Lesson 4: Functions

Functions are reusable pieces of programs. They allow you to give a name to a block of statements, allowing you to run that block using the specified name anywhere in your program and any number of times. This is known as *calling* the function. We have already used a few built-in functions such as `type`, `int`, `float`, and `format`.

Python provides a number of important built-in functions that we can use without needing to provide the function definition. The creators of Python wrote a set of functions to solve common problems and included them in Python for us to use. Here are a few examples:

The min and max Functions

The `max` and `min` functions give us the largest and smallest values in a list, respectively:

Code	Output
`max('Hello world')`	`w`
`max([10,9,3,5,7,2])`	`10`
`min('Hello world')`	*(space)*
`min([10,9,3,5,7,2])`	`2`

The `max` function tells us the "largest character" in the string (which turns out to be the letter "w") and the `min` function shows us the smallest character (which turns out to be a space).

The len Function

Another very common built-in function is the `len` function which tells us how many items are in its argument. If the argument to `len` is a string, it returns the number of characters in the string. If the argument is a sequence, then it counts the number of objects in the sequence.

Code	Output
`len('Hello world')`	`11`
`len([10,9,3,5,7,2])`	`6`

The function concept is probably the most important building block of any non-trivial software (in any programming language), so we will explore various aspects of functions in this chapter.

Defining Functions

In Python, functions are ***defined*** using the `def` keyword. After this keyword comes an identifier name for the function, followed by a pair of parentheses which may enclose some names of variables, and by the final colon that ends the line. Next follows the block of statements that are part of this function.

Just like with conditional statements and iterations, the first line of the function definition is called the ***header***; the rest is called the ***body***. The header has to end with a colon and the body has to be indented. By convention, the indentation is always four spaces. The body can contain any number of statements.

```
def [function_name]():
    [statements]
```

After we define the function, we can ***call*** it by name – and even pass information into it.

An example will show that this is actually very simple:

Example: Say Hello Function

```
def say_hello():
    # block belonging to the function
    print('hello world')
# End of function

say_hello()  # call the function
say_hello()  # call the function again
```

Output:

```
hello world
hello world
```

How It Works

We define a function called `say_hello` using the syntax as explained above. This function takes no parameters and hence there are no variables declared in the parentheses. Parameters to functions are just input to the function so that we can pass in different values to it and get back corresponding results.

Notice that we can call the same function twice which means we do not have to write the same code again.

Go online to complete the
Calling a Function Challenge

Here is another example:

Code	Output

`# Here we define the function`
`def print_lyrics():`
`    print("I'm a lumberjack, ")`
`    print("and I'm okay.")`
 	
`# Here we call the function`
	`I'm a lumberjack,`
	`and I'm okay.`

```
print_lyrics()
```

Once you have defined a function, you can use it inside another function. For example, to repeat the previous refrain, we could write an additional function called `repeat_lyrics`:

Code	Output

`# Here we define the second function`
`def repeat_lyrics():`
`    print_lyrics()`
`    print_lyrics()`
 	
`# Here we call the second function`
`repeat_lyrics ()`
	`I'm a lumberjack,`
	`and I'm okay.`
	`I'm a lumberjack,`
	`and I'm okay.`

Pulling together the code fragments from the previous section, the whole program looks like this:

Code	Output

`# Here we define the first function`
`def print_lyrics():`
`    print("I'm a lumberjack, ")`
`    print("and I'm okay.")`
 	
`# Here we define the second function`
`def repeat_lyrics():`
`    print_lyrics()`
`    print_lyrics()`
 	
`# Here we call the second function`
`repeat_lyrics ()`
	`I'm a lumberjack,`
	`and I'm okay.`
	`I'm a lumberjack,`
	`and I'm okay.`

Our program contains two function definitions: `print_lyrics` and `repeat_lyrics`. Function definitions get executed just like other statements, but the effect is to create function objects. The statements inside the function do not get executed until the function is called, and the function definition generates no output.

As you might expect, you have to create a function before you can execute it. In other words, the function definition has to be executed before the first time it is called.

Go online to complete the
Call a Function within a Function Challenge

Go online to complete the
Function Problem Challenge

Flow of Execution

In order to ensure that a function is defined before its first use, you have to know the order in which statements are executed, which is called the ***flow of execution***.

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.

Function Parameters

Some of the built-in functions we have seen require *arguments*. For example, when you call `max` you pass a sequence as an argument. Some functions take more than one argument: `range()` can take three: the start, the stop, and the step.

Inside the function, the arguments are assigned to variables called ***parameters***. Parameters are specified within the pair of parentheses in the function definition, separated by commas. When we call the function, we supply the values in the same way.

Note the terminology used - the names given in the function definition are called ***parameters*** whereas the values you supply in the function call are called ***arguments***.

Here is an example of a user-defined function that has a *parameter* and takes an *argument*:

Code	Notes

`def print_twice(the_phrase):`
`  print(the_phrase)`
`  print(the_phrase)`
	This function assigns the argument to a parameter named `the_phrase`. When the function is called, it prints the value of the parameter (whatever it is) twice.

This function works with any value that can be printed.

Code	Output

`def print_twice(the_phrase):`
`  print(the_phrase)`
`  print(the_phrase)`
 	
`print_twice("Spam")`
`print_twice(17)`
	`Spam`
	`Spam`
	`17`
	`17`

The same rules of composition that apply to built-in functions also apply to user-defined functions, so we can use any kind of expression as an argument for print_twice:

Code	Output

`def print_twice(the_phrase):`
`    print(the_phrase)`
`    print(the_phrase)`
 	
`print_twice("Spam"*4)`
`print_twice(max("Spam!"))`
	SpamSpamSpamSpam
	SpamSpamSpamSpam
	p
	p

The argument is evaluated before the function is called, so in the examples the expressions `"Spam"*4` and `max("Spam!")` are only evaluated once.

You can also use a variable as an argument:

Code	Output

`def print_twice(the_phrase):`
`    print(the_phrase)`
`    print(the_phrase)`
 	
`status = "I'm not dead yet"`
 	
`print_twice(status)`
	I'm not dead yet
	I'm not dead yet

The name of the variable we pass as an argument (`status`) has nothing to do with the name of the parameter (`the_phrase`). It doesn't matter what the value was called back home (in the caller); here in `print_twice`, we call everything `the_phrase`.

Go online to complete the
Passing an Argument Challenge

Example: Function with Multiple Parameters

```
def print_max(a, b):
    if a > b:
        print(a, 'is maximum')
    elif a == b:
        print(a, 'is equal to', b)
    else:
        print(b, 'is maximum')

# directly pass literal values
print_max(3, 4)

x = 5
y = 7

# pass variables as arguments
print_max(x, y)
```

Output:

```
4 is maximum
7 is maximum
```

How It Works

Here, we define a function called `print_max` that uses two parameters called a and b. We find out the greater number using a simple `if-else` statement and then print the bigger number.

The first time we call the function `print_max`, we directly supply the numbers as arguments. In the second case, we call the function with variables as arguments. `print_max(x, y)` causes the value of argument x to be assigned to parameter a and the value of argument y to be assigned to parameter b. The `print_max` function works the same way in both cases.

Go online to complete the
Passing Multiple Arguments Challenge

Fruitful Functions and Void Functions

Some of the functions we are using, such as the `math` functions, yield results; you can call them ***fruitful functions*** or return-value function. Other functions, like `print_twice`, perform an action but don't return a value. They are called ***void functions***.

When you call a fruitful function, you almost always want to do something with the result; for example, you might assign it to a variable or use it as part of an expression:

Code

`x = math.cos(radians)`
 	
`golden = (math.sqrt(5) + 1) / 2`

When you call a function in interactive mode, Python displays the result:

```
>>> math.sqrt(5)
2.2360679774997898
```

But in a script, if you call a fruitful function and do not store the result of the function in a variable, the return value vanishes into the mist!

Code	Output
`math.sqrt(5)`	*Nothing*

This script computes the square root of 5, but since it doesn't store the result in a variable or display the result using the `print` statement, it is not very useful.

Void functions might display something on the screen or have some other effect, but they don't have a return value. If you try to assign the result to a variable, you get a special value called None.

Code	Output	Notes
`hi = print("Hello!")`	`Hello`	In this case, we have assigned the variable `hi` to `print("Hello")` The print statement performs an action. It displays the word `"Hello"` on screen. `print` does not return a value.
`print(hi)`	`None`	But when we try to display the value of the variable `hi`, we get `None` as a result.

The value `None` is not the same as the string `'None'`. It is a special value that has its own type:

Code	Output
`print(type(None))`	`<class 'NoneType'>`

To return a result from a function, we use the `return` statement in our function. The `return` statement is used to return from a function i.e. break out of the function. We can optionally return a value from the function as well.

For example, we could make a very simple function called `addtwo` that adds two numbers together and returns a result.

Code	Output

`def addtwo(a, b):`
`  added = a + b`
`  return added`
 	
`x = addtwo(3, 5)`
 	
`print(x)`
	`8`

When this script executes, the `print` statement will print out `"8"` because the `addtwo` function was called with 3 and 5 as arguments. Within the function, the parameters a and b were assigned to 3 and 5 respectively. The function computed the sum of the two numbers and placed it in the local function variable named `added`. Then it used the `return` statement to send the computed value back to the calling code as the function result, which was assigned to the variable `x` and printed out.

Example: Simple Return Statement

```
def maximum(x, y):
    if x > y:
        return x
```

```
    elif x == y:
        return 'The numbers are equal'
    else:
        return y

print(maximum(2, 3))
```

Output:

```
3
```

How It Works

The maximum function returns the maximum of the parameters, in this case the numbers supplied to the function. It uses a simple `if-else` statement to find the greater value and then returns that value.

Note that a return statement without a value is equivalent to return `None`. `None` is a special type in Python that represents nothingness. For example, it is used to indicate that a variable has no value if it has a value of `None`.

Every function implicitly contains a return `None` statement at the end unless you have written your own `return` statement. You can see this by running `print(some_function())` where the function `some_function` does not use the return statement such as:

```
def some_function():
    pass
```

As we have seen, the `pass` statement is used in Python to indicate an empty block of statements.

TIP: As we have seen, there is a built-in function called `max` that already implements the 'find maximum' functionality, so use this built-in function whenever possible.

Go online to complete the
Returning Values Challenge

More about Return Values

You aren't limited to returning numbers. You can return strings and Boolean values as well – and even other objects we haven't yet discovered like lists, tuples, and dictionaries just to name a few. Here are a few examples:

Code	Output

`def is_blue(color):`
`  if color == 'blue':`
`  return True`
`  else:`
`  return False`
 	
`color_check = is_blue('green')`
`print(color_check)`
	`False`

```
def display_twice(word):                    Hi!Hi!
    return word*2

message = display_twice("Hi!")
print(message)
```

Returning Multiple Values

A function can return more than one value by separating the individual return values by commas.

```
return value1, value2, value3, etc
```

When you call the function, you assign it to multiple variables as well:

Code	Output

`def name():`
`    first = "George"`
`    last = "Weasley"`
`    return first, last`
 	
`first_name, last_name = name()`
 	
`print(last_name)`
`print(first_name)`
	`Weasley`
	`George`

In this example, the function `name` returned two values. The line:

```
first_name, last_name = name()
```

assigned the first returned value to the variable `first_name` and the second returned value to the variable `last_name`. You have to pay attention to which order the values are returned so you assign them to the correct variables when you call the function.

Go online to complete the
Returning Multiple Values Challenge

Why Functions?

It may not be clear why it is worth the trouble to divide a program into functions. There are several reasons:

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read, understand, and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.

- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

Throughout the rest of this course, often we will use a function definition to explain a concept. Part of the skill of creating and using functions is to have a function properly capture an idea such as “find the smallest value in a list of values”.

Variable Scope

When you declare variables *inside* a function definition, they are not related in any way to other variables with the same names used *outside* the function - i.e. variable names are local to the function.

Code	Output

`my_name = "Fred"`
 	
`def print_name():`
`    my_name = "George"`
`    print(my_name)`
 	
`print_name()`
 	
`print(my_name)`
	`George`
	`Fred`

In the above example, we have a variable called `my_name` which is outside of the function `print_name()`. There is also a variable called `my_name` inside of `print_name()`. These are two different variables. We can see this when we run the last two lines:

Code	Output	
`print_name()`	`George`	This displays the variable *inside* of the function, which is assigned to the string `George`.
`print(my_name)`	`Heather`	This displays the variable outside of the function, which is assigned to the string `Heather`.

This is called the ***variable scope***. The scope of all variables is the block they are declared in - starting from the point of definition of the name. The scope of `my_name` that is outside of the function (let's call it the "outside `my_name`") is the main body of the program. The scope of the `my_name` variable that is inside of the function (let's call it the "inside `my_name`") is the function `print_name()`. The inside `my_name` is considered to be local to the `print_name()`function. Here's another example:

Example: Local Variable

```
x = 50

def func(x):
    print('x is', x)
```

```
        x = 2
        print('Changed local x to', x)

func(x)
print('x is still', x)
```

Output:

```
x is 50
Changed local x to 2
x is still 50
```

How It Works

The first time that we print the value of the name x with the first line in the function's body, Python uses the value of the parameter declared in the main block, above the function definition.

Next, we assign the value 2 to x. The name x is *local* to our function. So, when we change the value of x in the function, the x defined in the main block remains unaffected.

With the last `print` statement, we display the value of x as defined in the main block, thereby confirming that it is actually unaffected by the local assignment within the previously called function.

Go online to complete the Variable Scope Challenge

The global Statement

If you want to assign a value to a name defined at the top level of the program (i.e. not inside any kind of scope such as functions or classes), then you have to tell Python that the name is not *local*, but it is ***global***. We do this using the `global` statement. It is impossible to assign a value to a variable defined outside a function without the `global` statement.

Example: Global Statement

```
x = 50

def func():
    global x
    print('x is', x)
    x = 2
    print('Changed global x to', x)

func()
print('Value of x is', x)
```

Output:

```
x is 50
Changed global x to 2
Value of x is 2
```

How It Works

The `global` statement is used to declare that x is a global variable - hence, when we assign a value to x inside the function, that change is reflected when we use the value of x in the main block.

You can specify more than one global variable using the same global statement e.g. `global x, y, z`.

Go online to complete the
Using a Global Variable Challenge

Global Constants

You can use the values of such variables defined outside the function (assuming there is no variable with the same name within the function). However, this is not encouraged and should be avoided since it becomes unclear to the reader of the program as to where that variable's definition is. Using the `global` statement makes it amply clear that the variable is defined in an outermost block.

That being said, having a global variable can cause a lot of confusion if *its value is changed* by multiple functions. In general, it is better to pass variables into functions as arguments, rather than share a global variable across an entire program.

However if the value assigned to the global variable does not ever change, it is called a ***global constant*** and its use is less likely to cause problems. In Python, there is no technical difference between a global variable which changes its value and a global constant which maintains its value – you declare and use them the same way. To avoid confusion, convention dictates that global constants are named in all capital letters, while global variables follow whatever naming convention you like.

Code

```
# Global Constant Example
# The name of the game will never change
# This is okay to use because it never changes
GAME_NAME = "League of Leg Ends"

# Global Variable Example
# The score will constantly change and by more than one function
# this could get confusing and would be better to pass as an
# argument
score = 0

def win_round():
    score++
    print("You won the round!")
    print("Your score in", GAME_NAME, "is now", score)

def lose_round():
    score--
    print("You lost the round!")
    print("Your score in", GAME_NAME, "is now", score)
```

Default Argument Values

For some functions, you may want to make some parameters optional and use default values in case the user does not want to provide values for them. This is done with the help of default argument values. You can specify default argument values for parameters by appending to the parameter name in the function definition the assignment operator (=) followed by the default value.

Note that the default argument value should be a constant. More precisely, the default argument value should be immutable - this is explained in detail in later lessons. For now, just remember this.

Example: Default Argument

```
def say(message, times=1):
    print(message * times)

say('Hello')
say('World', 5)
```

Output:

```
Hello
WorldWorldWorldWorldWorld
```

How It Works

The function named `say` is used to print a string as many times as specified. If we don't supply a value, then by default, the string is printed just once. We achieve this by specifying a default argument value of 1 to the parameter `times`.

In the first usage of say, we supply only the string and it prints the string once. In the second usage of say, we supply both the string and an argument 5 stating that we want to say the string message 5 times.

CAUTION: Only those parameters which are at the end of the parameter list can be given default argument values i.e. you cannot have a parameter with a default argument value preceding a parameter without a default argument value in the function's parameter list.

This is because the values are assigned to the parameters by position. For example, `def func(a, b=5)` is valid, but `def func(a=5, b)` is not valid.

Go online to complete the
Default Argument Values Challenge

Keyword Arguments

If you have some functions with many parameters and you want to specify only some of them, then you can give values for such parameters by naming them - this is called keyword arguments - we use the name (keyword) instead of the position (which we have been using all along) to specify the arguments to the function.

There are two advantages - one, using the function is easier since we do not need to worry about the order of the arguments. Two, we can give values to only those parameters to which we want to, provided that the other parameters have default argument values.

Example: Keyword Arguments:

```
def func(a, b=5, c=10):
    print('a is', a, 'and b is', b, 'and c is', c)

func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

Output:

```
a is 3 and b is 7 and c is 10
a is 25 and b is 5 and c is 24
a is 100 and b is 5 and c is 50
```

How It Works

The function named `func` has one parameter without a default argument value, followed by two parameters with default argument values.

In the first usage, `func(3, 7)`, the parameter a gets the value 3, the parameter b gets the value 7 and c gets the default value of 10.

In the second usage `func(25, c=24)`, the variable a gets the value of 25 due to the position of the argument. Then, the parameter c gets the value of 24 due to naming i.e. keyword arguments. The variable b gets the default value of 5.

In the third usage `func(c=50, a=100)`, we use keyword arguments for all specified values. Notice that we are specifying the value for parameter c before that for a even though a is defined before c in the function definition.

Go online to complete the
Keyword Arguments Challenge

DocStrings

Python has a nifty feature called ***documentation strings***, usually referred to by its shorter name ***docstrings***. DocStrings are an important tool that you should make use of since it helps to document the program better and makes it easier to understand.

Basically, docstrings document the function so you can see what it does and any other information about it.

Amazingly, we can even get the docstring back from, say a function, when the program is actually running!

Example: DocStrings

```
def print_max(x, y):
    '''Prints the maximum of two numbers.

    The two values must be integers.'''

    # convert to integers, if possible
    x = int(x)
    y = int(y)

    if x > y:
        print(x, 'is maximum')
    else:
        print(y, 'is maximum')

print_max(3, 5)
print(print_max.__doc__)
```

Output:

```
5 is maximum
Prints the maximum of two numbers.
    The two values must be integers.
```

How It Works

A string on the first logical line of a function is the *docstring* for that function.

The convention followed for a docstring is a multi-line string where the first line starts with a capital letter and ends with a dot. Then the second line is blank followed by any detailed explanation starting from the third line. You are strongly advised to follow this convention for all your docstrings for all your non-trivial functions.

We can access the docstring of the `print_max` function using the __doc__ (notice the double underscores) attribute (name belonging to) of the function. Just remember that Python treats everything as an object and this includes functions.

If you have used `help()` in Python, then you have already seen the usage of docstrings! What it does is just fetch the __doc__ attribute of that function and displays it in a neat manner for you. You can try it out on the function above - just include `help(print_max)` in your program. Remember to press the q key to exit help.

Automated tools can retrieve the documentation from your program in this manner, which makes it really simple to write up documentation for your application. Therefore, I strongly recommend that you use docstrings for any non-trivial function that you write.

Go online to complete the DocStrings Challenge

Modules

You have seen how you can reuse code in your program by defining functions once. What if you wanted to reuse a number of functions in other programs that you write? As you might have guessed, the answer is modules.

There are various methods of writing modules, but the simplest way is to create a file with a `.py` extension that contains functions and variables.

Another method is to write the modules in the native language in which the Python interpreter itself was written. For example, you can write modules in the C programming language and when compiled, they can be used from your Python code when using the standard Python interpreter.

A module can be imported by another program to make use of its functionality. This is how we can use the Python standard library as well. First, we will see how to use the standard library modules.

Example: Module

```
import math

print('The value of pi is', math.pi)
print('The square root of 4 is', math.sqrt(4))
```

Output:

```
The value of pi is 3.141592653589793
The square root of 4 is 2.0
```

How It Works

First, we import the `math` module using the `import` statement. Basically, this translates to us telling Python that we want to use this module. The `math` module contains various mathematical functions – like `sin`, `cos`, and `tan` – as well as constants such as `pi` and `e`.

When Python executes the `import math` statement, it looks for the `math` module. In this case, it is one of the built-in modules, and hence Python knows where to find it.

If it was not a compiled module i.e. a module written in Python, then the Python interpreter will search for it in the directories listed in system path which was set up when Python was installed. If the module is found, then the statements in the body of that module are run and the module is made available for you to use. Note that the initialization is done only the first time that we import a module.

The `pi` variable in the `math` module is accessed using the dotted notation i.e. `math.pi`. It clearly indicates that this name is part of the `math` module. Another advantage of this approach is that the name does not clash with any `pi` variable used in your program.

The `math.pi` variable is a 15-decimal approximation of the value of pi π.

The `sqrt()` function in the `math` module calculates the square root of any integer or floating point number entered as an argument. Just like with pi, it is accessed using the dotted notation of `math.sqrt()`.

Go online to complete the
Using the math Module Challenge

Example: More from the Math Module

Code	Output	Notes

`import math`
 	
`signal_power = 100`
`noise_power = 17`
 	
`ratio = signal_power/noise_power`
`decibels = 10 *`
`math.log10(ratio)`
 	
`print(decibels)`
	`7.695510786217261`	This example uses the function `math.log10()` to compute the logarithm base 10 of the signal-to-noise ratio.

`import math`
 	
`degrees = 45`
 	
`radians = degrees/360*2*math.pi`
 	
`print(math.sin(radians))`
	`0.7071067811865475`	The second example finds the sine of 45 degrees. First, it converts the degrees to radians (divide by 360 and multiply by 2 π) and then it uses the `math.sin()` function to calculate sine.
		
		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.

Here is a selection of the various functions available within the `math` module.

Function	Description	Example Code	Example Output
`ceil(x)`	Return the ceiling of x as an Integral. This is the smallest integer >= x.	`math.ceil(4.5)`	`5`
`cos(x)`	Return the cosine of x (measured in radians).	`math.cos(math.pi)`	`-1.0`
`degrees(x)`	Convert angle x from radians to degrees.	`math.degrees(math.pi)`	`180.0`

factorial(x)	Find x!. Raise a ValueError if x is negative or non-integral.	`math.factorial(5)`	`120`
floor(x)	Return the floor of x as an Integral. This is the largest integer <= x.	`math.floor(4.5)`	`4`
gcd(x, y)	Returns the greatest common divisor of x and y	`math.gcd(28,91)`	`7`
log(x[, base])	Return the logarithm of x to the given base. If the base not specified, returns the natural logarithm (base e) of x.	`math.log(10,2)`	`3.3219280948873626`
log10(x)	Return the base 10 logarithm of x.	`math.log10(2)`	`0.3010299956639812`
pow(x, y)	Return x**y (x to the power of y).	`math.pow(2,10)`	`1024.0`
radians(x)	Convert angle x from degrees to radians.	`math.radians(180)`	`3.141592653589793`
sin(x)	Return the sine of x (measured in radians).	`math.sin(1.777)`	`0.9788152469968777`
sqrt(x)	Return the square root of x.	`math.sqrt(10)`	`3.1622776601683795`
tan(x)	Return the tangent of x (measured in radians).	`math.tan(1.777)`	`-4.780643944967509`

The from-import statement

If you want to directly import a specific variable or function into your program (to avoid having to type the dot notation every time), then you can use the `from-import` statement.

```
from [module] import [function or variable]
```

Code	Output
`from math import sqrt` `print('The square root of 8 is', sqrt(8))`	`The square root of 8 is 2.8284271247461903`
`from math import pi` `print('pi * 2 =', (pi*2))`	`pi * 2 = 6.283185307179586`

Notice that you do not need to type `math` and a dot in front of `pi` or `sqrt()`. This is great if you are lazy, but bad if you have your own variable called `pi` or your own function called `sqrt()`.

In general, avoid using the `from-import` statement, use the `import` statement instead. This way your program will avoid name clashes and will be more readable.

Go online to complete the
The from-import statement Challenge

Byte-compiled .pyc files

Importing a module is a relatively costly affair, so Python does some tricks to make it faster. One way is to create byte-compiled files with the extension `.pyc` which is an intermediate form that Python transforms the program into (remember the introduction section on how Python works?). This `.pyc` file is useful when you import the module the next time from a different program - it will be much faster since a portion of the processing required in importing a module is already done. Also, these byte-compiled files are platform-independent.

NOTE: These `.pyc` files are usually created in the same directory as the corresponding `.py` files. If Python does not have permission to write to files in that directory, then the `.pyc` files will not be created.

Making Your Own Modules

Creating your own modules is easy, you've been doing it all along! This is because every Python program is also a module. You just have to make sure it has a `.py` extension. The following example should make it clear.

Example: Creating My Module

```
def say_hi():
    print('Hi, this is mymodule speaking.')

word = 'Hi!'
```

We save this code in a file called `mymodule.py`.

The above was a sample module. As you can see, there is nothing particularly special about it compared to our usual Python program. We will next see how to use this module in our other Python programs.

Remember that the module should be placed either in the same directory as the program from which we import it, or in one of the directories listed in the system path. Now let's import our new module:

Example: Using My Module

```
import mymodule

mymodule.say_hi()
print(mymodule.__version__)
```

Output:

```
Hi, this is mymodule speaking.
Version 0.1
```

How It Works

Notice that we use the same dotted notation to access members of the module. Python makes good reuse of the same notation to give the distinctive 'Pythonic' feel to it so that we don't have to keep learning new ways to do things.

Example: Using My Module with from-import

```
from mymodule import say_hi, __version__

say_hi()
print('Version', __version__)
```

Output:

```
Hi, this is mymodule speaking.
Version 0.1
```

Look! The output is the same!

Notice that if there was already a `__version__` name declared in the module that imports `mymodule`, there would be a clash. This is also likely because it is common practice for each module to declare its version number using this name. Hence, it is always recommended to prefer the import statement even though it might make your program a little longer.

Zen of Python

One of Python's guiding principles is that "Explicit is better than Implicit". Run `import this` in Python to learn more.

Go online to complete the Zen of Python Challenge

The Random Module

Given the same inputs, most computer programs generate the same outputs every time, so they are said to be ***deterministic***. Determinism is usually a good thing, since we expect the same calculation to yield the same result. For some applications, though, we want the computer to be unpredictable. Games are an obvious example, but there are more.

Making a program truly nondeterministic turns out to be not so easy, but there are ways to make it at least seem nondeterministic. One of them is to use algorithms that generate pseudorandom numbers. Pseudorandom numbers are not truly random because they are generated by a deterministic computation, but just by looking at the numbers it is all but impossible to distinguish them from random.

The `random` module provides functions that generate pseudorandom numbers (which I will simply call "random" from here on).

The function `random` returns a random float between 0.0 and 1.0 (including 0.0 but not 1.0). Each time you call `random`, you get the next number in a long series.

Code	Output

`import random`
 	
`for i in range(10):`
`    x = random.random()`
`    print(x)`
	`0.301927091705`
	`0.513787075867`
	`0.319470430881`
	`0.285145917252`
	`0.839069045123`
	`0.322027080731`
	`0.550722110248`
	`0.366591677812`
	`0.396981483964`
	`0.838116437404`

The `random` function is only one of many functions that handle random numbers. The function `randint` takes the parameters `low` and `high`, and returns an integer between `low` and `high` (including both).

Code	Output

`import random`
 	
`for i in range(10):`
`    x = random.randint(5, 10)`
`    print(x)`
	5
	6
	9
	5
	6
	5
	10
	9
	7
	5

The random module also provides functions to generate random values from continuous distributions including Gaussian, exponential, gamma, and a few more.

Go online to complete the
Importing the random Module Challenge

Another useful function in the `random` module is `randrange`. The `randrange` function accepts parameters for `low`, `high`, and `step` – which allows you to generate a random list of numbers in specific increments.

Code	Output

`import random`
 	
`for i in range(10):`
`    x = random.randrange(100, 1500, 50)`
`    print(x)`
	250
	100
	250
	700
	1100
	600
	300
	1200
	650
	1450

In this example, we generated a list of numbers from (and including) 100 to (and excluding) 1500 in increments of 50.

Go online to complete the
Random Values of 20 Challenge

Mainline Logic

This next topic covers mainline logic which is extremely important, especially if you are using modules.

Without functions, we have seen that code will execute as soon as the program is loaded. Functions help to control when code is executed.

When you import a module into a program, the module's code also needs to be encased in functions – otherwise the module's program will execute as soon as you import it. Just like with the math module, you can call the individual functions contained within the module.

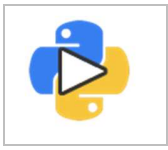
Since you can create a module out of any piece of code you write in Python, you also should encase your programs completely in functions. This is a way of controlling the *mainline logic* of the program.

Here is an example:

Code	Output

`def main():`
`  print("Hello!")`
`  secret_code()`
`  print("Bye!")`
 	
`def secret_code():`
`  print("Shh! It's a secret!")`
 	
`main()`
	`Hello!`
	`Shh! It's a secret!`
	`Bye!`

In this example, the main logic for the program is enclosed in the `main` function. This function performs actions, and calls a second function called `secret_code`.



Go online to complete the Mainline Logic Challenge

"But wait!" you say. By calling the `main` function (the last line), the mainline logic still executes if it is imported into another program. This is true. We need a way to automatically call the `main` function if we are running this module by itself, but to NOT call the `main` function if this code has been imported into another program. Here's how we do it:

Code	Output

`def main():`
`  print("Hello!")`
`  secret_code()`
`  print("Bye!")`
 	
`def secret_code():`
`  print("Shh! It's a secret!")`
 	
`if __name__ == "__main__":`
`  main()`
	`Hello!`
	`Shh! It's a secret!`
	`Bye!`

The __name__ variable is built-into Python. It tells the scope of the code which is being executed. If the code is running at the top level of the program, then __name__ is equal to `__main__`. If the code has been imported into the program, then the __name__ is equal to the name of the module. This allows us to automatically execute `main()` when we are running this code by itself – and when we are importing this code into another program, it will only execute the `main` function when we explicitly call it.

Moving forward as a best practice for all of your future programs:

1. You should define a `main` function
2. You should use the `if __name__ == "__main__":` conditional to execute the `main` function

Debugging

If you are using a text editor to write your scripts, you might run into problems with spaces and tabs. The best way to avoid these problems is to use spaces exclusively (no tabs). Most text editors that know about Python do this by default, but some don't.

Tabs and spaces are usually invisible, which makes them hard to debug, so try to find an editor that manages indentation for you.

Also, don't forget to save your program before you run it. Some development environments do this automatically, but some don't. In that case, the program you are looking at in the text editor is not the same as the program you are running.

Debugging can take a long time if you keep running the same incorrect program over and over!

Make sure that the code you are looking at is the code you are running. If you're not sure, put something like `print('hello')` at the beginning of the program and run it again. If you don't see `hello`, you're not running the right program!