Preview of Real Python Course 1, Intro to Python

If you like what you see, consider purchasing the entire course on RealPython.com - for just $60, you will get all three courses, with over 1,200 pages of content, packed with exercises, sample files, assignments, and videos!

Click here to purchase on RealPython.com.

Please share this document with anyone else who might benefit from learning practical programming.

Cheers!

Fundamentals: Functions and Loops

We already did some basic math using IDLE's interactive window. For instance, we saw that we could evaluate simple expressions just by typing them in at the prompt, which would display the answer:

```
>>> 6 * (1 + 6)
42
>>>
```

However, just putting that line into a script-

```
6 * (1 + 6)
```

-would be useless since we haven't actually told the program to do anything. If we want to display the result from a program, we have to rely on the print command again.

Go ahead and open a new script, save it as *arithmetic.py* and try displaying the results of some basic calculations:

```
print("1 + 1 =", 1 + 1)
print("2 * (2 + 3) =", 2 * (2 + 3))
print("1.2 / 0.3 =", 1.2 / 0.3)
print("5 / 2 =", 5 / 2)
```

Here we've used a single print statement on each line to combined two pieces of information by separating the values with a comma. The results of the numerical expressions on the right will automatically be calculated when we display it.

> **NOTE**: All of the spaces we included above were entirely optional, but they help to makes things easier to read.

When you save and run this script, it displays the results of your print commands as follows:

```
>>>
1 + 1 = 2
2 * (2 + 3) = 10
1.2 / 0.3 = 4.0
5 / 2 = 2.5
>>>
```

Assignment: Perform calculations on user input

Write a script called *exponent.py* that receives two numbers from the user and displays the result of taking the first number to the power of the second number. A sample run of the program should look like this (with example input that has been provided by the user included below):

```
>>>
Enter a base: 1.2
Enter an exponent: 3
1.2 to the power of 3 = 1.728
>>>
```

Keep the following in mind:

1. In Python, x^y (x raised to the power y) is calculated by using the expression `x ** y`
2. Before you can do anything with the user's input, you will have to store the results of both calls to `input()` in new objects
3. The `input()` function returns a string object, so you will need to convert the user's input into numbers in order to do arithmetic on them
4. You should use the string `format()` method to print the result
5. You can assume that the user will enter actual numbers as input

Create Your Own Functions

One of the main benefits of programming in Python is the ease with which different parts and pieces of code can be put together in new ways. Think of it like building with Lego bricks instead of having to craft everything by hand each time you start a project.

The Lego brick of programming is called a **function**. A function is basically a miniature program; it accepts input and produces output. We've already seen some examples of functions such as the `find()` string method - when called on a string, it takes some input and returns the location of that input within the string as its output.

> **NOTE**: Functions are like the functions from a math class: You provide the input and the function produces output.

We could create our own function that takes a number as its input and produces the square of that number as its output. In Python, this would look like:

```
def square(number):
    sqr_num = number ** 2
    return sqr_num
```

The `def` is short for "define" and lets Python know that we are about to define a new function. In this case, we called the function square and gave it one input variable (the part in parentheses) named number. A function's input (or, the value passed to a function) is called an **argument** of the function, and a function can take more than one argument.

The first line within our function multiplies number by itself and stores the result in a new variable named sqr_num. Then the last line of our function returns the value of sqr_num, which is the output of our function.

If you just type these three lines into a script, save it and run it, nothing will happen. The function doesn't do anything by itself.

However, now we can use the function later on from the main section of the script. For instance, try running this script:

```
def square(number):
    sqr_num = number ** 2
    return sqr_num

input_num = 5
output_num = square(input_num)

print(output_num)
```

By saying `output_num = square(input_num)`, we are calling up the function square and providing this function with the input variable `input_num`, which in this case has a value of 5. Our function then calculates 25 and returns the value of the variable `sqr_num`, which gets stored in our new variable `output_num`.

> **NOTE**: Notice the colon and the indentation after we defined our function. These aren't optional. This is how Python knows that we are still inside of the function. As soon as Python sees a line that isn't indented, that's the end of the function. Every line inside the function must be indented.

You can define many functions in one script, and functions can even refer to each other. However, it's important that a function has been defined before you try to use it. For instance, try running this code instead:

```
input_num = 5

output_num = square(input_num)

print(output_num)

def square(number):
    sqr_num = number * number
    return sqr_num
```

Here we've just reordered the two parts of our script so that the main section comes before the function. The problem here is that Python runs through our code from the top to the bottom - so when we call the square function on the second line, Python has no idea what we mean yet because we don't actually define the square function until later on in the script, and it hasn't gotten there yet. Instead we see an error:

```
NameError: name 'square' is not defined
```

To create a function that uses more than one input, all we need to do is separate each argument of the function with a comma. For instance, the following function takes two arguments as input and returns the difference, subtracting the second number from the first:

```
def return_difference(num1, num2):
    return num1 - num2
```

To call this function, we need to supply it with two inputs:

```
print(return_difference(3, 5))
```

This line will call our new `return_difference()` function, then display the result of -2 that the function returns.

> **NOTE**: Once a function returns a value with the return command, the function is done running; if any code appears inside the function after the return statement, it will never be run because the function has already returned its final result.

One last helpful thing about functions is that Python allows you to add special comments called **docstrings**. A docstring serves as documentation, helping to explain what a function does and how to use it. They're completely optional, but can be helpful if there's any chance that you'll either share your code with someone else or if you ever come back to your code later, once you've forgotten what it's supposed to do - which is why you should leave comments in the first place. A docstring looks just like a multi-line comment with three quotation marks, but it has to come at the very beginning of a function, right after the first definition line:

```
def return_difference(n1, n2):
    """Return the difference between two numbers.
       Subtracts n2 from n1."""
    return n1 - n2
```

The benefit of this (besides leaving a helpful comment in the code) is that we can now use the `help()` function to get information about this function. Assuming we defined this function by typing it in the interactive window or we already ran a script where it was defined, we can now type `help(return_difference)` and see:

```
>>> help(return_difference)
Help on function return_difference in module __main__:
return_difference(n1, n2)
    Return the difference between two numbers.
    Subtracts n2 from n1.
>>>
```

Of course, you can also call `help()` on the many other Python functions we'll see to get a quick reference on how they are used.

Functions Summary

So, what do we know about functions?

1. Functions require a function signature.
2. They do something useful.
3. They allow us re-use code without having to type each line out.
4. They can take an input and usually produce some output.
5. You call a function by using its name followed by empty parenthesis or its arguments in parenthesis.

Functions require a function signature.

Function signatures tell the user how the function should be called. They start with the `def` keyword, indicating to the Python interpreter that we're defining a function. Next comes a space along with the the name of the function, and then an open and closed parenthesis. If the function takes any inputs, we define these inside the parenthesis and separate each one with a comma, except the last. We call these parameters.

The names of parameters should describe what they are used for, while the names of functions should be descriptive enough for the user to understand what it does. Best practice: a function name should be a verb or an action and arguments names should be nouns.

For example:

```
def add_two_numbers(num1, num2):
```

They do something useful.

Within the function itself something useful should happen, otherwise you have to question, "Why does the function exist in the first place?"

In our `add_two_numbers()` function, we could, as the name describes, add two numbers:

```
def add_two_numbers(num1, num2):
    sum = num1 + num2
```

They allow us re-use code without having to type each line out.

We can use this function as many times as we want, passing in two numbers, which will then get combined.

They take an input and usually produce some output.

In the above function, we pass in two numbers as input parameters, but we don't return anything. Therefore, we don't get any output. Let's fix that by returning the sum:

```
def add_two_numbers(num1, num2):
    sum = num1 + num2
    return sum
```

You call a function by using its name followed by its arguments in parenthesis.

We can call this function as many times as we'd like, providing different arguments each time to produce new output:

```
>>> def add_two_numbers(num1, num2):
...     sum = num1 + num2
...     return sum
...
>>> print(add_two_numbers(1,1))
2
>>> print(add_two_numbers(1,2))
3
>>> print(add_two_numbers(-20,9))
-11
>>>
```

Try creating your own functions that do something useful following these steps.

Review exercise:

1. Write a `cube()` function that takes a number and multiplies that number by itself twice over, returning the new value; test the function by displaying the result of calling your `cube()` function on a few different numbers
2. Write a function `multiply()` that takes two numbers as inputs and multiplies them

together, returning the result; test your function by saving the result of `multiply(2, 5)` in a new variable and printing it

Assignment: Convert temperatures

Write a script *temperature.py* that includes two functions. One function takes a Celsius temperature as its input, converts that number into the equivalent Fahrenheit temperature and returns that value. The second function takes a Fahrenheit temperature and returns the equivalent Celsius temperature. Test your functions by passing input values to them and printing the output results.

For testing your functions, example output should look like:

```
72 degrees F = 22.2222222222 degrees C
37 degrees C = 98.6 degrees F
```

In case you didn't want to spend a minute searching the web or doing algebra (the horror!), the relevant conversion formulas are:

1. F = C * 9/5 + 32
2. C = (F - 32) * 5/9

Run in circles

One major benefit of computers is that we can make them do the same exact thing over and over again, and they rarely complain or get tired. The easiest way to program your computer to repeat itself is with a loop.

There are two kinds of loops in Python: **for loops** and **while loops**. The basic idea behind any kind of loop is to run a section of code repeatedly as long as a specific statement (called the test condition) is true. For instance, try running this script that uses a while loop:

```
n = 1
while (n < 5):
    print("n =", n)
    n = n + 1
print("Loop finished")
```

Here we create a variable `n` and assign it a value of 1. Then we start the `while` loop, which is organized in a similar way to how we defined a function. The statement that we are testing comes in parentheses after the `while` command; in this case we want to know if the statement `n < 5` is `true` or `false`. Since `1 < 5`, the statement is `true` and we enter into the loop after the colon.

> **NOTE**: Notice the indentation on the lines after the colon. Just like when we defined a function, this spacing is important The colon and indenting let Python know that the next lines are inside the while loop. As soon as Python sees a line that isn't indented, that line and the rest of the code after it will be considered outside of the loop.

Once we've entered the loop, we print the value of the variable `n`, then we add 1 to its value. Now `n` is 2, and we go back to test our `while` statement. This is still true, since `2 < 5`, so we run through the next two lines of code again… And we keep on with this pattern while the statement `n < 5` is `true`.

As soon as this statement becomes `false`, we're completely done with the loop; we jump straight to the end and continue on with the rest of the script, in this case printing out "Loop finished "

Go ahead and test out different variations of this code and try to guess what your output will be before you run each script. Just be careful: it's easy to create what's called an **infinite loop**; if you test a statement that's always going to be true, you will never break out of the loop, and your code will just keep running forever.

> **NOTE**: It's important to be consistent with indentation, too. Notice how you can hit tab and backspace to change indentation, and IDLE automatically inserts four space characters. That's because you can't mix tabs and spaces as indentation. Although IDLE won't let you make this mistake, if you were to open your script in a different text editor and replace some of the space indentation with tabs, Python would get confused - even though the spacing looks the same to you.

The second type of loop, a `for` loop, is slightly different in Python. We typically use `for` loops in Python in order to loop over every individual item in a set of similar things - these things could be numbers, variables, lines from an input file, etc.

For instance, the following code does the exact same thing as our previous script by using a `for` loop to repeatedly run code for a range of numbers:

```
for n in range(1, 5):
    print("n =", n)
print("Loop finished")
```

Here we are using the `range()` function, which is a function that is built into Python, to return a list of numbers. The `range()` function takes two input values, a starting number and a stopping number. So in the first line, we create a variable `n` equal to 1, then we run through the code inside of our loop for `n = 1`.

We then run through the loop again for `n = 2`, etc., all the way through our range of numbers. Once we get to the end of the range, we jump out of the loop. Yes, it's slightly counter-intuitive, but in Python a `range(x, y)` starts at x but ends right before y.

> **NOTE**: When we use a `for` loop, we don't need to define the variable we'll be looping over first. This is because the for loop reassigns a new value to the variable each time we run through the loop. In the above example, we assigned the value 1 to the object `n` as soon as we started the loop. This is different from a while loop, where the first thing we do is to test some condition - which is why we need to have given a value to any variable that appears in the while loop before we enter the loop.

Play around with some variations of this script to make sure you understand how the `for` loop is structured. Again, be sure to use the exact same syntax, including the `in` keyword, the colon, and the proper indentation inside the loop.

Don't just copy and paste the sample code into a script - type it out yourself. Not only will that help you learn the concepts, but the proper indentation won't copy over correctly anyway…

As long as we indent the code correctly, we can even put loops inside loops.

Try this out:

```
for n in range(1, 4):
    for j in ["a", "b", "c"]:
        print("n =", n, "and j =", j)
```

Here `j` is looping over a list of strings; we'll see how lists work in a later chapter, but this example is only to show that you don't have to use the `range()` function with a `for` loop. Since the `j` loop is inside the `n` loop, we run through the entire `j` loop (`j="a", j="b", j="c"`) for each value that gets assigned to `n` in the outer loop.

We will use `for` loops in future chapters as an easy way to loop over all the items stored in many different types of objects - for instance, all the lines in a file.

> **NOTE**: Once your code is running in the interactive window, sometimes you might accidentally end up entering a loop that takes much longer than you expected. If that's the case (or if your code seems "frozen" because of anything else that's taking longer than expected), you can usually break out of the code by typing CTRL+C in the interactive window. This should immediately stop the rest of your script from running and take you back to a prompt in the interactive window.

If that doesn't seem to have any effect (because you somehow managed to freeze the IDLE window), you can usually type CTRL+Q to quit out of IDLE entirely, much like "End Task" in Windows or "Force Quit" on a Mac.

Review exercises:

1. Write a `for` loop that prints out the integers 2 through 10, each on a new line, by using the `range()` function
2. Use a `while` loop that prints out the integers 2 through 10 (Hint: you'll need to create a new integer first; there isn't a good reason to use a while loop instead of a `for` loop in this case, but it's good practice...)

3. Write a function `doubles()` that takes one number as its input and doubles that number three times using a loop, displaying each result on a separate line; test your function by calling `doubles(2)` to display 4, 8, and 16

Assignment: Track your investments

Write a script *invest.py* that will track the growing amount of an investment over time. This script includes an `invest()` function that takes three inputs: the initial investment amount, the annual compounding rate, and the total number of years to invest. So, the first line of the function will look like this:

```
def invest(amount, rate, time):
```

The function then prints out the amount of the investment for every year of the time period.

In the main body of the script (after defining the function), use the following code to test your function:

```
invest(100, .05, 8)
invest(2000, .025, 5)
```

Running this test code should produce the following output exactly:

```
principal amount: $100
annual rate of return: 0.05
year 1: $105.0
year 2: $110.25
year 3: $115.7625
year 4: $121.550625
year 5: $127.62815625
year 6: $134.009564063
year 7: $140.710042266
year 8: $147.745544379

principal amount: $2000
annual rate of return: 0.025
year 1: $2050.0
year 2: $2101.25
year 3: $2153.78125
year 4: $2207.62578125
year 5: $2262.81642578
```

> **NOTE**: Although functions usually return output values, this is entirely optional in Python. Functions themselves can print output directly as well. You can print as much as you like from inside a function, and the function will continue running until you reach its end.*

Some additional pointers if you're stuck:

1. You will need to use a `for` loop over a range that's based on the function's time input.
2. Within your loop, you will need to reassign a new value to the amount every year based on how it grows at 1 + rate.
3. Remember that you need to use either the string `format()` method or `str()` to convert numbers to strings first if you want to print both strings and numbers in a single statement.
4. Using the `print()` function without passing any arguments will print a blank line.

Questions?

Please email *info@realpython.com* with any questions.

Again, if you like what you see, consider purchasing the entire course on RealPython.com - for just $60, you will get all three courses, with over 1,200 pages of content, packed with exercises, sample files, assignments, and videos!

Click here to purchase on RealPython.com.

Please share this document with anyone else who might benefit from learning practical programming.

Cheers!