Functions & Modules

Reusing Code

Code reuse is a very important part of programming in any language. Increasing code size makes it harder to maintain.
For a large programming project to be successful, it is essential to abide by the **Don't Repeat Yourself**, or **DRY**, principle. We've already looked at one way of doing this: by using loops. In this module, we will explore two more: functions and modules.

> Bad, repetitive code is said to abide by the **WET** principle, which stands for **Write Everything Twice**, or **We Enjoy Typing**.

Functions

You've already used **functions** in previous lessons.
Any statement that consists of a word followed by information in **parentheses** is a function call.
Here are some examples that you've already seen:

```
print("Hello world!")
range(2, 20)
str(12)
range(10, 20, 3)
```

> The words in front of the parentheses are function **names**, and the comma-separated values inside the parentheses are function **arguments**.

Functions

In addition to using pre-defined functions, you can create your own functions by using the **def** statement.
Here is an example of a function named **my_func**. It takes no arguments, and prints "spam" three times. It is defined, and then called. The statements in the function are executed only when the function is called.

```
def my_func():
  print("spam")
  print("spam")
  print("spam")

my_func()
```

Try It Yourself

Result:

```
>>>
spam
spam
spam
>>>
```

The code block within every function starts with a colon (:) and is **indented**.

Functions

You must define functions before they are called, in the same way that you must assign variables before using them.

```
hello()

def hello():
  print("Hello world!")
```

Try It Yourself

Result:

```
>>>
NameError: name 'hello' is not defined
>>>
```

Tap **Try It Yourself** to play around with the code!

Arguments

All the function definitions we've looked at so far have been functions of zero arguments, which are called with empty parentheses.
However, most functions take arguments.
The example below defines a function that takes one argument:

```
def print_with_exclamation(word):
  print(word + "!")

print_with_exclamation("spam")
print_with_exclamation("eggs")
print_with_exclamation("python")
```

Try It Yourself

Result:

```
>>>
spam!
eggs!
python!
>>>
```

As you can see, the argument is defined inside the parentheses.

Arguments

You can also define functions with more than one argument; separate them with commas.

```
def print_sum_twice(x, y):
  print(x + y)
  print(x + y)

print_sum_twice(5, 8)
```

Try It Yourself

Result:

```
>>>
13
13
>>>
```

Tap **Try It Yourself** to play around with the code!

Arguments

Function arguments can be used as variables inside the function definition. However, they cannot be referenced outside of the function's definition. This also applies to other variables created inside a function.

```
def function(variable):
  variable += 1
  print(variable)

function(7)
print(variable)
```

Try It Yourself

Result:

```
>>>
8

NameError: name 'variable' is not defined
>>>
```

Technically, **parameters** are the variables in a function definition, and **arguments** are the values put into parameters when functions are called.

Returning from Functions

Certain functions, such as **int** or **str**, return a value that can be used later.
To do this for your defined functions, you can use the **return** statement.

For example:

```
def max(x, y):
    if x >=y:
        return x
    else:
        return y

print(max(4, 7))
z = max(8, 5)
print(z)
```

Try It Yourself

Result:

```
>>>
7
8
>>>
```

The **return** statement cannot be used outside of a function definition.

Returning from Functions

Once you return a value from a function, it immediately stops being executed. Any code after the **return** statement will never happen.
For example:

```
def add_numbers(x, y):
  total = x + y
  return total
  print("This won't be printed")

print(add_numbers(4, 5))
```

Try It Yourself

Result:

```
>>>
9
>>>
```

Tap **Try It Yourself** to play around with the code!

Comments

Comments are annotations to code used to make it easier to understand. They don't affect how code is run.
In Python, a comment is created by inserting an **octothorpe** (otherwise known as a number sign or hash symbol: #). All text after it on that line is ignored.
For example:

```
x = 365
y = 7
# this is a comment

print(x % y) # find the remainder
# print (x // y)
# another comment
```

Try It Yourself

Result:

```
>>>
1
>>>
```

Python doesn't have general purpose multiline comments, as do programming languages such as C.

Docstrings

Docstrings (documentation strings) serve a similar purpose to comments, as they are designed to explain code. However, they are more specific and have a different syntax. They are created by putting a multiline string containing an explanation of the function below the **function's first line**.

```
def shout(word):
    """
    Print a word with an
    exclamation mark following it.
    """
    print(word + "!")

shout("spam")
```

Try It Yourself

Result:

```
>>>
spam!
>>>
```

Unlike conventional comments, **docstrings** are retained throughout the runtime of the program. This allows the programmer to inspect these comments at run time.

Functions

Although they are created differently from normal variables, **functions** are just like any other kind of value.
They can be assigned and reassigned to variables, and later referenced by those names.

```
def multiply(x, y):
  return x * y

a = 4
b = 7
operation = multiply
print(operation(a, b))
```

Try It Yourself

Result:

```
>>>
28
>>>
```

The example above assigned the function **multiply** to a variable **operation**. Now, the name **operation** can also be used to call the function.

Functions

Functions can also be used as **arguments** of other functions.

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

def do_twice(func, x, y):
  return func(func(x, y), func(x, y))

a = 5
b = 10

print(do_twice(add, a, b))
```

Try It Yourself

Result:

```
>>>
30
>>>
```

As you can see, the function **do_twice** takes a function as its argument and calls it in its body.

Modules

Modules are pieces of code that other people have written to fulfill common tasks, such as generating random numbers, performing mathematical operations, etc.

The basic way to use a module is to add **import module_name** at the top of your code, and then using **module_name.var** to access functions and values with the name **var** in the module.
For example, the following example uses the **random** module to generate random numbers:

```
import random

for i in range(5):
  value = random.randint(1, 6)
  print(value)
```

Try It Yourself

Result:

```
>>>
2
3
6
5
4
>>>
```

The code uses the **randint** function defined in the **random** module to print 5 random numbers in the range 1 to 6.

Modules

There is another kind of **import** that can be used if you only need certain functions from a module. These take the form **from module_name import var**, and then **var** can be used as if it were defined normally in your code.
For example, to import only the **pi** constant from the **math** module:

```
from math import pi

print(pi)
```

Try It Yourself

Result:

```
>>>
3.141592653589793
>>>
```

Use a comma separated list to import multiple objects. For example:

```
from math import pi, sqrt
```

* imports all objects from a module. For example: **from math import ***
This is generally discouraged, as it confuses variables in your code with variables in the external module.

Modules

Trying to import a module that isn't available causes an ImportError.

```
import some_module
```

Result:

```
>>>
ImportError: No module named 'some_module'
>>>
```

Trying to import a module that isn't available causes an ImportError.

Modules

You can import a module or object under a different name using the **as** keyword. This is mainly used when a module or object has a long or confusing name.
For example:

```
from math import sqrt as square_root
print(square_root(100))
```

Try It Yourself

Result:

```
>>>
10.0
>>>
```

Tap **Try It Yourself** to play around with the code!

Modules

There are three main types of modules in Python, those you write yourself, those you install from external sources, and those that are preinstalled with Python.
The last type is called the **standard library**, and contains many useful modules. Some of the standard library's useful modules include string, **re**, **datetime**, **math**, **random**, **os**, **multiprocessing**, **subprocess**, **socket**, **email**, **json**, **doctest**, **unittest**, **pdb**, **argparse** and **sys**.

Tasks that can be done by the standard library include string parsing, data serialization, testing, debugging and manipulating dates, emails, command line arguments, and much more!

Python's extensive standard library is one of its main strengths as a language.

The Standard Library

Some of the modules in the standard library are written in Python, and some are written in C. Most are available on all platforms, but some are Windows or Unix specific.

We won't cover all of the modules in the standard library; there are simply too many. The complete documentation for the standard library is available online at www.python.org.

Modules

Many third-party Python modules are stored on the **Python Package Index** (PyPI).
The best way to install these is using a program called **pip**. This comes installed by default with modern distributions of Python. If you don't have it, it is easy to install online. Once you have it, installing libraries from PyPI is easy. Look up the name of the library you want to install, go to the command line (for Windows it will be the Command Prompt), and enter **pip install library_name**. Once you've done this, import the library and use it in your code.

Using **pip** is the standard way of installing libraries on most operating systems, but some libraries have prebuilt binaries for Windows. These are normal executable files that let you install libraries with a GUI the same way you would install other programs.

> It's important to enter **pip** commands at the command line, not the Python interpreter.

End.