `x` is assigned the value `x + first(9)`. To do this requires knowing the result of `first(9)`. So here, the execution of `second` is suspended, and `first` is executed with parameter 9. `first` returns the value `14`. At this point, execution returns back to `second`, and `x + first(9)` has value `3 + 14 = 17`. Note that the `x` in `second` is still `3` when `first` returns. Each `x`, even though it has the same name as the other `x`s, is independent and local to each function.

Finally, let's try `third(3)`. To understand the output for this one, we really have to be careful "remembering" where we are in each function execution. First of all, we have `print(second(2))`. To do this, we temporarily suspend execution of `third` in order to run `second`. So, `second` gets called with its `x` parameter set to `2`. For `second` to execute, it has to further call `first`, as described above. `second(2)` evaluates to 16, at which point execution returns back to `third` and 16 can be printed. `third` then continues with its next line (remember that functions "remember" their place after making a function call), whose result is to output `Hi!`. Finally, the result of `first(4)` is printed, which again requires `third` to be suspended to allow `first` to run. Once `first` returns `9`, we again continue where we left off in `third`. The only thing remaining is the `return` statement, which returns `3` to the Python shell. `x` still has the value `3` here: nowhere in `third` was the value of `x` changed!

At this point, I'd recommend playing with this example in the visualizer. Step through the code to follow the program flow. Notice that the function definitions are executed in one step, but produce no output: what they do is "store" the function definition for later calling. Only when you call the function does the program jump up to the function code and execute the function body.

1.4 Variable Lookup

To determine whether a variable access is legal, Python uses a rule that we will refer to as the LGB rule. This means that there are three possibilities for where Python can find a variable. First, it checks to see whether you are referring to a local variable (that's the L). If not, it checks to see if there is a global variable of that name (that's the G). If not, and finally, Python checks for a built-in variable having the name you specified. If it can't find it there, then it finally gives up and gives you an error saying that the variable is not defined.

Local variables are those that are defined inside of a function or listed as parameters to a function. To create a global variable, you define it outside of any function definition.

A common error is to try to access a variable that is defined but not covered by this rule. Here is an example:

```
>>> def first():
...   total = 5
...   second()
...
>>> def second():
...   print(total)
...
>>> first()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 3, in first
File "<stdin>", line 2, in second
NameError: global name 'total' is not defined
>>>
```

When `first` is called, a local variable named `total` is assigned. Then, `first` calls `second`, who in turn tries to access `total`. Can `second` make this access? No! `total` is not local to `second` (no L), and `total` is not a global variable (no G), and `total` is not built-in to Python (no B). (`total` is a local variable of `first`.) So, accessing `total` from `second` is an error.

Also, note the traceback provided by Python when the variable access failed. From bottom to top, it indicates that `second` was active, and that `second` was called from `first`, and that `first` was called by `<module>` (which just means that you called `first`, not some other function). These errors are very useful for helping you determine where an error occurred and how that condition arose.