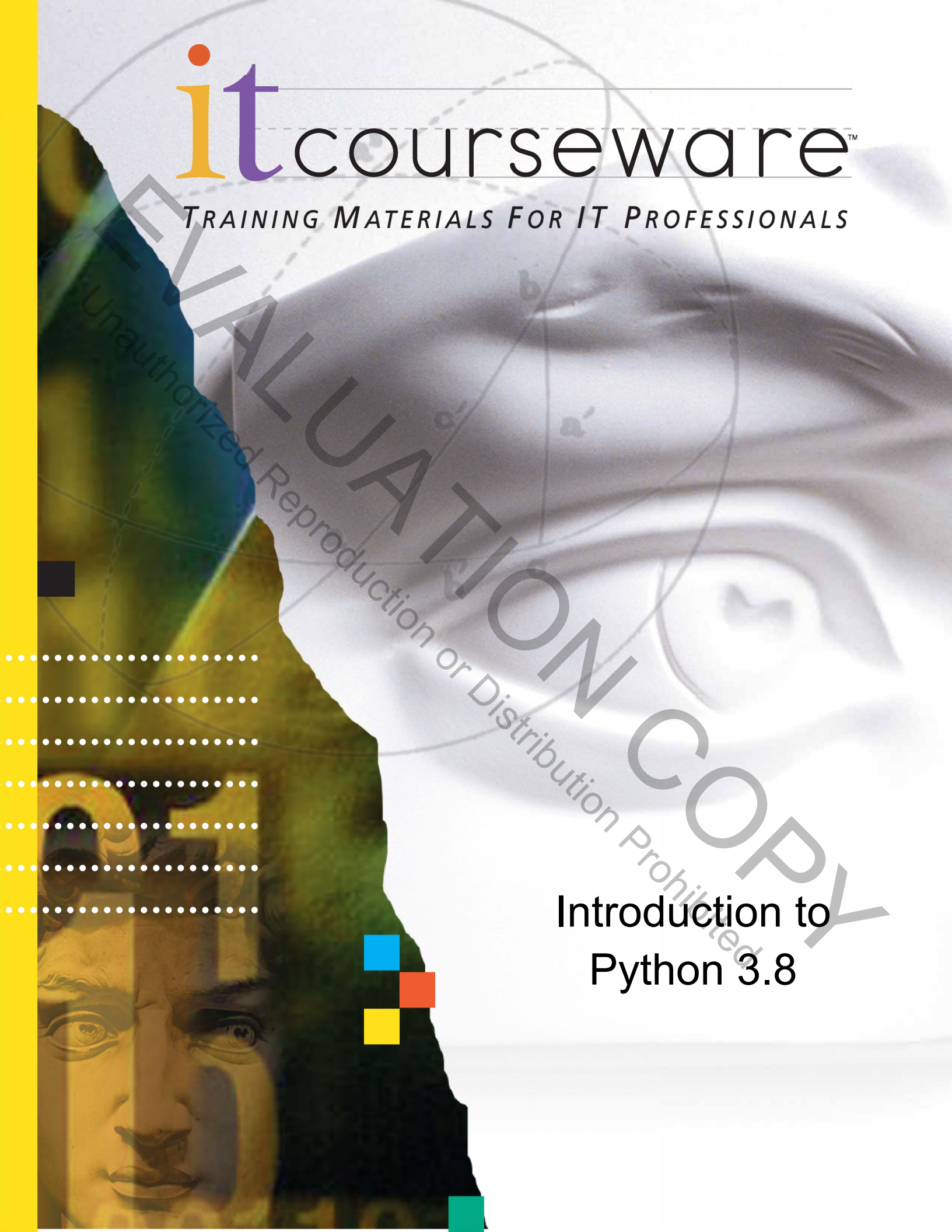

itcourseware™
TRAINING MATERIALS FOR IT PROFESSIONALS
EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited
Introduction to
Python 3.8

Introduction to Python 3.8

with examples and
hands-on exercises

WEBUCATOR

Version: PYT138.1.1.2

Class Files

Download the class files used in this manual at https://www.webucator.com/class-files/index.cfm?versionId=4775.

Errata

Corrections to errors in the manual can be found at https://www.webucator.com/books/errata.cfm.

Table of Contents

LESSON 1

Python Basics

Topics Covered

☑ How Python works.

☑ Python's place in the world of programming languages.

☑ Python literals.

☑ Python comments.

☑ Variables and Python data types.

☑ Simple modules.

☑ Outputting data with `print()`.

☑ Collecting user input.

> The pythons had entered into Mankind. No man knew at what moment he might be Possessed!
>
> *– Plague of Pythons, Frederik Pohl*

Introduction

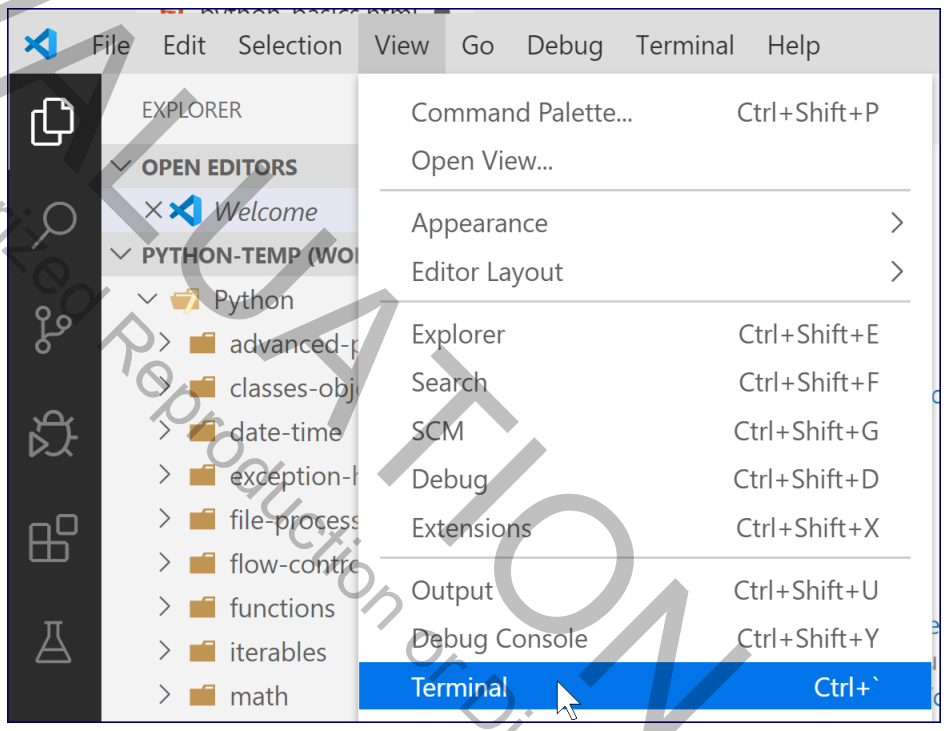
Python, which first appeared in 1991, is one of the most popular programming languages used today.[1] Python is a high-level programming language, meaning that it uses a syntax that is relatively human readable, which gets translated by a Python Interpreter into a language your computer can understand. Examples of other popular high-level programming languages are C#, Objective-C, Java, PHP, and JavaScript. Interestingly, all of these other languages, unlike Python, share a C-like syntax. If you use one or more of those languages, you may find Python's syntax a little strange. But give it a little time. You'll find it's quite programmer friendly.

1. https://pypl.github.io/PYPL.html

Getting Familiar with the Terminal

Python developers need to be comfortable navigating around and running commands at the terminal.[2] We'll walk you through some basics:

1. Open a terminal. In Visual Studio Code, you can open a terminal by selecting **Terminal** from the **View** menu:



You can also open a terminal by pressing **Ctrl+`**. The ` key is on the upper left of most keyboards. It looks like this:

The terminal will open at the root of your Visual Studio Code workspace. If you're working in the workspace we had you set up, that should be in a `Webucator\Python` directory (the

2. The generic term for the various terminals is *command line shell* (`https://en.wikipedia.org/wiki/Shell_(computing)`). Visual Studio Code will select an appropriate terminal for you. On Windows, that will most likely be **Command Prompt** or **PowerShell**. On a Mac, it will most likely be **bash** or **Zsh**. The names "prompt," "command prompt," "shell," and "terminal" are used interchangeably.

words *folder* and *directory* are used interchangeably). The prompt on Windows will read something like:

```
PS C:\Webucator\Python>
```

On a Mac, it will show some combination of your computer name, the directory you are in, and your username, followed by a `$` or `%` sign. For example:

```
NatsMac:Python natdunn$

natdunn@NatsMac:Python %
```

2. Use `cd` to **c**hange **d**irectories. From the `Webucator\Python` directory, run:

```
PS C:\Webucator\Python> cd python-basics
PS …\Python\python-basics>
```

Your prompt now shows that your are in the `python-basics` directory.

3. Move up to the parent directory by running:

```
cd ..
```

You will now be back in your `Python` directory.

4. Run:

```
cd python-basics/Demos
```

Your prompt will show that you are in the `Demos` directory. Depending on your environment, it may also show one or more directories above the `Demos` directory. To get the full path to your current location, run `pwd` for **p**resent **w**orking **d**irectory:

```
pwd
```

On Windows, that will look something like this:

```
PS …\python-basics\Demos> pwd

Path
----
C:\Webucator\Python\python-basics\Demos
```

On a Mac, it will look something like this:

```
natdunn@NatsMac Demos % pwd
/Users/natdunn/Documents/Webucator/Python/python-basics/Demos
```

5. Run `cd ..` to back up to the `python-basics` directory.
6. Type `cd De` and then press the **Tab** key. On Windows, you should see something like this:

```
PS …\Python\python-basics> cd .\Demos\
```

The "." at the beginning of the path represents the *current* (or *present*) directory. So, `.\Demos` refers to the `Demos` directory within the current directory. A Mac won't include the current directory in the path. When you press **Tab**, it will just fill out the rest of the folder name.

```
cd Demos
```

Press **Enter** to run the command. Then run `cd ..` to move back up to the `python-basics` directory.

7. Each of our lesson folders will contain `Demos`, `Exercises`, and `Solutions` folders. Some may contain additional folders. To see the contents of the current directory, run `dir` on Windows or `ls` on Mac/Linux:[3]

Windows PowerShell

```
PS …\Python\python-basics> dir


    Directory: C:\Webucator\Python\python-basics

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        2/18/2020   6:13 AM                data
d-----        2/18/2020   6:13 AM                Demos
d-----        2/18/2020   5:09 AM                Exercises
d-----        2/18/2020   6:13 AM                Solutions
```

Mac Terminal

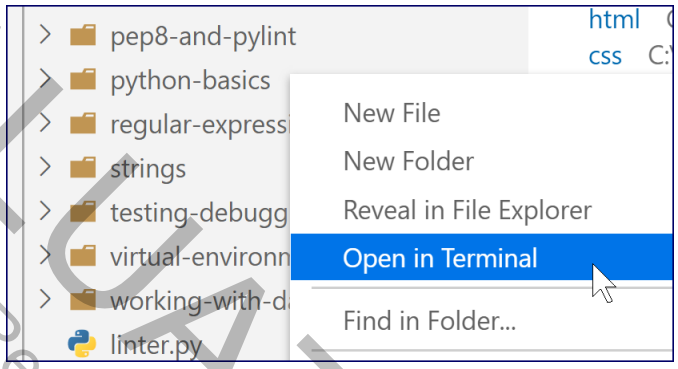
```
NatsMac:python-basics natdunn$ ls
Demos           Exercises       Solutions       data
```

3. The `ls` command works in Windows PowerShell as well.

8. Play with switching between directories using the `cd` command until you feel comfortable navigating the terminal.

Visual Studio Shortcut

Visual Studio provides a shortcut for opening a specific directory at the terminal. Simply right-click on the directory in the **Explorer** panel and select **Open in Terminal**:



Running Python

Python runs on Microsoft Windows, Mac OS X, Linux, and other Unix-like systems. The first thing to do is to make sure you have a recent version of Python installed:

1. Open the terminal in Visual Studio Code.
2. Run `python -V`:

```
PS C:\Webucator\Python> python -V
Python 3.8.1
```

If you have Python 3.6 or later, you are all set.

If you do not have Python 3.6 or later installed, download it for free at `https://www.python.org/downloads`. After running through the installer, run `python -V` at the terminal again to make sure Python installed correctly.

Python Versions on Macs

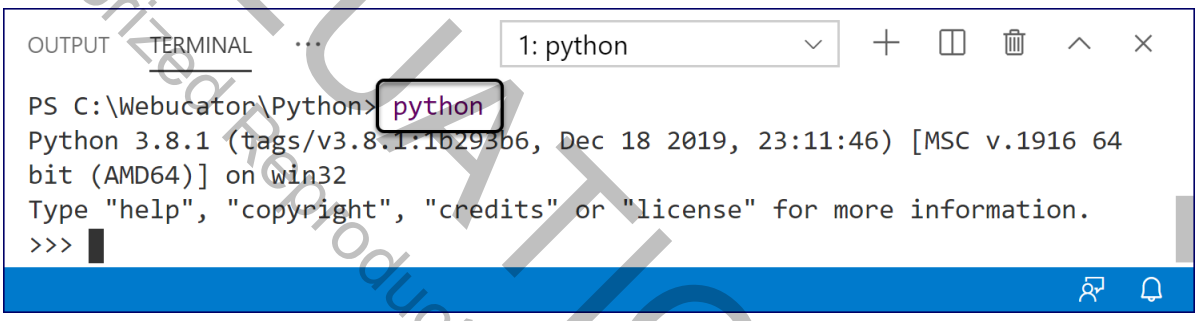
Your Mac will likely have a version of Python 2 already installed. After you install Python 3, you may find that running `python -V` still shows the Python 2 version. In that case, try running

`python3 -V`. That should output the version of Python 3 that you have. If it does, then you should use the `python3` command instead of the `python` command to run Python 3.

If you would prefer to be able to use the `python` command for Python 3 (and who wouldn't), visit `https://www.webucator.com/blog/2020/02/mapping-python-to-python-3-on-your-mac/` to see how you can map `python` to `python3`.

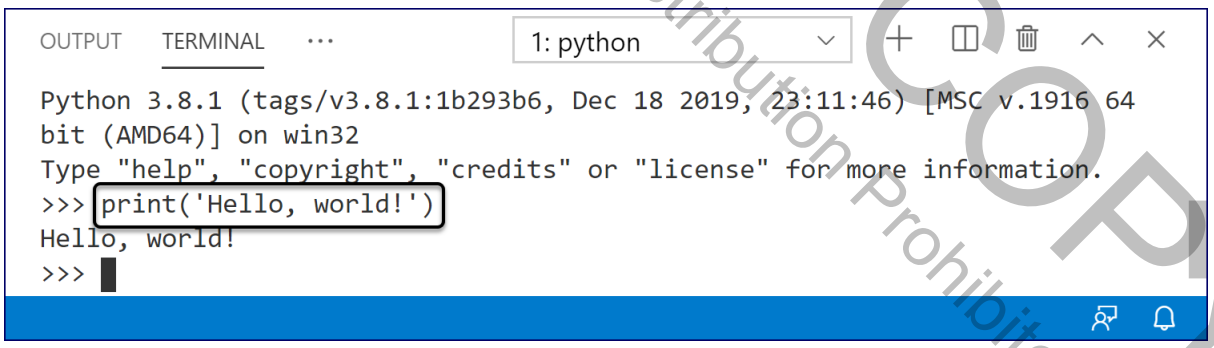
❖ Python Interactive Shell

You can run Python in *Interactive Mode* by running `python` at the terminal:



This will open up the *Python shell*, at which you can run Python commands. For example, to print out "Hello, world!", you would run:

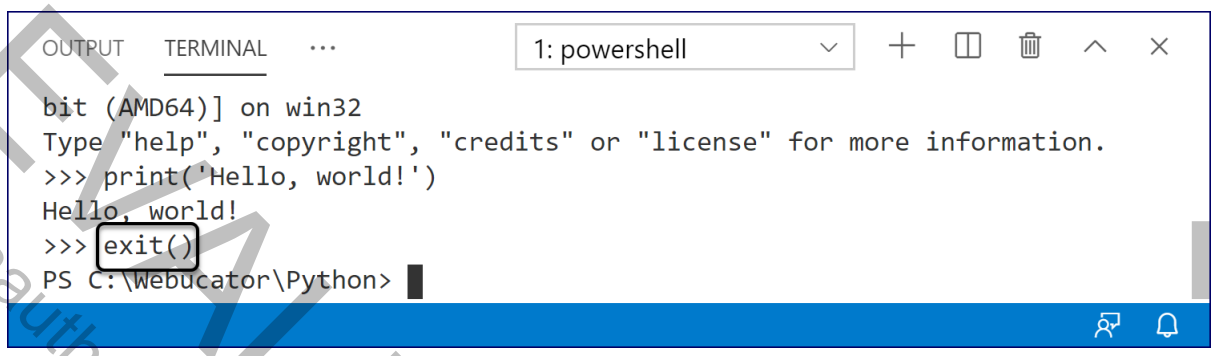
```
print('Hello, world!')
```



You can tell that you are at the Python prompt by the three right-angle brackets:

To exit the Python shell, run:

```
exit()
```



Now, you are back at the regular prompt.

Running a Python File

While working at the Python shell can be useful in some scenarios, you will often be writing Python files, so you can save, re-run, and share your code. Let's get started with a simple "Hello, world!" demo using your editor. We will open a *script*, which is simply a file with a `.py` extension that contains Python code. After the demonstration, you will add another line of code to the script in an exercise.

Here is the script we are going to run:

Demo 1: python-basics/Demos/hello_world.py

```
1.   # Say Hello to the World
2.   print("Hello, world!")
```

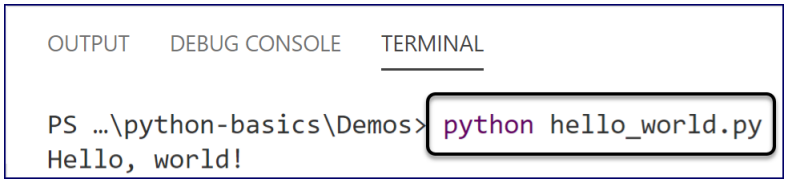
Code Explanation

The `print()` function simply outputs content either to standard output (e.g., the terminal) or to a file if specified.

To run this code:

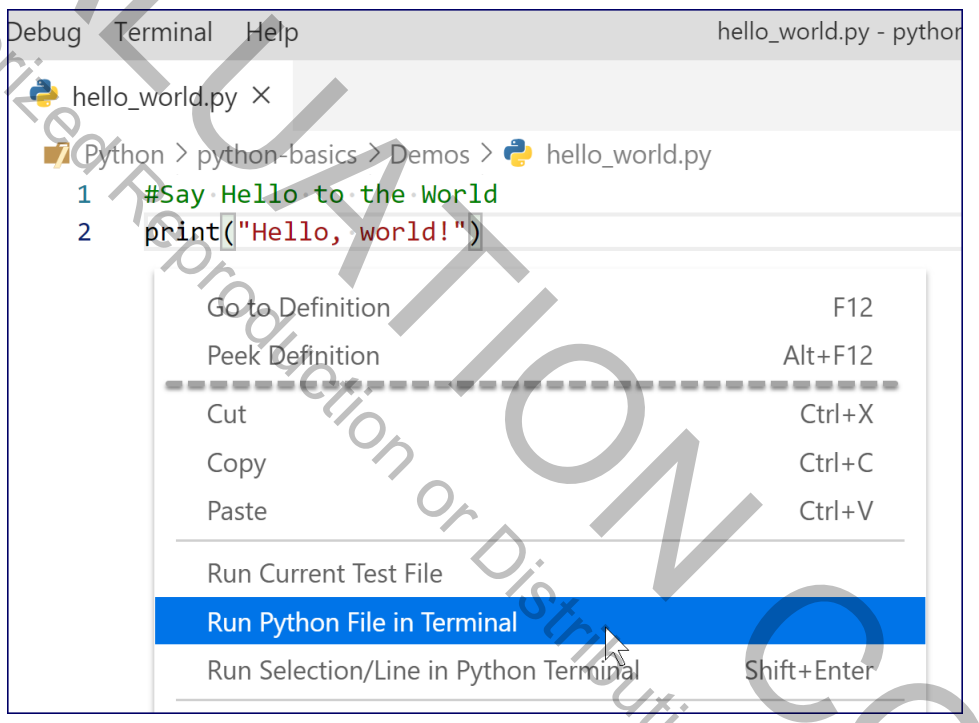
1. Open the terminal at `python-basics/Demos`.
2. Run `python hello_world.py`
3. The "Hello, world!" message will be displayed.

Here it is in the terminal:



Right-click and Run

Another way to run a file from within Visual Studio Code is to right-click on the open file and select **Run Python File at Terminal**:



If you don't see this option, then you don't have the Python extension installed.

Exercise 1: Hello, world!

5 to 10 minutes

1. Open `python-basics/Demos/hello_world.py` in your editor.
2. Add the following line after the "Hello, world!" line:

```
print("Hello again, world!")
```

3. Save your changes.
4. Run the Python file just as you did earlier for the demonstration.
5. The output should look like this:

```
Hello, world!
Hello again, world!
```

Solution: python-basics/Solutions/hello_again_world.py

```
# Say Hello to the World (twice!)
print("Hello, world!")
print("Hello again, world!")
```

Code Explanation

The extra line of code will cause a second message to be printed to the standard output.

Literals

When a value is hard coded into an instruction, as `"Hello"` is in `print("Hello")`, that value is called a *literal* because the Python interpreter should not try to interpret it but should take it *literally*. If we left out the quotation marks (i.e., `print(Hello)`), Python would output an error because it would not know how to interpret the name `Hello`.

Either single quotes or double quotes can be used to create *string* literals. Just make sure that the open and close quotation marks match.

Literals representing numbers (e.g., `42` and `3.14`) are note enclosed in quotation marks.

Python Comments

In the previous demo, you may have noticed this line of code:

```
# Say Hello to the World
```

That number sign (or hash or pound sign) indicates a comment. Everything that trails it on the same line will be ignored.

❖ Multi-line Comments

There is no official syntax for creating multi-line comments; however, Guido van Rossum, the creator of Python, tweeted this tip[4] as a workaround:

4. https://twitter.com/gvanrossum/status/1126706055050772 48?lang=en

Multi-line strings are created with triple quotes, like this:

```
"""This is a
very very helpful and informative
comment about my code!"""
```

Because multi-line strings generate no code, they can be used as pseudo-comments. In certain situations, these pseudo-comments can get confused with *docstrings*, which are used to auto-generate Python documentation, so we recommend you avoid using them until you become familiar with docstrings. Instead, use:

```
# This is a
# very very helpful and informative
# comment about my code!
```

Data Types

In Python programming, objects have different *data types*. The data type determines both what an object can do and what can be done to it. For example, an object of the data type *integer* can be subtracted from another integer, but it cannot be subtracted from a *string* of characters.

In the following list, we show the basic data types used in Python. Abbreviations are in parentheses.

1. boolean (bool) – A `True` or `False` value.
2. integer (int) – A whole number.
3. float (float) – A decimal.

4. string (str) – A sequence of Unicode[5] characters.
5. list (list) – An ordered sequence of objects, similar to an array in other languages.
6. tuple (tuple) – A sequence of fixed length, in which an element's position is meaningful.
7. dictionary (dict) – An unordered grouping of key-value pairs.
8. set (set) – An unordered grouping of values.

We will cover all of these data types in detail.

5. Unicode is a 16-bit character set that can handle text from most of the world's languages.

Exercise 2: Exploring Types

10 to 15 minutes

In this exercise, you will use the built-in `type()` function to explore different data types.

1. Open the terminal.
2. Start the Python shell by typing `python` and then pressing **Enter**:

```
PS …\python-basics\Demos> python
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

   You are now in the Python shell.

3. To check the type of an object, use the `type()` function. For example, `type(3)` will return `<class 'int'>`:

```
>>> type(3)
<class 'int'>
```

4. Find the types of all of the following:
   A. `3`
   B. `3.1`
   C. `'3'`
   D. `'pizza'`
   E. `True`
   F. `('1', '2', '3')`
   G. `['1', '2', '3']`
   H. `{'1', '2', '3'}`

Solution

When you're done, the output should appear as follows:

```
>>> type(3)
<class 'int'>
>>> type(3.1)
<class 'float'>
>>> type('3')
<class 'str'>
>>> type('pizza')
<class 'str'>
>>> type(True)
<class 'bool'>
>>> type(('1', '2', '3'))
<class 'tuple'>
>>> type(['1', '2', '3'])
<class 'list'>
>>> type({'1', '2', '3'})
<class 'set'>
```

Don't worry if you're not familiar with all of the preceding data types. We will cover them all.

Class and Type

You may wonder at Python's use of the word "class" when outputting a data type. In Python, "class" and "type" mean the same thing.

Variables

Variables are used to hold data in memory. In Python, variables are untyped, meaning you do not need to specify the data type when creating the variable. You simply assign a value to a variable. Python determines the type by the value assigned.

❖ Variable Names

Variable names are case sensitive, meaning that `age` is different from `Age`. By convention, variable names are written in all lowercase letters and words in variable names are separated by underscores

(e.g., `home_address`). Variable names must begin with a letter or an underscore and may contain only letters, digits, and underscores.

Keywords

The following list of keywords have special meaning in Python and may not be used as variable names:

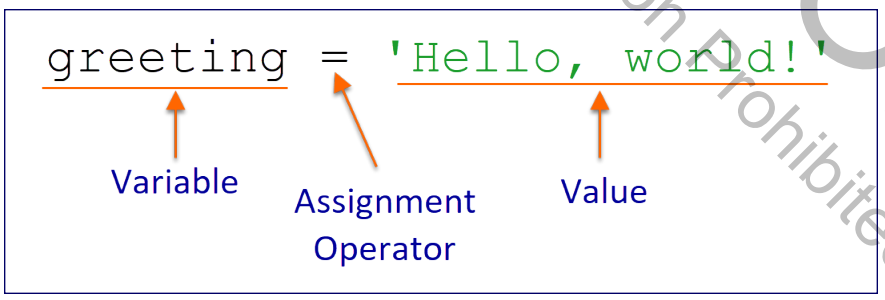
Python Keywords

and	del	if	pass
as	elif	import	raise
assert	else	in	return
async	except	is	True
await	False	lambda	try
break	finally	None	while
class	for	nonlocal	with
continue	from	not	yield
def	global	or	

❖ Variable Assignment

There are three parts to variable assignment:

1. Variable name.
2. Assignment operator.
3. Value assigned. This could be any data type.



Here is the "Hello, world!" script again, but this time, instead of outputting a literal, we assign a string to a variable and then output the variable:

Demo 2: python-basics/Demos/hello_variables.py

```
1.   greeting = "Hello, world!"
2.   print(greeting)
```

Code Explanation

Run this Python file at the terminal. The output will be the same as it was in the previous demo (see page 7):

```
Hello, world!
```

❖ Simultaneous Assignment

A very cool feature of Python is that it allows for simultaneous assignment. The syntax is as follows:

```
var_name1, var_name2 = value1, value2
```

This can be useful as a shortcut for assigning several values at once, like this:

```
>>> first_name, last_name, company = "Nat", "Dunn", "Webucator"
>>> first_name
'Nat'
>>> last_name
'Dunn'
>>> company
'Webucator'
```

Exercise 3: A Simple Python Script

5 to 10 minutes

In this exercise, you will write a simple Python script from scratch. The script will create a variable called `today` that stores the day of the week.

1. In your editor create a new file and save it as `today.py` in the `python-basics/Exercises` folder.
2. Create a variable called `today` that holds the current day of the week as literal text (i.e., in quotes).
3. Use the `print()` function to output the variable value.
4. Save your changes.
5. Test your solution.

Solution: python-basics/Solutions/today.py

```
1.    today = "Monday"
2.    print(today)
```

Constants

In programming, a constant is like a variable in that it is an identifier that holds a value, but, unlike variables, constants are not variable, they are constant. Good name choices, right?

Python doesn't really have constants, but as a convention, variables that are meant to act like constants (i.e., their values are not meant to be changed) are written in all capital letters. For example:

```
PI = 3.141592653589793
RED = "FF0000" # hexadecimal code for red
```

Deleting Variables

Although it's rarely necessary to do so, variables can be deleted using the `del` statement, like this:

```
>>> a = 1
>>> print(a)
1
>>> del a
>>> print(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

Notice that trying to print `a` results in an error, because after `del a`, the `a` variable no longer exists.

Writing a Python Module

A Python module is simply a file that contains code that can be reused. The `today.py` file is really a module, albeit a very simple one. A module can be run by itself or as part of a larger program. It is too early to get into all the aspects of code reuse and modular programming, but you want to start with good habits, one of which is to use a `main()` function in your programs.

❖ The `main()` Function

Let's look at the basic syntax of a function. A function is created using the `def` keyword like this:

Demo 3: python-basics/Demos/indent_demo.py

```
1.   def main():
2.       print("I am part of the function.")
3.       print("I am also part of the function.")
4.       print("Hey, me too!")
5.   print("Sad not to be part of the function. I've been outdented.")
6.
7.   main()
```

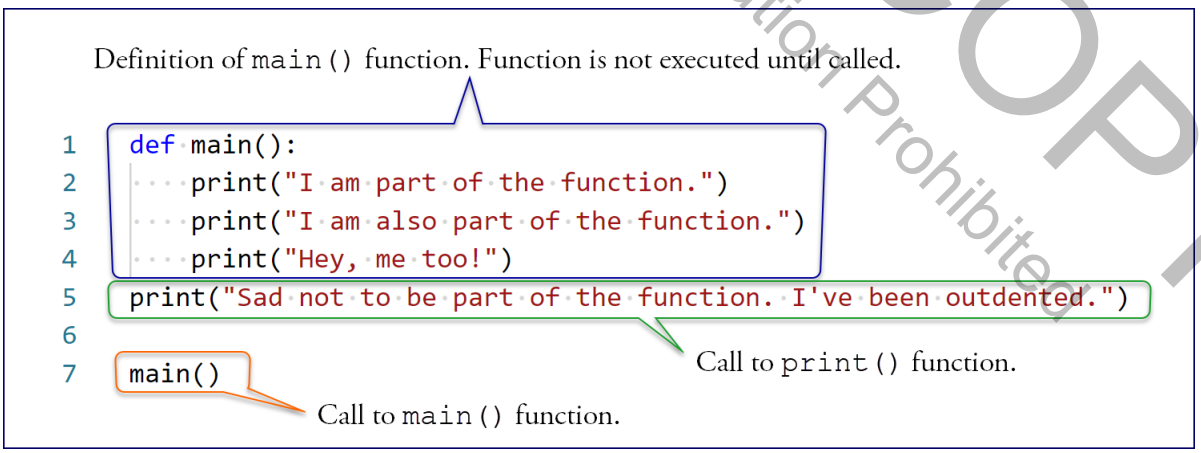
Code Explanation

Running this Python file will render the following:

```
Sad not to be part of the function. I've been outdented.
I am part of the function.
I am also part of the function.
Hey, me too!
```

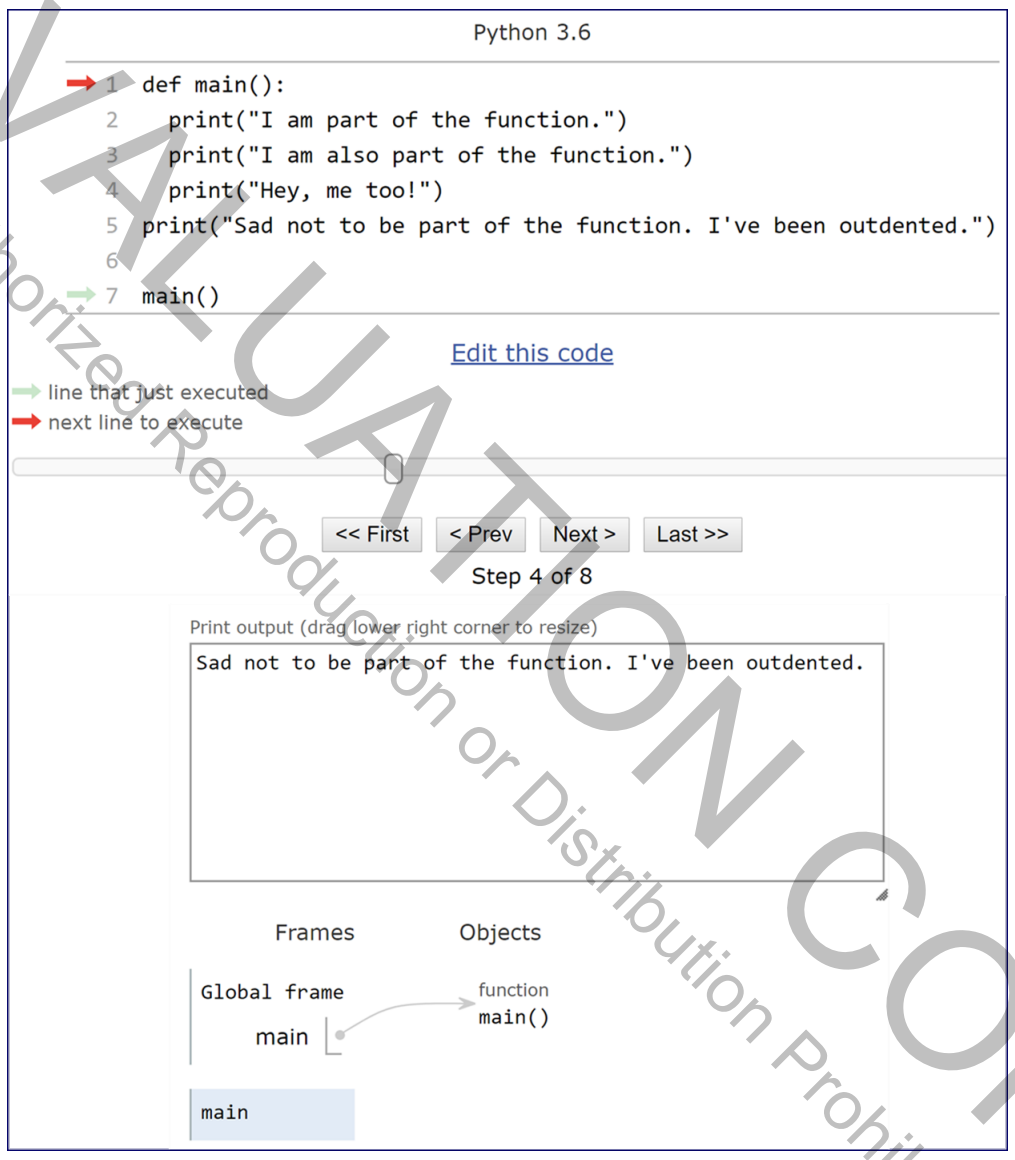
Notice that the first line of output is the line that is not part of the function. That is because the function does not run until it is called, and it is called after the `print()` function that outputs "Sad not to be part of the function. I've been outdented."

The code is read by the Python interpreter from top to bottom, but the function definition just defines the function; it does not *invoke the function* (programmer speak for "call the function to execute").



pythontutor.com

Visit `http://bit.ly/pythontutor-indentdemo` to see this demo in pythontutor.com[6], a web application for visualizing how Python executes code.



A few things to note about functions:

6. http://www.pythontutor.com/visualize.html

1. Functions are created using the `def` keyword. The content that follows the `def` keyword on the same line is called the *function signature*.
2. The convention for naming functions is the same as that for variables: use all lowercase letters and separate words with underscores.
3. In the function definition, the function name is followed by a pair of parentheses, which may contain parameters (more on that soon), and a colon.
4. The contents of the function starts on the next line and must be indented. Either spaces or tabs can be used for indenting, but spaces are preferred.
5. The first line of code after the function definition that is not indented is not part of the function and effectively marks the end of the function definition.
6. Functions are invoked using the function name followed by the parentheses (e.g., `indent_demo()`).

There is nothing magic about the name "main". It is simply the name used by convention for the function that starts the program or module running.[7]

Grouping of Statements

A programming *statement* is a unit of code that does something. The following code shows two statements:

```
greeting = "Hello!"
print(greeting)
```

In Python, statements are grouped using indenting. As we just saw with the `main()` function, lines that are indented below the function signature are part of the function. It is important to understand this: *changes in indentation level denote code groups*. You must be careful with your indenting.

`print()` Function

You have already seen how to output data using the built-in `print()` function. Now, let's see how to output a variable and a literal string together. The `print()` function can take multiple arguments. By

7. Although only a convention, the name "main()" was not chosen arbitrarily. It is used because modules refer to themselves as "__main__", so it seems fitting to get them started with a corresponding "main()" function.

default, it will output them all separated by single spaces. For example, the following code will output "H e l l o !"

```
print('H', 'e', 'l', 'l', 'o', '!')
```

This functionality allows for the combination of literal strings and variables as shown in the following demo:

Demo 4: python-basics/Demos/variable_and_string_output.py

```
1.  def main():
2.      today = "Monday"
3.      print("Today is", today, ".")
4.
5.  main()
```

Code Explanation

Run the Python file. It should render the following:

```
Today is Monday .
```

Notice the extra space before the period in the output of the last demo:

```
Today is Monday␣.
```

We'll get rid of that soon.

❖ Named Arguments

As we have seen with `print()`, functions can take multiple arguments. These arguments can be *named* or *unnamed*. To illustrate, let's look at some more arguments the `print()` function can take:

```
print('H', 'e', 'l', 'l', 'o', '!', sep=' ', end='\n')
```

Those last two arguments are **named arguments**.

- `sep` is short for "separator." It specifies the character that separates the list of objects to output. The default value is a single space, so specifying `sep=" "` doesn't change the default behavior at all.
- `end` specifies the character to print at the very end (i.e., after the last printed object). The default is a newline character (denoted with `\n`). You can use an empty string (e.g., `''`) to specify that nothing should be printed at the end.

The following demo shows how the `sep` argument can be used to get rid of the extra space we saw in the previous example:

Demo 5: python-basics/Demos/variable_and_string_output_fixed_spacing.py

```
1.   def main():
2.       today = "Monday"
3.       print("Today is ", today, ".", sep="")
4.
5.   main()
```

Code Explanation

Run the Python file. It should render the following:

```
Today is Monday.
```

Collecting User Input

Functions may or may not return values. The `print()` function, for example, does not return a value.

Python provides a built-in `input()` function, which takes a single argument: the prompt for the user's input. Unlike `print()`, the `input()` function *does* return a value: the input from the user as a string. The following demo shows how to use it to prompt the user for the day of the week.

Demo 6: python-basics/Demos/input.py

```
1.   def main():
2.       today = input("What day is it? ")
3.       print("Wow! Today is ", today, "? Awesome!", sep="")
4.
5.   main()
```

Code Explanation

Run the Python file. It should immediately prompt the user:

```
What day is it?
```

Enter the day and press **Enter**. It will output something like:

```
Wow! Today is Monday? Awesome!
```

The full output will look like this:

```
What day is it? Monday
Wow! Today is Monday? Awesome!
```

Exercise 4: Hello, You!

5 to 10 minutes

In this exercise, you will greet the user by name.

1. Open a new script. Save it as `hello_you.py` in `python-basics/Exercises`.
2. Write code to prompt for the user's name.
3. After the user has entered their name, output a greeting.
4. Save your changes.
5. Test your solution.

Solution: python-basics/Solutions/hello_you.py

```
1.   def main():
2.       your_name = input("What is your name? ")
3.       print("Hello, ", your_name, "!", sep="")
4.
5.   main()
```

Code Explanation

The code should work like this:

```
PS …\python-basics\Solutions> python hello_you.py
What is your name? Nat
Hello, Nat!
```

Reading from and Writing to Files

To built-in `open()` method is used to open a file from the file system. We will cover this in the **File Processing** lesson (see page 261). For now, you just need to know how to read from a file and how to write to a file.

❖ Reading from a File

The following code shows the steps to:

1. Open a file and assign the file to a *file handler*.
2. Read the content of the file into a variable.
3. Print that variable.
4. Close the file.

```
f = open("my-file.txt") # Open my-file.txt and assign result to f.
content = f.read() # Read contents of file into content variable.
print(content) # Print content.
f.close() # Close the file.
```

Because we referenced the file name directly, Python will look in the current directory for the file. If the file is located in a different directory, you must provide the path, either as an absolute or relative path.[8]

with Blocks

It is important to close the file to free up the memory space the handler is taking up. It's also easy to forget to do so. Fortunately, Python provides a structure that makes explicitly closing the file unnecessary:

```
with open("my-file.txt") as f:
    content = f.read()
    print(content)
```

When Python reaches the end of the `with` block, it understands that the file is no longer necessary and automatically closes it.

❖ Writing to a File

In addition to the path to the file, the `open()` function takes a second parameter: `mode`, which indicates whether the file is being opened for reading (`"r"`), writing (`"w"`), or appending (`"a"`). The default value for `mode` is `"r"`, which is why we didn't need to pass a value in when opening the file for reading. If we want to write to a file, we do have to pass in a value: `"w"`.

```
with open("my-file2.txt", "w") as f:
    f.write("Hello, world!!!!!")
```

When you run the code above, it will open `my-file2.txt` if it exists and overwrite the file with the text you write to it. If it doesn't find a file named `my-file2.txt`, it will create it.

8. Absolute paths start from the top of the file system and work their way downwards towards the referenced file. Relative paths start from the current location (the location of the file containing the path) and work their way to the referenced file from that location.

Exercise 5: Working with Files

10 to 15 minutes

In this exercise, you will open two files that contain lists of popular boys and girls names from 1880,[9] read their contents into two variables, and then write the combined content of the two files into a new file.

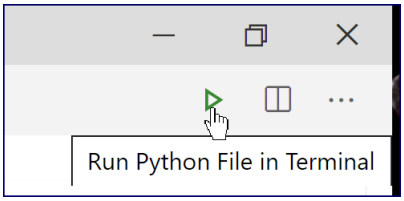
1. Open a new script. Save it as `files.py` in `python-basics/Exercises`.
2. Write code to open `python-basics/data/1880-boys.txt` and read its contents into a variable called `boys`.
3. Write code to open `python-basics/data/1880-girls.txt` and read its contents into a variable called `girls`.
4. Write code to open a new file named `1880-all.txt` in the `python-basics/data` folder and write the combined contents of the `boys` and `girls` variables into the file. Note that you can combine the two strings like this:

   ```
   boys + "\n" + girls
   ```

   That will place one newline between the content in `boys` and the content in `girls`.
5. Save your file.
6. Test your solution. When you run it, it should create the new file. Look in the `data` folder for the `1880-all.txt` file. Does it exist? If so, open it up. Does it have a list of all the boys and girls names?

Run Python File in Terminal

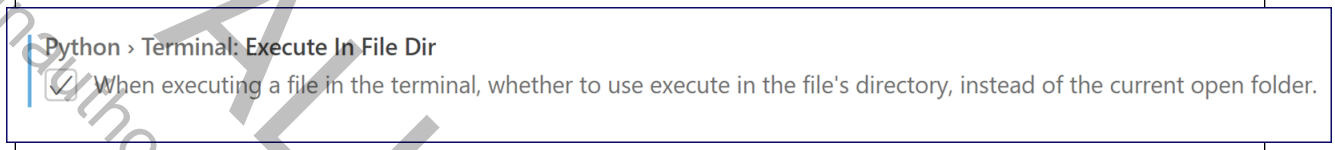
You may have discovered that you can run Python files in the terminal using the green triangle in the upper-right of Visual Studio Code:



9. The lists of names come from `https://www.ssa.gov/oact/babynames/`.

By default, VS Code will run the file from the Workspace root folder and search for any files being referenced with relative paths from that folder. As a result, you may get `FileNotFoundError` errors. You can fix this by changing a setting:

1. From the **File** menu, select **Preferences > Settings**.
2. Search for "execute in file dir".
3. Check the **Python > Terminal: Execute in File Dir** setting.



Solution: python-basics/Solutions/files.py

```
with open("../data/1880-boys.txt") as f_boys:
    boys = f_boys.read()

with open("../data/1880-girls.txt") as f_girls:
    girls = f_girls.read()

with open("../data/1880-all.txt", "w") as f:
    f.write(boys + "\n" + girls)
```

Conclusion

In this lesson, you have begun to work with Python. Among other things, you have learned to use variables, to output data, to collect user input, and to write simple Python functions and modules.

LESSON 2

Functions and Modules

Topics Covered

☑ Defining and calling functions.

☑ Parameters and arguments.

☑ Default values for parameters.

☑ Variable scope.

☑ Return values.

☑ Creating and importing modules.

> It is your duty as a magistrate, and I believe and hope that your feelings as a man will not revolt from the execution of those functions on this occasion.
>
> *– Frankenstein, Mary Shelley*

Introduction

You have seen some of Python's built-in functions. In this lesson, you will learn to write your own.

Defining Functions

We discussed functions a little in the last lesson, but let's quickly review the syntax. Functions are defined in Python using the `def` keyword. The syntax is as follows:

```
def function_name():
    # content of function is indented
    do_something()
# This is no longer part of the function
do_something_else()
```

Here is a modified solution to the "Hello, You!" exercise:

Demo 7: functions/Demos/hello_you.py

```
1.  def say_hello():
2.      your_name = input("What is your name? ")
3.      print("Hello, ", your_name, "!", sep="")
4.
5.  def main():
6.      say_hello()
7.
8.  main()
```

Code Explanation

The code works in the same way, but the meat of the program has been moved out of the `main()` function and into another function. This is common. Usually, the `main()` function handles the flow of the program, but the actual "work" is done by other functions in the module.

The following expanded demo further illustrates how the `main()` function can be used to control flow.

Demo 8: functions/Demos/hello_you_expanded.py

```
1.  def say_hello():
2.      your_name = input("What is your name? ")
3.      insert_separator()
4.      print("Hello, ", your_name, "!", sep="")
5.
6.  def insert_separator():
7.      print("===")
8.
9.  def recite_poem():
10.     print("How about a Monty Python poem?")
11.     insert_separator()
12.     print("Much to his Mum and Dad's dismay")
13.     print("Horace ate himself one day.")
14.     print("He didn't stop to say his grace,")
15.     print("He just sat down and ate his face.")
16.
17. def say_goodbye():
18.     print("Goodbye!")
19.
20. def main():
21.     say_hello()
22.     insert_separator()
23.     recite_poem()
24.     insert_separator()
25.     say_goodbye()
26.
27. main()
```

Code Explanation

The preceding code will render the following:

```
What is your name? Nat
===
Hello, Nat!
===
How about a Monty Python poem?
===
Much to his Mum and Dad's dismay
Horace ate himself one day.
He didn't stop to say his grace,
He just sat down and ate his face.
===
Goodbye!
```

Not All Modules are Programs

Not all Python modules are programs. Some modules are only meant to be used as helper files for other programs. Sometimes these modules are more or less generic, providing functions that could be useful to many different types of programs. And sometimes these modules are written to work with a specific program. Modules that aren't programs probably wouldn't have a `main()` function.

Variable Scope

Question: Why doesn't the `say_goodbye()` function use the user's name (e.g., `print("Goodbye,", your_name)`)?

Answer: It doesn't know what `your_name` is.

Variables declared within a function are *local* to that function. Consider the following code:

Demo 9: functions/Demos/local_var.py

```
1.    def set_x():
2.        x = 1
3.
4.    def get_x():
5.        print(x)
6.
7.    def main():
8.        set_x()
9.        get_x()
10.
11.   main()
```

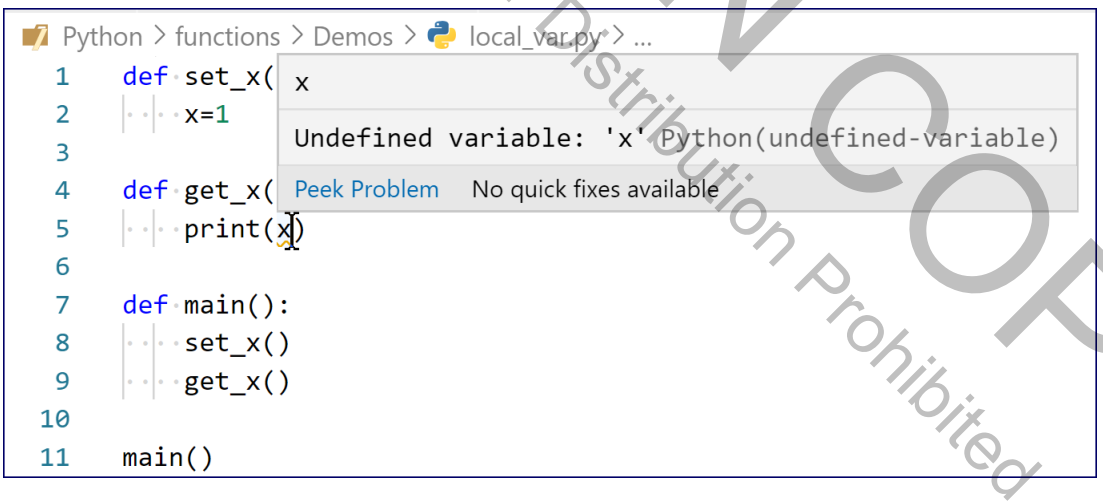
Code Explanation

Run this and you'll get an error similar to the following:

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

That's because `x` is defined in the `set_x()` function and is therefore *local* to that function.

A good Python IDE, like Visual Studio Code, will let you know when it detects such an error. In VS Code, a squiggly underline will appear beneath the undefined variable. Depending on how your settings are configured, you may be able to hover over the variable to see the error:



Global Variables

Global variables are defined outside of a function as shown in the following demo:

Demo 10: functions/Demos/global_var.py

```
1.   x = 0
2.
3.   def set_x():
4.       x = 1
5.       print("from set_x():", x)
6.
7.   def get_x():
8.       print("from get_x():", x)
9.
10.  def main():
11.      set_x()
12.      get_x()
13.
14.  main()
```

Code Explanation

`x` is first declared outside of a function, which means that it is a *global* variable.

Question: What do you think the `get_x()` function will print: 0 or 1?

Answer: It will print 0. That's because the `x` in `set_x()` is not the same as the global `x`. The former is local to the `set_x()` function.

Global variables, by default, can be referenced but not modified within functions:

- When a variable is *referenced* within a function, Python first looks for a local variable by that name. If it doesn't find one, then it looks for a global variable.
- When a variable is *assigned* within a function, it will be a local variable, even if a global variable with the same name already exists.

To modify global variables within a function, you must explicitly state that you want to work with the global variable. That's done using the `global` keyword, like this:

Demo 11: functions/Demos/global_var_in_function.py

```
x = 0

def set_x():
    global x
    x = 1

def get_x():
    print(x)

def main():
    set_x()
    get_x()

main()
```

Code Explanation

Now, the `set_x()` function explicitly references the global variable `x`, so the code will print `1`.

Naming global variables?

Some developers feel that any use of global variables is bad programming. While we won't go that far, we do have two recommendations:

1. Prefix your global variables with an underscore[10] (e.g., `_x`). That makes it clear that the variable is global and minimizes the chance of it getting confused with a local variable of the same name. It is also a convention that lets developers know that those variables are not meant to be used outside the module (i.e., by programs importing the module).
2. When possible, rather than using global variables, design your code so that values can be passed from one function to another using parameters (see next section).

Function Parameters

Consider the `insert_separator()` function in the `hello_you_expanded.py` file that we saw earlier:

10. https://www.python.org/dev/peps/pep-0008/#global-variable-names

```
def insert_separator():
    print("===")
```

What if we wanted to have different types of separators? One solution would be to create multiple functions, like `insert_large_separator()` and `insert_small_separator()`, but that can get pretty tiresome and hard to maintain. A better solution is to use function parameters using the following syntax:

```
def function_name(param1, param2, param3):
    do_something(param1, param2, param3)
```

Here is our "Hello, You!" program using parameters:

Demo 12: functions/Demos/hello_you_with_params.py

```
1.  def say_hello(name):
2.      print('Hello, ', name, '!', sep='')
3.
4.  def insert_separator(s):
5.      print(s, s, s, sep="")
6.
7.  def recite_poem():
8.      print("How about a Monty Python poem?")
9.      insert_separator("-")
10.     print("Much to his Mum and Dad's dismay")
11.     print("Horace ate himself one day.")
12.     print("He didn't stop to say his grace,")
13.     print("He just sat down and ate his face.")
14.
15. def say_goodbye(name):
16.     print('Goodbye, ', name, '!', sep='')
17.
18. def main():
19.     your_name = input('What is your name? ')
20.     insert_separator("-")
21.     say_hello(your_name)
22.     insert_separator("=")
23.     recite_poem()
24.     insert_separator("=")
25.     say_goodbye(your_name)
26.
27. main()
```

Code Explanation

Now that `insert_separator()` takes the separating character as an argument, we can use it to separate lines with any character we like.

We have also modified `say_hello()` and `say_goodbye()` so that they receive the name of the person they are addressing as an argument. This has a couple of advantages:

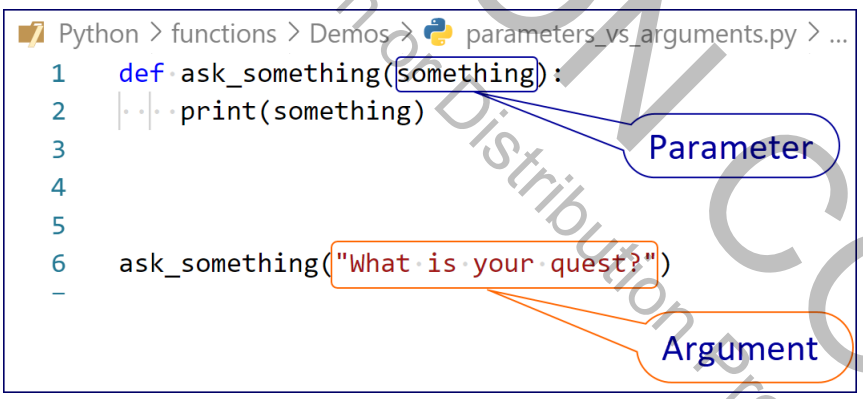
1. We can move `your_name = input('What is your name? ')` to the `main()` function so we can pass `your_name` into both `say…` functions.
2. We can move the call to `insert_separator()` out of the `say_hello()` function as the separator doesn't have anything to do with saying "hello."

Parameters vs. Arguments

The terms *parameter* and *argument* are often used interchangeably, but there is a difference:

Parameters are the variables in the function definition. They are sometimes called *formal parameters*.

Arguments are the values passed into the function and assigned to the parameters. They are sometimes called *actual parameters*.



❖ Using Parameter Names in Function Calls

When calling a function, you can specify the parameter by name when passing in an argument. When you do so, it's called passing in a *keyword argument*. For example, you can call the following `divide()` function in several ways:

```
def divide(numerator, denominator):
    return numerator / denominator

divide(10, 2)
divide(numerator=10, denominator=2)
divide(denominator=2, numerator=10)
divide(10, denominator=2)
```

As you can see, using keyword arguments allows you to pass in the arguments in an arbitrary order. You can combine non-keyword arguments with keyword arguments in a function call, but you must pass in the non-keyword arguments first.

Later, we'll see that a function can be written to require keyword arguments.

Exercise 6: A Function with Parameters

15 to 25 minutes

In this exercise, you will write a function for adding two numbers together.

1. Open `functions/Exercises/add_nums.py` in your editor and review the code.
2. Now, run the file in Python. The output should look like this:

```
3 + 6 = 9
10 + 12 = 22
```

3. Replace the two crazy `add…()` functions with an `add_nums()` function that accepts two numbers, adds them together, and outputs the equation.

Exercise Code: functions/Exercises/add_nums.py

```
1.  def add_3_and_6():
2.      total = 3 + 6
3.      print('3 + 6 = ', total)
4.
5.  def add_10_and_12():
6.      total = 10 + 12
7.      print('10 + 12 = ', total)
8.
9.  def main():
10.     add_3_and_6()
11.     add_10_and_12()
12.
13. main()
```

Code Explanation

The first function adds `3` and `6`. The second function adds `10` and `12`. These functions are only useful if you want to add those specific numbers.

Solution: functions/Solutions/add_nums.py

```
1.  def add_nums(num1, num2):
2.      total = num1 + num2
3.      print(num1, '+', num2, ' = ', total)
4.
5.  def main():
6.      add_nums(3, 6)
7.      add_nums(10, 12)
8.
9.  main()
```

Code Explanation

The `add_nums()` function is flexible and reusable. It can add any two numbers.

Default Values

Parameters that do not have default values require arguments to be passed in. You can assign default values to parameters using the following syntax:

```
def function_name(param=default):
    do_something(param)
```

For example, the following code would make the "=" sign the default separator:

```
def insert_separator(s="="):
    print(s, s, s, sep="")
```

When an argument is not passed into a parameter that has a default value, the default is used.

See `functions/Demos/hello_you_with_defaults.py` to see a working demo.

Exercise 7: Parameters with Default Values

15 to 25 minutes

In this exercise, you will write a function that can add two, three, four, or five numbers together.

1. Open `functions/Exercises/add_nums_with_defaults.py` in your editor.
2. Notice the `add_nums()` function takes five arguments, adds them together, and outputs the sum.
3. Modify `add_nums()` so that it can accept all of the following calls:

```
add_nums(1, 2)
add_nums(1, 2, 3, 4, 5)
add_nums(11, 12, 13, 14)
add_nums(101, 201, 301)
```

4. For now, it's okay for the function to print out 0s for values not passed in, like this:

```
1 + 2 + 0 + 0 + 0  =  3
1 + 2 + 3 + 4 + 5  =  15
11 + 12 + 13 + 14 + 0  =  50
101 + 201 + 301 + 0 + 0  =  603
```

Exercise Code: functions/Exercises/add_nums_with_defaults.py

```
def add_nums(num1, num2, num3, num4, num5):
    total = num1 + num2 + num3 + num4 + num5
    print(num1, '+', num2, '+', num3, '+', num4, '+', num5, ' = ', total)

def main():
    add_nums(1, 2, 0, 0, 0)
    add_nums(1, 2, 3, 4, 5)
    add_nums(11, 12, 13, 14, 0)
    add_nums(101, 201, 301, 0, 0)

main()
```

Solution: functions/Solutions/add_nums_with_defaults.py

```
1.  def add_nums(num1, num2, num3=0, num4=0, num5=0):
2.      total = num1 + num2 + num3 + num4 + num5
3.      print(num1, '+', num2, '+', num3, '+', num4, '+', num5, ' = ', total)
4.
5.  def main():
6.      add_nums(1, 2)
7.      add_nums(1, 2, 3, 4, 5)
8.      add_nums(11, 12, 13, 14)
9.      add_nums(101, 201, 301)
10.
11. main()
```

Code Explanation

We have given the last three parameters default values of `0`, making them optional. The first two parameters don't have default values, so they are still required.

Returning Values

Functions can return values. The `add_nums()` function we have been working with does more than add the numbers passed in, it also prints them out. You can imagine wanting to add numbers for some other purpose than printing them out. Or you might want to print the results out in a different way. We can change the `add_nums()` function so that it just adds the numbers together and returns the sum. Then our program can decide what to do with that sum. Take a look at the following code:

Demo 13: functions/Demos/add_nums_with_return.py

```
1.   def add_nums(num1, num2, num3=0, num4=0, num5=0):
2.       total = num1 + num2 + num3 + num4 + num5
3.       return total
4.
5.   def main():
6.       result = add_nums(1, 2)
7.       print(result)
8.       result = add_nums(result, 3)
9.       print(result)
10.      result = add_nums(result, 4)
11.      print(result)
12.      result = add_nums(result, 5)
13.      print(result)
14.      result = add_nums(result, 6)
15.      print(result)
16.
17.  main()
```

Code Explanation

The `add_nums()` function now returns the sum to the calling function via the `return` statement. We assign the result to a local variable named `result`. Then we print `result` and pass it back to `add_nums()` in subsequent calls.

Note that once a function has returned a value, the function is finished executing and control is transferred back to the code that invoked the function.

Importing Modules

As we saw, part of the beauty of writing functions is that they can be reused. Imagine you write a really awesome function. Or even better, a module with a whole bunch of really awesome functions in it. You'd want to make those functions available to other modules so you (and other Python developers) could make use of them elsewhere.

Modules can import other modules using the `import` keyword as shown in the following example:

Demo 14: functions/Demos/import_example.py

```
1.   import add_nums_with_return
2.
3.   total = add_nums_with_return.add_nums(1, 2, 3, 4, 5)
4.   print(total)
```

Code Explanation

Now, the `add_nums()` function from `add_nums_with_return.py` is available in `import_example.py`; however, it must be prefixed with "`add_nums_with_return.`" (the module name) when called.

The main() Function

It is common for a module to check to see if it is being imported by checking the value of the special `__name__` variable. Such a module will only run its `main()` function if `__name__` is equal to `'__main__'`, indicating that it is not being imported. The code usually goes at the bottom of the module and looks like this:

```
if __name__ == '__main__':
    main()
```

Note that those are two underscores before and after `name` and before and after `main`.

A short video explanation of this is available at `https://bit.ly/python_main`.

Another way to import functions from another module is to use the following syntax:

```
from module_name import function1, function2
```

For example:

Demo 15: functions/Demos/import_example2.py

```
from add_nums_with_return import add_nums

total = add_nums(1, 2, 3, 4, 5)
print(total)
```

When you use this approach, it is not necessary to prefix the module name when calling the function. However, it's possible to have naming conflicts, so be careful.

Another option, which is helpful for modules with long names, is to create an alias for a module, so that you do not have to type its full name:

```
import add_nums_with_return as anwr

total = anwr.add_nums(1, 2, 3, 4, 5)
```

Using aliases is also a way of preventing naming conflicts. If you import `do_this` from `foo` and `do_this` from `bar`, you can use an alias to call one of them `do_that`:

```
from foo import do_this
from bar import do_this as do_that
```

❖ Module Search Path

The Python interpreter must locate the imported modules. When `import` is used within a script, the interpreter searches for the imported module in the following places sequentially:

1. The current directory (same directory as the script doing the importing).
2. The library of standard modules.[11]
3. The paths defined in `sys.path`, which you can see by running the following code at the Python shell:

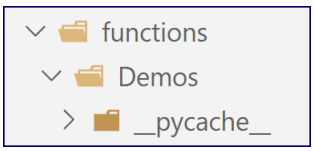
   ```
   >>> import sys
   >>> sys.path
   ```

   This will output a list of paths, which are searched in order for the imported module.

11. https://docs.python.org/3/tutorial/modules.html#standard-modules

❖ pyc Files

Files with a `.pyc` extension are *compiled* Python files. They are automatically created in a `__pycache__` folder the first time a file is imported:



These files are created so that modules you import don't have to be compiled every time they run. You can just leave those files alone. They will automatically be created/updated each time you import a module that is new or has been changed.

Methods vs. Functions

You have already seen some built-in functions, like `print()` and `input()`. You have also written some of your own, like `insert_separator()` and `divide()`. In the upcoming lessons, you will learn to use many more of Python's built-in functions. You will also learn about *methods*, which are similar to functions, except that they are called on an object using the syntax `object_name.method_name()`.

An example of a simple built-in function is `len()`, which returns the length of the passed-in object:

```
>>> len('Webucator')
9
```

An example of a method of a string object is `upper()`, which returns the string it is called upon in all uppercase letters:

```
>>> Webucator'.upper()
'WEBUCATOR'
```

Again, you will work with many functions and methods in upcoming lessons.

Conclusion

In this lesson, you have learned to define functions with or without parameters. You have also learned about variable scope and how to import modules.

7400 E. Orchard Road, Suite 1450 N
Greenwood Village, Colorado 80111
Ph: 303-302-5280
www.ITCourseware.com