Extracted from:

Practical Programming

An Introduction to Computer Science Using Python

The Pragmatic Programmers

Practical Programming

An Introduction to Computer Science Using Python

Jennifer Campbell
Paul Gries
Jason Montojo
Greg Wilson

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragprog.com

Printed in the United States of America.

ISBN-10: 1-934356-27-1

ISBN-13: 978-1-934356-27-2

Printed on acid-free paper.

P1.0 printing, April 2009

Version: 2009-5-9

Operator	Symbol
**	Exponentiation
-	Negation
*, /, %	Multiplication, division, and remainder
+-	Addition and subtraction

Figure 2.5: Arithmetic operators by precedence

precedence than -; in other words, when an expression contains a mix of operators, * and / are evaluated before - and +. This means that what we actually calculated was 212 - ((32.0 * 5.0) / 9.0).

We can alter the order of precedence by putting parentheses around parts of the expression, just as we did in Mrs. Singh's fourth-grade class:

Download basic/precedence_diff.cmd

```
>>> (212 - 32.0) * 5.0 / 9.0
100.0
```

The order of precedence for arithmetic operators is listed in Figure 2.5. It's a good rule to parenthesize complicated expressions even when you don't need to, since it helps the eye read things like 1+1.7+3.2*4.4-16/3.

2.4 Variables and the Assignment Statement

Most handheld calculators[3] have one or more memory buttons. These store a value so that it can be used later. In Python, we can do this with a *variable*, which is just a name that has a value associated with it. Variables' names can use letters, digits, and the underscore symbol. For example, X, species5618, and degrees_celsius are all allowed, but 777 isn't (it would be confused with a number), and neither is no-way! (it contains punctuation).

You create a new variable simply by giving it a value:

Download basic/assignment.cmd

```
>>> degrees_celsius = 26.0
```

3. And cell phones, and wristwatches, and...

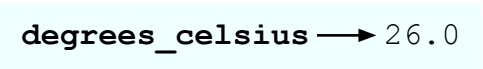

Figure 2.6: Memory model for a variable and its associated value

This statement is called an *assignment statement*; we say that degrees_celsius is *assigned* the value 26.0. An assignment statement is executed as follows:

1. Evaluate the expression on the right of the = sign.
2. Store that value with the variable on the left of the = sign.

In Figure 2.6, we can see the *memory model* for the result of the assignment statement. It's pretty simple, but we will see more complicated memory models later.

Once a variable has been created, we can use its value in other calculations. For example, we can calculate the difference between the temperature stored in degrees_celsius and the boiling point of water like this:

Download basic/variable.cmd

```
>>> 100 - degrees_celsius
74.0
```

Whenever the variable's name is used in an expression, Python uses the variable's value in the calculation. This means that we can create new variables from old ones:

Download basic/assignment2.cmd

```
>>> difference = 100 - degrees_celsius
```

Typing in the name of a variable on its own makes Python display its value:

Download basic/variable2.cmd

```
>>> difference
74.0
```

What happened here is that we gave Python a very simple expression—one that had no operators at all—so Python evaluated it and showed us the result.

It's no more mysterious than asking Python what the value of 3 is:

Download basic/simplevalue.cmd

```
>>> 3
3
```

Variables are called variables because their values can change as the program executes. For example, we can assign difference a new value:

Download basic/variable3.cmd

```
>>> difference = 100 - 15.5
>>> difference
84.5
```

This does *not* change the results of any calculations done with that variable before its value was changed:

Download basic/variable4.cmd

```
>>> difference = 20
>>> double = 2 * difference
>>> double
40
>>> difference = 5
>>> double
40
```

As the memory models illustrate in Figure 2.7, on the following page, once a value is associated with double, it stays associated until the program explicitly overwrites it. Changes to other variables, like difference, have no effect.

We can even use a variable on both sides of an assignment statement:

Download basic/variable5.cmd

```
>>> number = 3
>>> number
3
>>> number = 2 * number
>>> number
6
>>> number = number * number
>>> number
36
```

This wouldn't make much sense in mathematics—a number cannot be equal to twice its own value—but = in Python doesn't mean "equals to." Instead, it means "assign a value to."

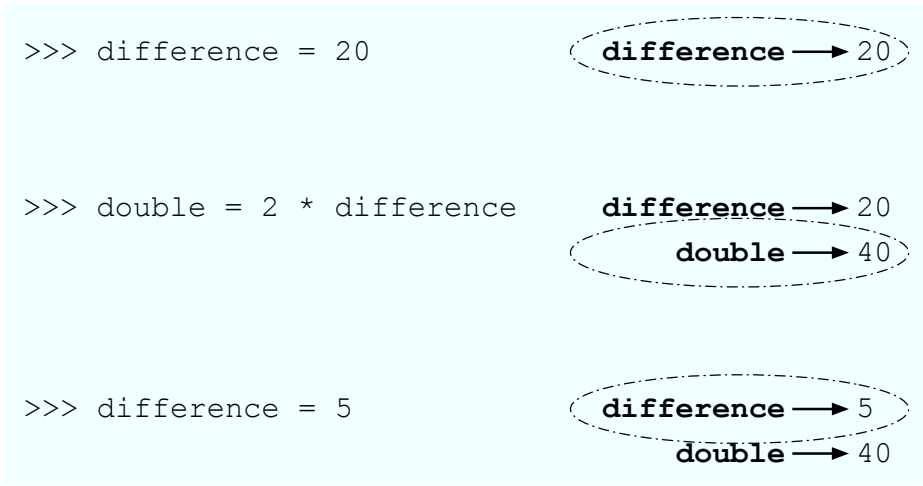

Figure 2.7: Changing a variable's value

When a statement like number = 2 * number is evaluated, Python does the following:

1. Gets the value currently associated with number
2. Multiplies it by 2 to create a new value
3. Assigns that value to number

Combined Operators

In the previous example, variable number appeared on both sides of the assignment statement. This is so common that Python provides a shorthand notation for this operation:

Download basic/variable6.cmd

```
>>> number = 100
>>> number -= 80
>>> number
20
```

Here is how a *combined operator* is evaluated:

1. Evaluate the expression to the right of the = sign.
2. Apply the operator attached to the = sign to the variable and the result of the expression.
3. Assign the result to the variable to the left of the = sign.

Note that the operator is applied *after* the expression on the right is evaluated:

Download basic/variable7.cmd

```
>>> d = 2
>>> d *= 3 + 4
>>> d
14
```

All the operators in Figure 2.5, on page 27, have shorthand versions. For example, we can square a number by multiplying it by itself:

Download basic/variable8.cmd

```
>>> number = 10
>>> number *= number
>>> number
100
```

which is equivalent to this:

Download basic/variable9.cmd

```
>>> number = 10
>>> number = number * number
>>> number
100
```

2.5 When Things Go Wrong

We said earlier that variables are created by assigning them values. What happens if we try to use a variable that hasn't been created yet?

Download basic/undefined_var.cmd

```
>>> 3 + something
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'something' is not defined
```

This is pretty cryptic. In fact, Python's error messages are one of its few weaknesses from the point of view of novice programmers. The first two lines aren't much use right now, though they'll be indispensable when we start writing longer programs. The last line is the one that tells us what went wrong: the name something wasn't recognized.

Here's another error message you might sometimes see:

Download basic/syntax_error.cmd

```
>>> 2 +
  File "<stdin>", line 1
    2 +
      ^
SyntaxError: invalid syntax
```

The rules governing what is and isn't legal in a programming language (or any other language) are called its *syntax*. What this message is telling us is that we violated Python's syntax rules—in this case, by asking it to add something to 2 but not telling it what to add.

2.6 Function Basics

Earlier in this chapter, we converted 80 degrees Fahrenheit to Celsius. A mathematician would write this as $f(t)=\frac{5}{9}(t\text{-}32)$, where t is the temperature in Fahrenheit that we want to convert to Celsius. To find out what 80 degrees Fahrenheit is in Celsius, we replace t with 80, which gives us $f(80) = \frac{5}{9}(80\text{-}32)$, or $26\frac{2}{3}$.

We can write functions in Python, too. As in mathematics, they are used to define common formulas. Here is the conversion function in Python:

Download basic/fahr_to_cel.cmd

```
>>> def to_celsius(t):
...     return (t - 32.0) * 5.0 / 9.0
...
```

It has these major differences from its mathematical equivalent:

- A function definition is another kind of Python statement; it defines a new name whose value can be rather complicated but is still just a value.
- The *keyword* def is used to tell Python that we're defining a new function.
- We use a readable name like to_celsius for the function rather than something like f whose meaning will be hard to remember an hour later. (This isn't actually a requirement, but it's good style.)
- There is a colon instead of an equals sign.
- The actual formula for the function is defined on the next line. The line is indented four spaces and marked with the keyword return.

Python displays a triple-dot prompt automatically when you're in the middle of defining a new function; you do not type the dots any more than you type the greater-than signs in the usual >>> prompt. If you're using a smart editor, like the one in Wing 101, it will automatically indent the *body* of the function by the required amount. (This is another reason to use Wing 101 instead of a basic text editor like Notepad or Pico: it saves a lot of wear and tear on your spacebar and thumb.)

Here is what happens when we ask Python to evaluate to_celsius(80), to_celsius(78.8), and to_celsius(10.4):

Download basic/fahr_to_cel_2.cmd

```
>>> to_celsius(80)
26.666666666666668
>>> to_celsius(78.8)
26.0
>>> to_celsius(10.4)
-12.0
```

Each of these three statements is called a *function call*, because we're calling up the function to do some work for us. We have to define a function only once; we can call it any number of times.

The general form of a function definition is as follows:

```
def function_name(parameters):
    block
```

As we've already seen, the def keyword tells Python that we're defining a new function. The name of the function comes next, followed by zero or more *parameters* in parentheses and a colon. A *parameter* is a variable (like t in the function to_celsius) that is given a value when the function is called. For example, 80 was assigned to t in the function call to_celsius(80), and then 78.8 in to_celsius(78.8), and then 10.4 in to_celsius(10.4). Those actual values are called the *arguments* to the function.

What the function does is specified by the *block* of statements inside it. to_celsius's block consisted of just one statement, but as we'll see later, the blocks making up more complicated functions may be many statements long.

Finally, the return statement has this general form:

```
return expression
```

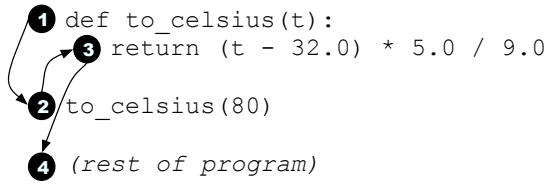

Figure 2.8: Function control flow

and is executed as follows:

1. Evaluate the expression to the right of the keyword return.
2. Use that value as the result of the function.

It's important to be clear on the difference between a function *definition* and a function *call*. When a function is defined, Python records it but doesn't execute it. When the function is called, Python jumps to the first line of that function and starts running it (see Figure 2.8). When the function is finished, Python returns to the place where the function was originally called.

Local Variables

Some computations are complex, and breaking them down into separate steps can lead to clearer code. Here, we break down the evaluation of the polynomial $ax^2 + bx + c$ into several steps:

Download basic/multi_statement_block.cmd

```
>>> def polynomial(a, b, c, x):
...     first  = a * x * x
...     second = b * x
...     third  = c
...     return first + second + third
...
>>> polynomial(2, 3, 4, 0.5)
6.0
>>> polynomial(2, 3, 4, 1.5)
13.0
```

Variables like first, second, and third that are created within a function are called *local variables*. These variables exist only during function execution; when the function finishes executing, the variables no longer exist. This means that trying to access a local variable from outside the

function is an error, just like trying to access a variable that has never been defined:

Download basic/local_variable.cmd

```
>>> polynomial(2, 3, 4, 1.3)
11.280000000000001
>>> first
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'first' is not defined
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

As you can see from this example, a function's parameters are also local variables. When a function is called, Python assigns the argument values given in the call to the function's parameters. As you might expect, if a function is defined to take a certain number of parameters, it must be passed the same number of arguments:[4]

Download basic/matching_args_params.cmd

```
>>> polynomial(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: polynomial() takes exactly 4 arguments (3 given)
```

The *scope* of a variable is the area of the program that can access it. For example, the scope of a local variable runs from the line on which it is first defined to the end of the function.

2.7 Built-in Functions

Python comes with many *built-in functions* that perform common operations. One example is `abs`, which produces the absolute value of a number:

Download basic/abs.cmd

```
>>> abs(-9)
9
```

4. We'll see later how to create functions that take any number of arguments.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Practical Programming's Home Page
http://pragprog.com/titles/gwpy
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/gwpy.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	support@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com
Contact us:	1-800-699-PROG (+1 919 847 3884)