

Python Jump Start

Bob Brown

College of Computing and Software Engineering
Kennesaw State University

About Reading and Writing

We learned to write English by first reading. We started with simple material and progressed to increasingly complex material. Then we wrote, also starting with simple material. We should learn to write computer programs in the same way. The purpose of this “Jump Start” is to provide *just enough* information to let you read many Python programs. As with reading English, when you come upon something you don’t understand in a Python program, *look it up!* You will increase your Python vocabulary just as you increased your English vocabulary. This document uses some programming terms of art, *i.e.* jargon. If you come across a term that’s unfamiliar to you, *look it up!*

This is not a reference document; you should read and absorb all of it, then start reading, and perhaps writing, Python programs. You will expand your knowledge of Python as you go along.

Why is it Called “Python?”

“When he began implementing Python, Guido van Rossum was also reading the published scripts from “Monty Python’s Flying Circus”, a BBC comedy series from the 1970s. Van Rossum thought he needed a name that was short, unique, and slightly mysterious, so he decided to call the language Python.”¹ So, despite the many graphics you will see, Python has nothing to do with snakes.

Getting Help

If you are learning Python, you are probably working in an IDE – Integrated Development Environment. The first thing you should do is to read the documentation... yes, really, read the documentation! Many IDEs have built-in help. Many Python IDEs present two windows, one in which you type your program and another into which you can type Python statements to See What Happens.™

There is a wealth of information on the Web. Start with <https://docs.python.org/>.

Python Identifiers

Identifiers are the names of things. Identifiers can name variables, functions, objects, and other entities that are a part of Python. Python identifiers must begin with a letter, then consist of letters, digits, and underscores; other special characters are not allowed. Python identifiers are not length-limited, but **are case sensitive**, so *Number* and *number* are not the same.

¹ From the *General Python FAQ*: <https://docs.python.org/2/faq/general.html>

Identifiers should be meaningful. While `i = 1` is perfectly valid, `index = 1` is preferable. When an identifier comprises two or more words, use camelCase (`rowIndex`), CapWords (`RowIndex`), or underscores (`row_index`.) Don't mix them; pick a style and stick with it.

Python Keywords and Operators

There are more than thirty [Python keywords](#), including things like `if`, `else`, `while`, and `import`. The two Boolean constants `True` and `False` are capitalized; the other keywords are not. Rather than memorizing a list of keywords, learn them as you need them.

Python arithmetic operators include the usual suspects, `+`, `-`, `*`, and `/` for addition, subtraction, multiplication, and division. You can also use `//` for floor division, which returns the integer part of the quotient, `**` for exponentiation, and `%` for modulus.

The assignment operator is the equal sign, `=`. There are numerous compound assignment operators, so `x += 1` is the same as `x = x + 1`.

The comparison operators are `==` (two consecutive equal signs) for equality, `!=` for not equal, and `<` and `>` for less than and greater than. There are two other comparison operators and several other types of [Python operators](#). Learn them as you need them.

Comments

Everything following the hash symbol, `#`, to the end of the line is a comment. Comments are ignored by the Python interpreter; they exist help those reading the program, including the original programmer, to understand it.

```
# This is a comment that takes up the entire line
rowIndex += offset # This comment might explain "offset"
```

Resist the urge to comment the obvious; it clutters your programs and makes them hard to read.

```
rowIndex += 1 # Add 1 to rowIndex
```

An easy way to remove code temporarily when testing is to “comment it out,” that is, to prefix each line with a `#`-symbol. Many Python IDEs have a function that does this automatically. You can also enclose such a block in triple apostrophes.

Variables, Data Types

Python variable names follow the rules for identifiers, as described earlier. A variable name is nothing more than a name for an area in the computer's memory. How the contents of that area are used depends on the *data type* associated with the variable.

Python has four primitive data types: integers, floating-point numbers, strings, and Booleans. Integers are whole numbers, including zero and the negative numbers. Floating point numbers can have a whole number part and fractional part. Strings hold sequences of characters, like `"Hello, world!"` Booleans have only two possible values: `True` and `False`.

Python also has many compound data types, including arrays and lists, data types for date and time, and for enumeration. You will learn these as you need them.

Assigning a value to a variable for the first time *declares* the variable, that is, creates it.

```
answer = 'yes' # The variable "answer" exists and has a value
```

Python is *dynamically typed*. That means the data type of a variable can change when a new value is assigned to it.

```
answer = 'yes' # The variable "answer" is a string
answer = 42    # And now it's an integer
```

It is, in general, good practice not to change the type of a variable during the execution of a program. Python has a built-in `type()` that will return the type of a variable is passed to it.²

```
>>> answer = 'yes'
>>> type(answer)
<class 'str'>
>>> answer = 42
>>> type(answer)
<class 'int'>
```

Block structure and Indentation

In many languages, indentation is used for pretty-printing, that is to make the program more readable, but is ignored by the language processor. In Python, indentation is used to delineate syntactic structures called *blocks*, and so it has meaning to the Python language processor. It also has the effect of making the program more readable.

```
if i < 0:                # The colon starts a block
    print "i is negative" # This is within the block
else:                    # A new block at the same level
    print "i is nonnegative"
    if i < 10:           # An inner block; indented further
        print "i has one digit"
    else:
        print "i has multiple digits"
```

Indentation must be consistent; “ragged” indentation will cause an error, as will mixing tabs and spaces. Python’s specified best practice in [PEP-8](#) is to use four spaces for each block level.³

Defining and Using Functions

Python has hundreds of functions, like the `type()` function described earlier. You can write your own functions, too. You’d do that when the same bit of code would otherwise have to

² The Python interpreter uses `>>>` to prompt for input from the user.

³ That’s right... four keystrokes when one ought to do. Happily, [autopep8](#) will fix things up for you. Some editors and IDEs can also help with this.

appear in two or more places within your program. A function means the code appears in only one place and is called from several places as needed.⁴

A function is defined by the keyword `def` (for “define”) followed by the function name, the parameters of the function enclosed in parentheses, and a colon. The colon begins a block, and the function body – the code and data that comprise the function – are the contents of the block. *Parameters* are placeholders for data; when the function is called, the parameters are replaced by the *arguments* with which the function was called. If there are no parameters, the parentheses are empty but must still be present.

Here is a tiny Python program with an example of defining and using a function.

```
def greet(who, when):
    print("Good " + when + ", " + who + "!")

greet("Bob", "afternoon")    # Prints "Good afternoon, Bob!"
player = "Gina"
time = "evening"
greet(player, time)         # Prints "Good evening, Gina!"
```

In this program, the function is called `greet` and the parameters are `who` and `when`. There’s something a little strange, too. The `print` function is printing strings, but there are plus signs in there. The reason is that the plus operator is *overloaded* in Python to serve as the string concatenation operator, too. If plus is given two numbers, it adds; if it’s given two strings, it concatenates. Mixing strings and numbers gives an error. `z = 3 + "george" # Error!`

The first time the function is called, the arguments are two strings, “Bob” and “afternoon.” The program prints, “Good afternoon, Bob!”

Suppose we were writing a game in which a troll snarkily greets any player who approaches the troll bridge. We’d need variables to hold the player name and the time, and the last three lines in the example above show how that might work.

Scope of Variables and Other Objects

Before you can use a named object, such as a variable name or function name, it must be defined.

```
print (answer)           # Doesn't work because "answer" is not defined.
answer = 42              # This defines "answer"
print (answer)           # Prints 42
```

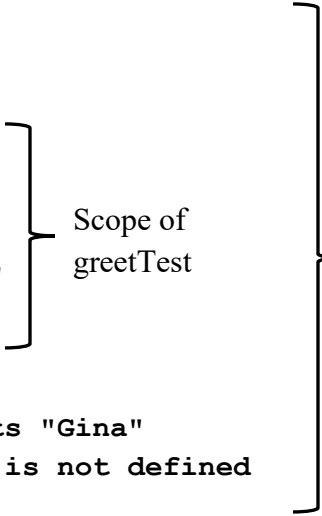
⁴ And *that* has the happy property that there’s only one place to change things if you find an error or improve your code. Otherwise, you’d have to search the entire program and make the same changes in many places, a process that is fraught with opportunity for error.

To use an object, it must be defined either within its own scope or within an enclosing scope. “Scope” only means whether an object is visible to a particular piece of code. Let’s look at an example:

```
# Program to illustrate "scope"
name = "Gina"

def greetTest():
    print ("Entering greetTest")
    print (name)          # Prints "Gina"
    when = "morning"
    print (when)         # Prints "morning"
    print ("Leaving greetTest")

greetTest()
print (name)           # Works as expected, prints "Gina"
print (when)          # Name Error: name 'when' is not defined
# End of the program
```



There are two important things to notice about this example. The variable **name** is defined in the outer scope, and so is available anywhere in the program, including within the function **greetTest**. The variable **when** is defined within **greetTest**; it is available within the scope of **greetTest**, including any inner scopes – there aren’t any inner scopes in the example – but it is not available outside the scope of **greetTest**.

If the same name is defined in an outer and inner scope, the definition in the current scope is used. Such a definition makes the name a *local variable*. Let’s look at the example again, slightly changed.

```
name = "Gina"
def greetTest():
    name = "George"      # Redefines name, now a local variable
    print (name)
greetTest()             # Prints "George"
print (name)            # Prints "Gina"
```

The variable **name** is defined in two different places, and has two different meanings. Within the function **greetTest**, **name** is a *local variable*, and changes to it do not propagate to the outer scope.

The fact the declaring a variable within a function creates a new variable is a Good Thing TM and can prevent hard-to-find errors. However, sometimes you want to be able a variable defined outside a function from within the function. The **global** keyword does that.

```

name = "Gina"
def greetTest():
    global name          # 'name' will NOT be redefined
    name = "George"     # The variable OUTSIDE the function changes
    print (name)
greetTest()             # Prints "George"
print (name)            # Prints "George" again

```

Python's predefined functions and objects have *global scope*. That is, they are available everywhere. It is possible to re-use the names of Python's predefined objects by writing new definitions. Doing so is very bad practice because it leads to subtle and hard-to-find errors.

Importing Modules

Python has a huge number of pre-written and pre-tested modules to perform various functions. The ability to *import* modules means you don't have to write code that's already been written by someone else. You can write your own modules, too, but doing so is beyond the scope⁵ of this document.

You get access to these modules using Python's **import** keyword. For example, the `time` module has a number of useful methods⁶, including pausing the program for a given number of seconds. Let's look at an example:

```

import time
print ("Here comes the punch line!")
print ("Wait for it...")
time.sleep (5)
print ("He's not that shaggy.")

```

This prints the first two lines, pauses for five seconds, then prints the punch line. The name of the `sleep` method is qualified by prefixing it with the module name and a dot: `time.sleep`. (Your Python code will generally not have embedded links especially not to shaggy dog stories.)

Suppose all we wanted was the `sleep` method. We could write something like this:

```

from time import sleep
print ("Here comes the punch line!")
print ("Wait for it...")
sleep (5)
print ("He's not that shaggy.")

```

⁵ "Beyond the scope" means either that the writer doesn't know enough to write about it, or that it won't fit in the space available. The question of which one is always left to the reader to decide. Some writers make "beyond the scope" implicit, omitting any mention of the subject in the hope that readers will ignore it or refrain from wondering about it. This depends for its success upon the phenomenon Douglas Adams called "S.E.P. – someone else's problem."

⁶ Functions encapsulated within objects are called *methods*.

We no longer have to qualify `sleep` by prefixing the module name, so we type less. Cool! The whole `time` module is still imported, but only the `sleep()` method becomes a part of the current program's name space.

Instantiating and Using Objects

Python is an object-oriented language, and in Python, everything is an *object*, even the variables and functions we've described so far. Object-oriented languages include the concept of classes. A *class* is a description of an object in much the same way a set of blueprints is a description of a house. We create an object from its class, similar to the way in which a house is built from a set of blueprints. Creating an object is called *instantiation*.

An object can hold properties, that is data, and methods, that is functions, both of which are described in the class definition. We create an object with the assignment operator, with the object name on the left and the class name, with optional arguments, on the right. By convention, Python objects are named using the CapWords⁷ style. Here's an example:

```
account = CheckingAccount("Bob Brown", "12345", 50.00)
```

Such a statement is a *constructor* for the object, analogous to constructing a house from blueprints. Executing the constructor is called *instantiating* the object. When it has been instantiated, `account` becomes an object of class `CheckingAccount`. It can have properties like `name`, `accountNumber`, and `balance`, and methods like `deposit()` and `withdraw()`. The properties and methods were defined in the class. We might be able to use it like this:

```
account.deposit(25.00) # Deposit $25
```

Each module that you might import can define one or more classes. Their properties and methods are described in the documentation for that module. Let's take a quick look at the LED object from the `gpiozero` module for Python on the Raspberry Pi. It is documented [here](#). We instantiate an LED object this way:

```
from gpiozero import LED  
redLed = LED(17) # The LED will use GPIO pin 17
```

We could have used any name we wanted in place of `redLed`. The only required argument is the pin number but there are three others. To see what they do, click the link above and read the documentation. Once the `redLed` object has been instantiated, it will have methods that we can use to make it do things using only one line of code each. Examples are `redLed.on()` and `redLed.blink(3, 1)`. Check the documentation to see what those will do.

You can write your own class definitions and modules, and will begin doing that as you advance with Python.

⁷ CapWords differs from camelCase in that the initial word is capitalized.

The Error Messages Are Your Friends

One of the sad things about learning to program is that we make mistakes. If we're lucky, the mistakes lead to error messages.⁸ Since we don't like making mistakes, those error messages sometimes make our brains slam shut, and that's a bad thing.

So, *read* the error messages; they exist to help you. Python's error messages will tell you where (at what line) the error was *detected*, how Python got there, and something about the error. Note that where the error was detected isn't always where the error is. You may have to put in some effort! For example, suppose a message says, "Name Error: name 'when' is not defined." The actual error is almost certainly somewhere else. Perhaps **when** wasn't defined at all, or it was misspelled, perhaps as **wjen**, when it was defined. On the other hand, if the message says, "Name Error: name 'wjen' is not defined" then the error is probably right there, in the form of a misspelled variable name.

A Sample Program

Here is a Python program taken from <https://www.programiz.com/python-programming/examples/prime-number> Read it carefully. Identify those features you learned here and look up anything that's new. That's how you learn to read and write!

```
# Python program to check if the input number is prime or not
# take input from the user
num = int(input("Enter a number: ")) # Look up int and input
# prime numbers are greater than 1
if num > 1:
    # check for factors
    for i in range(2,num):          # Look up for...in
        if (num % i) == 0:
            # Look back at Keywords and Operators; look up modulus, maybe
            print(num,"is not a prime number")
            print(i,"times",num//i,"is",num)
            break
    else:
        print(num,"is a prime number")

# if input number is less than
# or equal to 1, it is not prime
else:
    print(num,"is not a prime number")
```

⁸ If we're unlucky, our programs run, but give wrong answers with no indication of error. Language processors like Python can detect *syntax errors*, that is, errors in the expression of the language. They usually can't detect *logic errors*, in which we do the wrong thing, but in the right way.