

Python Programming

INFO.2820.061

Instructor: Andy Garside

Introducing Python

Introduction

Welcome to Python Programming! In this lecture we will introduce the fundamentals of Python including: Python programs (**scripts**), the Python **Interactive Shell**, **simple expressions**, **variables**, some useful **functions** (including functions to gather user input and display results), along with Python **help** and documentation (**docstrings**), and some common **syntax errors**.

There's lots to cover so let's begin by answering the question, **what is Python?**

What is Python? How does it compare to other programming languages?

There are many kinds of programming languages. Python differs from some others you may have seen.

C

The C programming language is considered a low-level, procedural language. It has a long history and it is widely used in the development of operating systems. It can manipulate low-level machine functionality (in fact, the developer is responsible for allocating and freeing chunks of memory the program uses), but it has no native support for higher-level concepts like strings, lists, hashtables, sets, tuples, or user-definable objects. To achieve high-performance, C programs are "compiled" and "linked". The C source code (program) is typed into a file with a .c extension and the file is run through a compiler to turn the source code into machine code. The machine code is then linked against other objects, like run-time libraries, to produce a platform-dependent executable machine code that is specific to the type of machine the program will run on.

C++

The programming language C++ extends the functionality of C by adding native object support. It allows a developer to declare their own objects and object-related components (for example, a car and all its parts), along with native tools to instantiate a specific instance of an object (for example, a Toyota Camry) and tools to operate on that object (for example, to print the list of parts with a single statement). Like C, C++ is a compiled and linked language, combining high-performance with user-definable objects. As with C, it lacks native support for strings, lists, hashtables, sets, tuples, and other high-level programming constructs.

Python

Python is an object-oriented, high-level programming language released by Guido van Rossum in 1991. Python supports most of the native types found in languages like C & C++ but also natively supports strings, lists, dictionaries (hashtables), sets, tuples, and other programming constructs such as “classes”, “objects”, “inheritance”, and more.

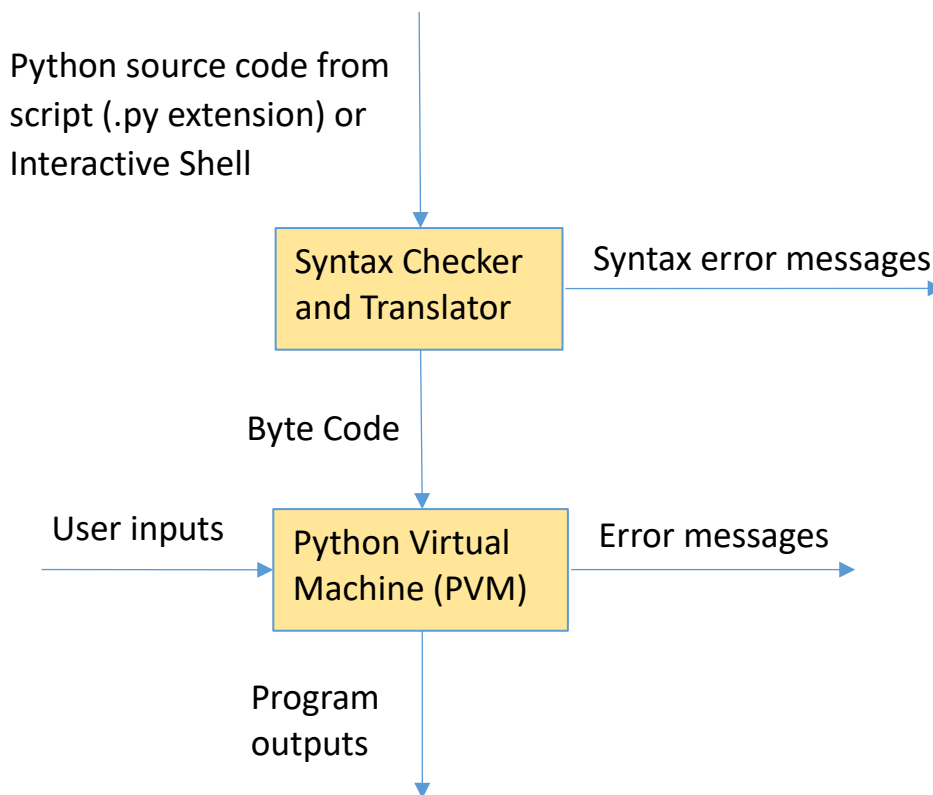
Python is not compiled into machine-dependent machine code like C & C++ but is, instead, compiled into an intermediate form called byte-code which is then “interpreted” by something called the **Python Virtual Machine (PVM)**

A Python program (**source code**) is typed into a file with a **.py** extension (**script**) or typed into an **Interactive Shell**.

When executed, Python source code is checked for **syntax errors** to make sure it is valid Python code. If errors are found, a **syntax error message** is printed, and execution is halted. If syntax checking is successful, the **source code** is automatically translated (compiled) into a lower-level, platform-independent representation called **byte-code**. If the source was a script, the byte-code is stored in a file with the same name as the script and a **.pyc** extension.

The byte-code is then passed to the **Python Virtual Machine (PVM)** where it is **interpreted**, effectively running your program. Any requested **user input** is gathered along the way and any **program output** is displayed. Execution typically continues until the program completes unless a **run-time error** occurs (for example, an expression that results in division by zero), in which case an error is printed and execution is halted.

The diagram below outlines the flow of Python compilation and interpretation.



Note: If your source code is contained in a Python script (rather than being run interactively from the Interactive Shell), each time you run the program, Python passes the .pyc file to the **Python Virtual Machine (PVM)** to execute the byte-code and run your program. This byte-code is machine independent, so you can move the .py or a .pyc file to any platform that supports Python and it should execute without modification.

Why is Python so popular?

Python is popular for many reasons.

Python is easy to write. Unlike statically-typed languages like C, C++ and Java, Python is a dynamically-typed. This means you don't need to declare variable types (like *int*, *float*, *char*, etc). The type of a Python variable is determined by the context of its first assignment. For example, if it's first assigned an integer, then it will be given the type "int". When safe to do so, Python can automatically convert variables between types depending on the context of their use. Python also provides access to many complex native types like strings, lists, dictionaries (hashtables), sets and tuples that simplify the development of complex programs.

Python is object oriented. While many traditional programming languages like C are “procedural”, more modern languages like C++, Java and Python are object oriented, allowing the user to define objects which contain data and methods to access that data. Python, in particular, has a simpler object oriented language structure than C++ or Java, making it easier to learn and use.

Python is portable. You can write a Python program on one platform and run it on a huge number of other platforms – without needing to explicitly recompile it. For instance, you can write a Python program on a Mac and run it on a Windows PC, or even on Linux.

Python has a huge set of standard and open source libraries. The standard libraries include text processing, numeric and mathematical calculations, file and directory access, data compression, cryptographic services, operating system services, concurrent execution (multithreading and multiprocessing), networking and interprocess communication, and much more. You’ll find a good list here: <https://docs.python.org/3/library/>. There is also a huge set of available open source libraries for Data Science, Machine Learning, Statistics, Web Development, and more.

With Python, you never need to manage memory. Low-level languages like C & C++ require the developer to allocate (malloc) and release (free) dynamic memory as it is used. This process is common when you cannot determine the maximum amount of memory needed at compile time, when you need to allocate a very large object, or when you build data structures without a fixed size. In Python, the virtual machine allocates memory when needed and frees it for you when it’s safe to do so, so the programmer need not manage memory.

Finally, like so many things, **Python is popular because so many people already use it.** Python is heavily used in the Internet of Things (IoT), Data Science, Artificial Intelligence, Web development, and more. People want to learn it because the skill is so valuable.

Python scripting and the Interactive Shell

There are two ways to execute Python commands, using an **interactive shell** or by including Python commands in a **script**.

*Note: If you have not yet installed Python on the computer you will be using for this class, please do so using the steps outlined in the file “**Downloading, installing, configuring and Testing Python 3**” which can be found under the “**Home Page**” within the “**Start Here**” folder.*

Also, if you’ve previously installed Python 2, you’ll want to install Python 3. Python 2 has reached its end of life and should not be used for this course

The Interactive Shell

Many programmers think of programming as entering a series of statements into a file (or multiple files) and then “running” the program, possibly after compiling or linking the files. However, because Python is an interpreted language, it’s possible to execute individual statements without following this process.

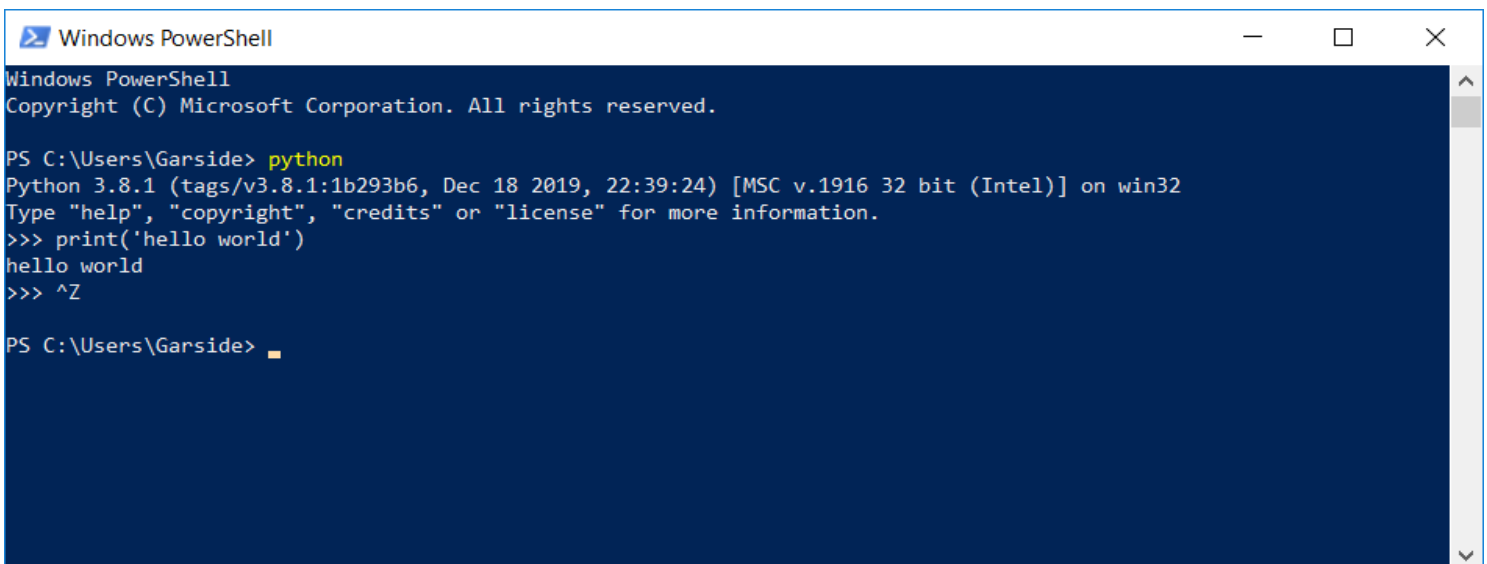
Python provides an interactive environment called a **shell** that provides a prompt where you can enter any valid python expression or statement. After entering an expression or statement and pressing the *Enter* key, the statement is evaluated and the prompt is displayed again.

Note: The Python shell is a useful tool for evaluating simple expressions or short snippets of code. It’s particularly good for understanding how an expression is evaluated and for validating the syntax of an expression. However, when writing a program of any size, you will find it best to use Python scripts. Python scripts allow you to edit, save and rerun your commands, as well as make it easy to share your program with others.

There are a couple ways to invoke the Python interactive shell.

If you’re a user of your operating system’s command line, you can type the word “python” at your command line prompt to start the interactive shell. For example, if you’re a Windows user who uses the Windows PowerShell, you can open the PowerShell and type “python” at the PowerShell prompt. This will start the Python interpreter and present you with the Python interpreter prompt, >>>.

In the example below, the Python statement **print('hello world!')** has been typed at the >>> prompt followed by the **Enter** key. This results in Python printing the string “hello world!”.

A screenshot of a Windows PowerShell window. The title bar says "Windows PowerShell" with standard window controls. The content area has a dark blue background with white text. It shows the PowerShell prompt "PS C:\Users\Garside>" followed by the command "python". Below that, it shows the Python 3.8.1 startup banner, including version, date, and architecture. Then, the Python prompt ">>>" is shown, followed by the command "print('hello world!')". The output "hello world" is printed on the next line. Finally, the user presses Ctrl-Z (^Z) and the PowerShell prompt "PS C:\Users\Garside>" is shown again.

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Users\Garside> python
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print('hello world')
hello world
>>> ^Z


PS C:\Users\Garside> 
```

Invoking the Python interpreter using IDLE (the Integrated Development Environment)

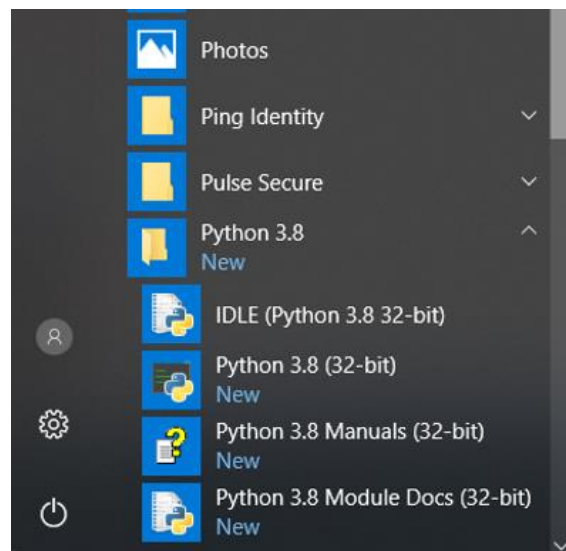
If you're a frequent command line user it can be convenient to use the Python interpreter at your operating system's command line. However, there's a richer interpreter environment that's installed when you install Python, called **IDLE**.

IDLE is an **Integrated Development Environment (IDE)**. An Integrated Development Environment is an environment for developing software (as opposed to a simple editor for entering source code into a file). There are many advantages to using an IDE over a Terminal Command Prompt and a file editor. With IDLE, this includes context-sensitive menu items for performing programming-related tasks, color-coding of the program elements which makes it easier to understand how each program statement is constructed, context-sensitive help, and associated tools, like a debugger.

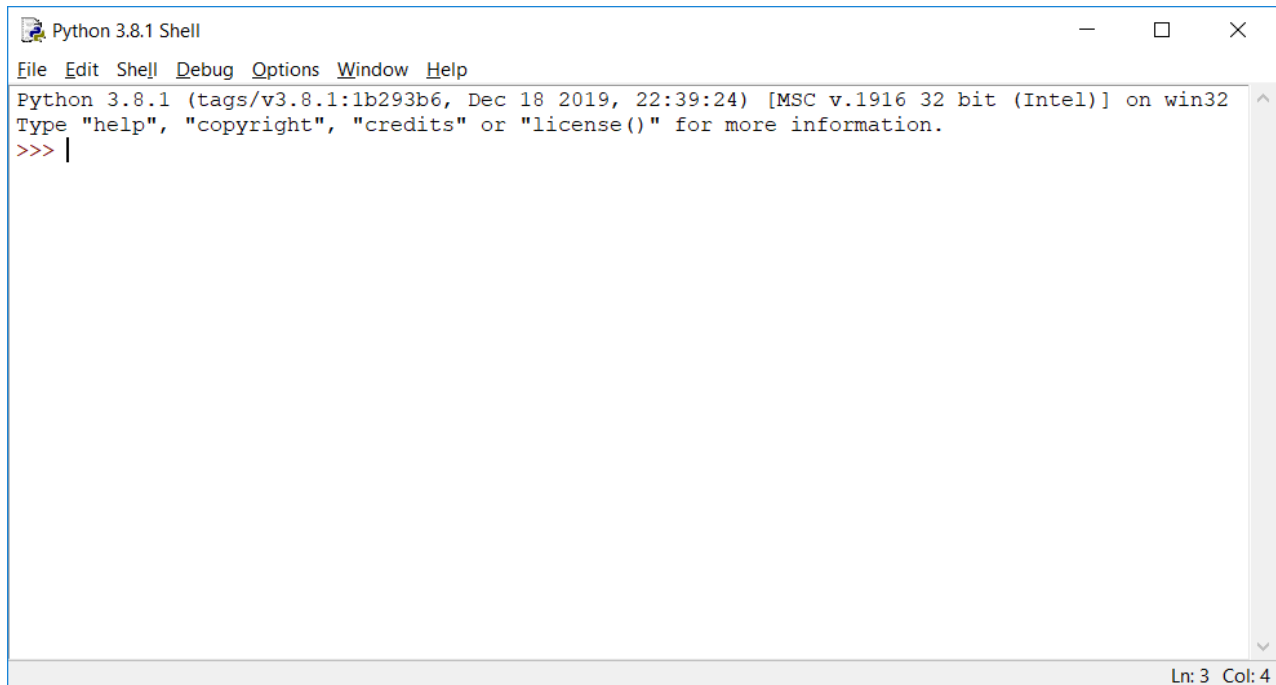
Note: There are many freely-available IDEs that support Python. In the class notes we will focus on IDLE since it's a standard part of the Python kit. However, I encourage you to download and explore the many Python IDEs that are available.

To start IDLE on a Windows desktop, left-click on the Windows logo  on your Windows Taskbar and scroll through the menu items until you find the **Python 3.8** folder and its associated sub-menu items. (If necessary, left-click on the **Python 3.8** folder to reveal the sub-menu items.) Double click on the menu item labeled **IDLE (Python 3.8 32-bit)**

Note: The lecture notes refer to Python version 3.8. You can use any version of Python as long as the first digit in the version number is a 3.



When the IDLE window appears it will have the title **Python 3.x.x Shell** and include a number of menus for interacting with the shell. It will also display the Python shell prompt, `>>>`.



Notes:

Exiting IDLE. You can quit the IDLE Python Shell in 4 different ways.

- 1) Left-click the **X** in the upper-right corner of the IDLE Python Shell window.
- 2) Select **Exit** from the **File** menu.
- 3) Type **Ctrl-Q** (press and release the Ctrl and Q keys at the same time)
- 4) Type **Ctrl-D**

Getting IDLE help. You can get help on using IDLE by selecting **IDLE Help** from the window's drop-down menu or by typing "**help()**" at the `>>>` prompt. If you type "**help()**" at the `>>>` prompt, you can exit the help by simply pressing the **Enter** key at the `>>>` prompt.

Evaluating Simple expressions in the Interactive Shell

Statements vs Expressions: In general, an **expression** evaluates to a value while a **statement** does something. For example, `8 + 3` is an **expression** that evaluates to the value 11, while `print('hello world!')` is a **statement** that displays the string 'hello world!'

You can type any valid Python **expression** or **statement** at the >>> prompt of the Interactive Shell. The expression or statement is then evaluated when you press **Enter** and the results, if any, are displayed, followed by a new prompt. For example, below, the expression 8 + 3 has been typed followed by **Enter** and the result 11 is displayed.

```
>>> 8 + 3
11
```

In its simplest form, an expression is just a value. That is, an expression need not include any operators. For example, you can type an expression which is a single number and the value of that expression is displayed.

```
>>> 8
8
```

In a similar manner, a **fixed** string is an expression. In Python, a **fixed** string is any sequence of characters enclosed in either **single quotes** or **double quotes**. For example, "Hello World!".

If you type "Hello World!" at the Python Shell's >>> prompt, the expression is evaluated and its value is displayed, as shown below.

```
>>> "Hello World"
'Hello World'
```

Notice that displayed result is enclosed in single quotes. Single quotes are displayed by the Python Shell to indicate that the characters within the quotes represent a string. The single quotes are not, themselves, part of the actual value.

*Note: Python supports a string **data type** (**str**). The string data type is **immutable**, meaning it cannot be modified once it is created, however, it can be operated on using various Python operators. A **fixed** string, like "Hello World", is of data type **str**.*

Creating Variables in the Interactive Shell

As in other programming languages, a Python **variable** is a name that refers to an object that holds information. The name is used to reference the object, including assigning a value to the object, printing the object, or otherwise modify the object.

In Python a variable name...

- must **start with a letter or the underscore** character
- can only contain **alpha-numeric** characters and **underscores** (A-z, 0-9, and _)
- is **case-sensitive** (hello, Hello and HELLO are three different variables)
- cannot be one of the words **reserved** by Python for other uses including those in the following table:

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

Unlike many other languages, you do not declare a Python variable's **type**. The **type** of a variable (for example, a string, integer, floating-point value, etc) is **inferred** by the context of its use. For example, below a Python variable named *saying* is created and assigned the **str** value "Hello World!"

```
>>> saying = "Hello World!"
```

Python considers the variable *saying* as a string type because it is assigned a sequence of characters enclosed in quotes.

Invoking Simple Functions in the Interactive Shell

A Python **function** is an encapsulated sequence of Python instructions which can be called from one or more places in a program. We will discuss Python functions at length in a future lecture but, for now, we will introduce the concept of a function in the context of explicitly converting variable types, printing (displaying) output, and gathering input.

The “type”, “int”, and “float” functions

On the first line of the example below, a **variable** named *saying* is created and assigned the string value "Hello World!". On the second line, the built-in function **type** is called and passed the variable *saying*. The **type** function takes an expression and returns the type of that expression. In this case, the type is **'str'**, which is short for **string**.

```
>>> saying = "Hello World!"
>>> type(saying)
<class 'str'>
```

As another example, the **type** command below is passed the expression `3 + 5`. Notice that the returned type is `'int'`, which is short for **integer**.

```
>>> type(3+5)
<class 'int'>
```

While you don't have to declare a variable's **type** in Python, it is important that you use variables appropriately based on their type. For example, below, `x` is assigned the integer value 3 (Python interprets `x` to be an **integer** rather than a **string** because the value 3 is not in quotes). The variable `y` is then assigned the **string** `"4"` (yes, it looks like an integer but it's enclosed in quotes so Python interprets it as a **string**). The simple expression `x + y` then fails and raises a **Traceback** error because Python is being asked to add a variable of type **string** to a variable of type **int**.

```
>>> x = 3
>>> type(x)
<class 'int'>
>>> y = "4"
>>> type(y)
<class 'str'>
>>> x + y
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    x + y
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

When necessary, you can tell Python to convert a variable from one type to another. For example, you can convert a string to an integer using the **int()** function. Below we use the **int()** function to convert the previously created `y` from a **string** to an **integer** and store the value in `z`. Python will then successfully compute the expression `x + z`.

```
>>> z = int(y)
>>> x + z
7
```

The **float()** function can be used similarly to convert an object to a floating-point value (a numeric value with a decimal point and/or an exponent).

The “print” function

The simple expressions we saw earlier were each evaluated by the Python interpreter and displayed by the Python Shell. We will see below, when we discuss Python scripts, that **expressions** are evaluated within scripts but they are not automatically displayed.

To display the value of an expression in a Python script we will call the Python **print()** function. The simplest form of the Python **print()** function is

print(<expression>)

where <expression> represents any valid Python expression.

*Note: The angle brackets < and > are used to indicate that **expression** is not a string but a Python expression. Within Python source code you would replace **<expression>** with any valid python expression like “**Hello World!**”, **3 + 5**, etc.*

For example

```
>>> print("Hello World!")  
Hello World!
```

In this example, the **print** function is called with the fixed string expression “**Hello World!**” and the result is the display of the string **Hello Word!**.

*Note: Strings displayed by the **print** function are not enclosed in quotes. When the Python Shell calls the **print** function it is the **print** function that displays the string **Hello World!**, not the Shell. Unlike the Python Shell, the **print** function does not enclose a string in quotes when displayed.*

Assuming the variable *saying* is defined as it was earlier, we can also pass the variable *saying* to the **print** function

```
>>> print(saying)  
Hello World!
```

In reality, the **print** function can take an unlimited number of expressions separated by **commas**. A more complete version of the Python **print** function syntax is as follows:

print(<expression>,... expression>)

When used in this form, the **print** function displays the value of each expression separated by a space. For example

```
>>> print("Three plus five is", 3 + 5)
Three plus five is 8
```

By default, the Python **print** function prints all the requested text followed by a **newline**. That is, it prints the value of the expressions and then moves the cursor down to the beginning of the next line. You can tell the **print** function not to print the **newline** by adding the expression **end = ""** to your **print** function call. For example

```
print("I have more to add to this line", end = "")
```

Note: This behavior is not visible in the IDLE shell but will be visible with Python scripts.

We will learn more about the Python **print** function in future lectures.

The “input” function

It’s common for programs to prompt a user for input. In Python, you do this with the **input** function.

The **input** function causes a Python program to stop and wait for user input from the keyboard. When the user enters a value and presses the Enter key, the input function accepts the value and returns a string containing what the user has typed as the value of the function.

Below, the **input** function is passed the argument string “Type anything: “. When executed, the input function displays the prompt **Type anything: .** When the user types “Hello World!” and presses Enter, the string “Hello World!” is read by the **input** function and the Python Shell redisplay the string as the returned value of the function.

```
>>> input("Type anything: ")
Type anything: Hello World!
'Hello World!'
```

It is common to want to capture the user's input for additional processing. The example below behaves similar to the one above only the return value of the **input** function is assigned to the variable *saying* and the value of the variable is then displayed with the **print** function.

```
>>> saying = input("Type anything: ")
Type anything: Hello World!
>>> print(saying)
Hello World!
>>>
```

Combining functions

The example below combines a number of the functions we've discussed.

```
>>> number1 = int(input("Enter a number: "))
Enter a number: 8
>>> number2 = int(input("Enter another number: "))
Enter another number: 3
>>> print("The sum is", number1 + number2)
The sum is 11
```

When the first line is typed and you press Enter, the **input** function is passed the string "Enter a number: " which is displayed the screen. When you type 8, the input function reads the value and returns it as the string "8". However, rather than display the value to the screen, the string "8" is passed as input to the **int** function, which converts the string "8" to the integer 8. The integer 8 is then assigned to the variable *number1*. A similar thing happens with the variable *number2*.

The final **print** statement prints the string expression "The sum is", a single space, and the sum of the expression *number1 + number2*.

Benefits of IDLE

Color-coded elements

There are many benefits to using an IDE like IDLE rather than a Terminal Command Prompt. One of those benefits are **color-coded** elements. You may have noticed that the IDLE examples we've shown are multi-colored. IDLE colors the components of the command line to make it easier for the programmer to pick out different elements in the code.

For example, output displayed by IDLE is colored **blue**, strings entered at the `>>>` prompt are colored **green**, and built-in functions are colored **purple**. This color coding can help programmers understand how IDLE and the Python interpreter will interpret their code.

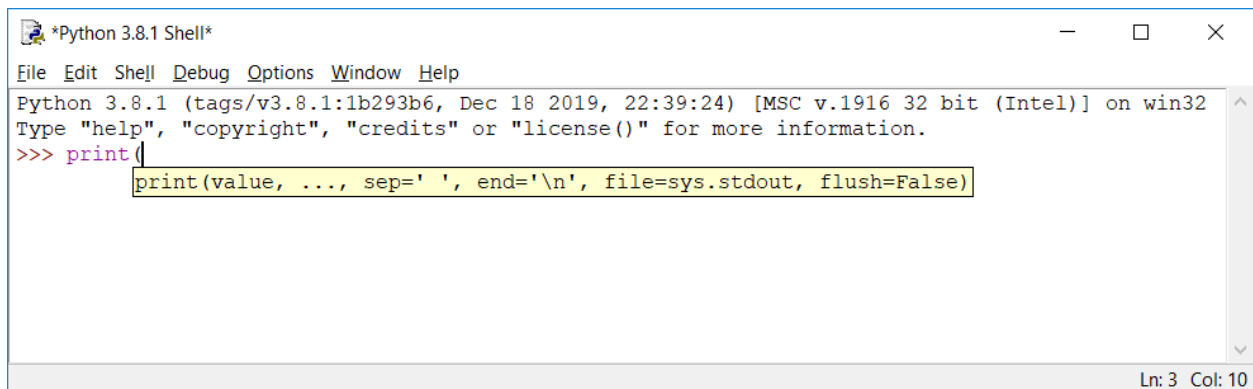
The color-coding rules used by IDLE:

Color	Type of Element	Examples
Black	Input into the IDLE shell including: numbers, operators, variable names, function names, method references and punctuation marks.	11 + y = factorial(x) total
Blue	Output from the IDLE shell including function, class and method names in definitions	'Hello World!' def factorial(n)
Green	Strings input into the IDLE shell.	"Hello World!"
Orange	Keywords	def, if, while
Purple	Built-in function names	print, round, int
Red	Program comments and error messages in the IDLE shell.	# This is a program comment

Context-Sensitive help (Intellisense)

Another benefit of using an IDE like IDLE rather than a Terminal Command Prompt is an automatic, **context-sensitive** help sometimes referred to as **Intellisense**.

Below, at IDLE's >>> prompt, the word **print** has been typed along with an open parenthesis, (. When IDLE sees the open parenthesis it realizes that **print** is the name of a function and it looks up the **syntax** (grammatical definition) of the **print** function and displays the syntax beneath the statement in a yellow box, as seen below.



```
*Python 3.8.1 Shell*
File Edit Shell Debug Options Window Help
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print(
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

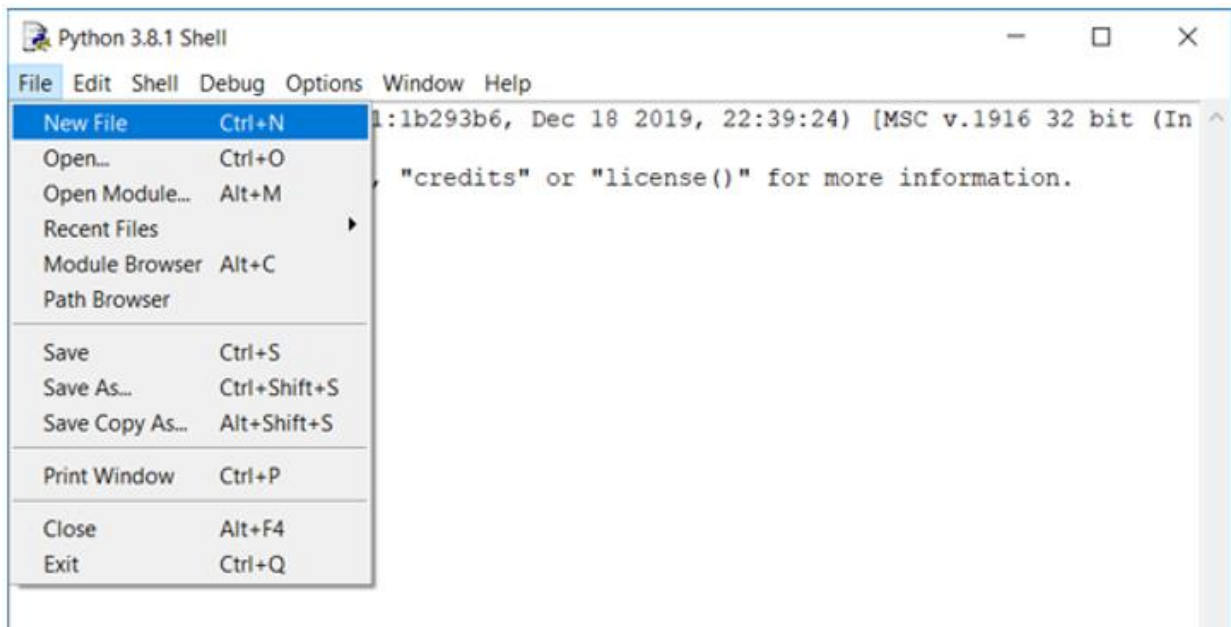
Context-sensitive help gives the programmer quick access to help when they are developing code. Rather than searching for help on a topic, the context-sensitive help evaluates what the programmer is trying to do and displays help that is appropriate for the current statement. Given the number of Python functions and statements, context-sensitive help can be a big time saver.

Python Scripts

The Python Interactive shell is a useful tool for evaluating simple expressions or short snippets of code, however, when you're writing a program that's more than a few lines, you will find it easiest to write a Python **script**.

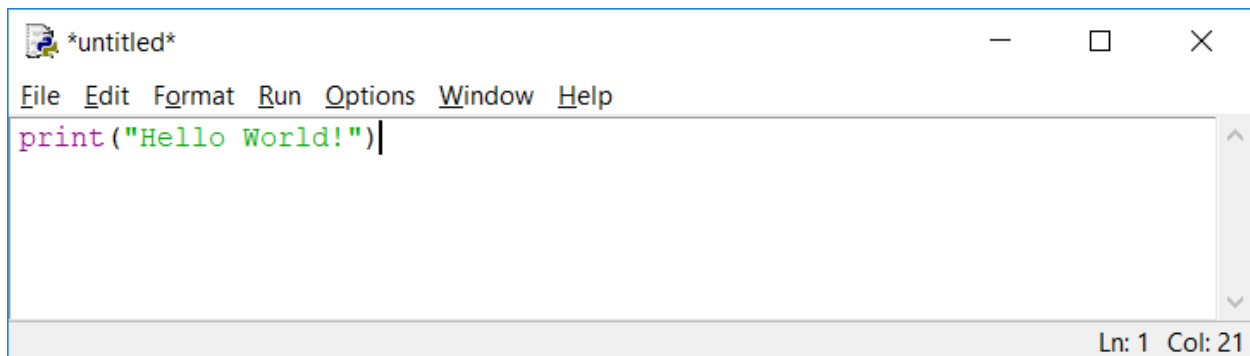
A Python script is really nothing more than a plain text file containing Python source code statements, similar to what you would type at the Python Interactive Shell. Python scripts allow you to edit, save and rerun your commands, as well as make it easy to share your program with others.

There are multiple ways to create a Python script. One way is to create one from within the menus of the IDLE Interactive Shell. To do this, start the IDLE Interactive Shell as discussed earlier and select **New File** from the **File** menu (or type **Ctrl+N**)

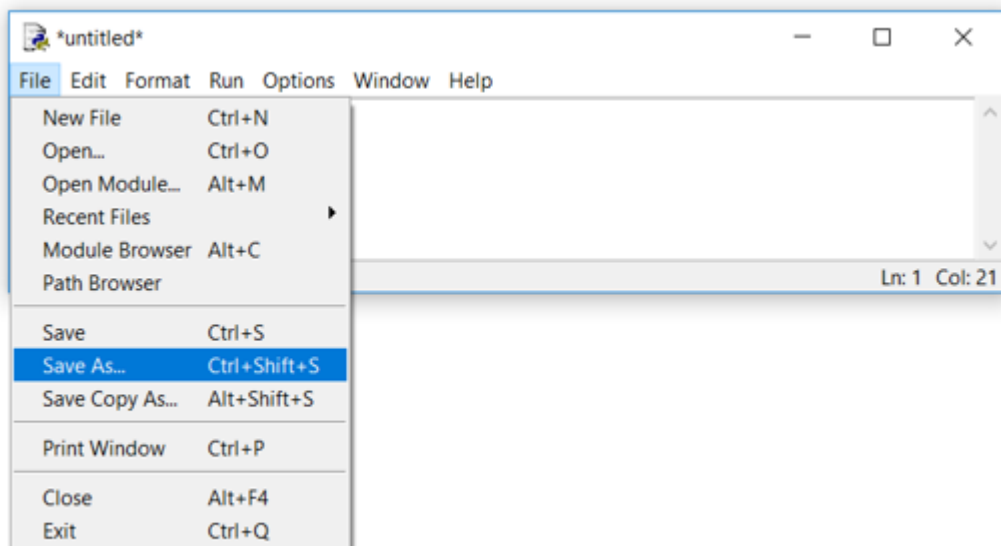


When you select **New File** from the **File** menu (or type **Ctrl+N**), a new window labeled *untitled* is displayed for you to enter your Python source code.

Below I've entered a simple Python program consisting of a single statement (We will, of course, be creating much more complex scripts in the future. 😊)

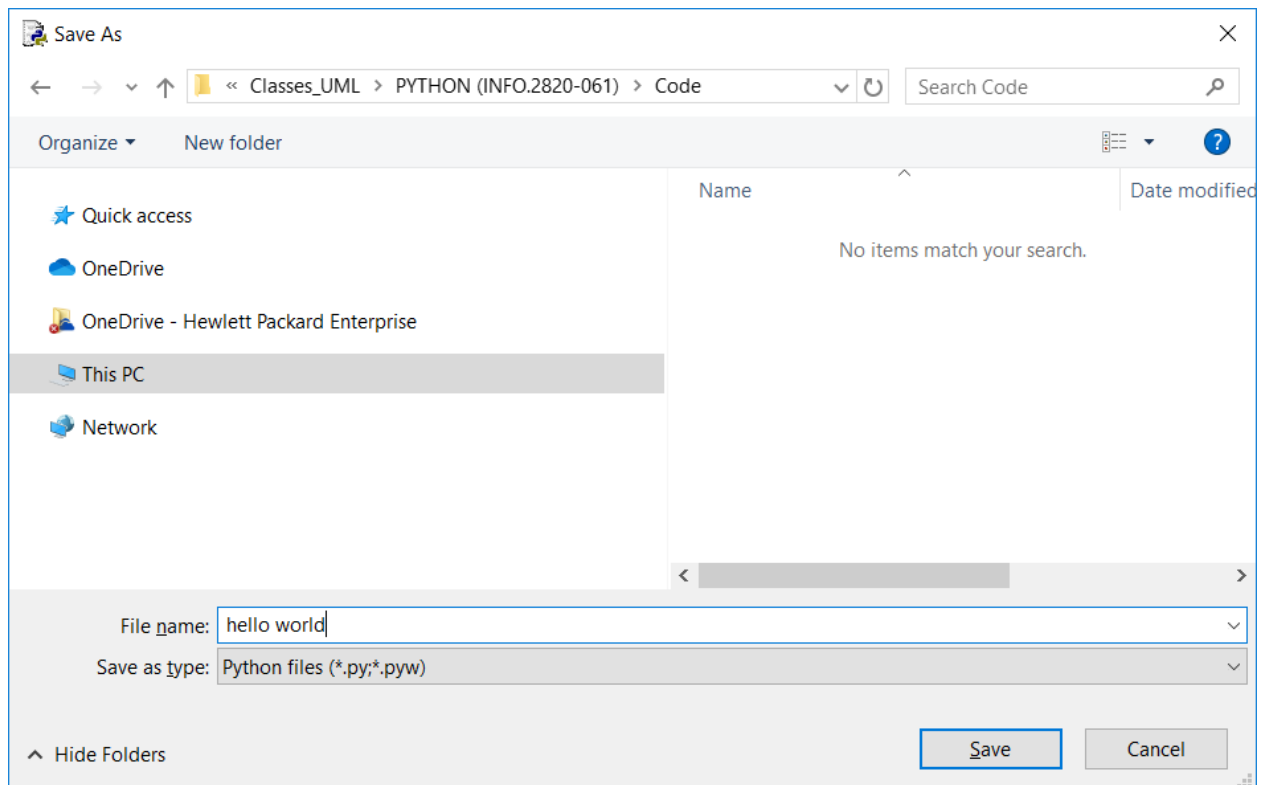


When you're done entering your source code, you can save the file by selecting **Save As...** from the **File** menu (or typing **Ctrl+Shift+S**). When the **Save As** window is displayed, enter a name for your script.

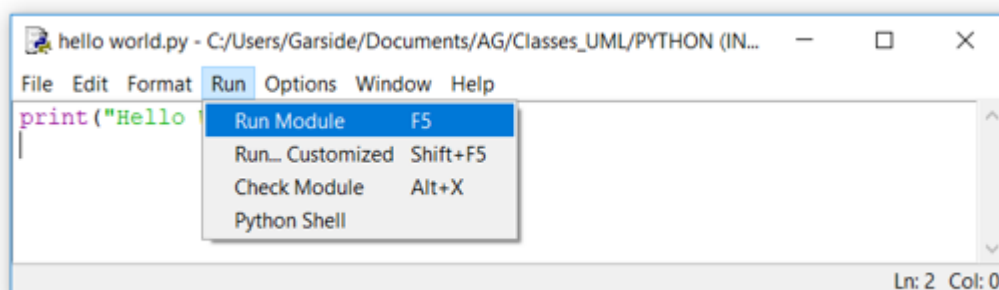


Below, I've entered the name "hello world". Notice that the line **Save as type** indicates **Python files**. This will cause the script to be saved as a file called **hello world.py**

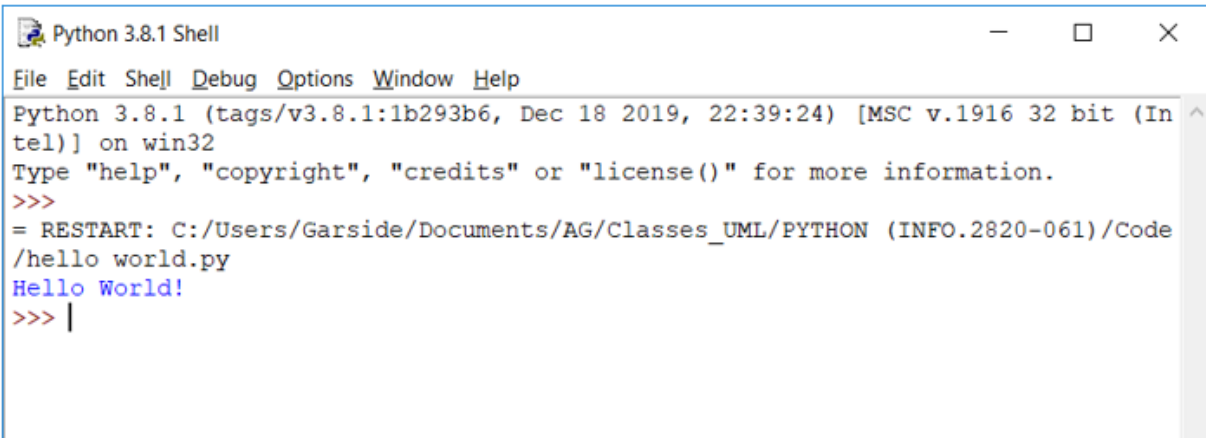
Notice too that I've changed the location of where my script will be saved. This is optional. By default, on my Windows PC, the program would be stored in **Local > Programs > Python > Python38-32**. Since I'd like to be able to easily find and view the program under my usual folders, I've created a **Code** folder and selected it as the destination.



I can now run my Python script by selecting **Run Module** from the **Run** menu of the source editor (or pressing **F5**)

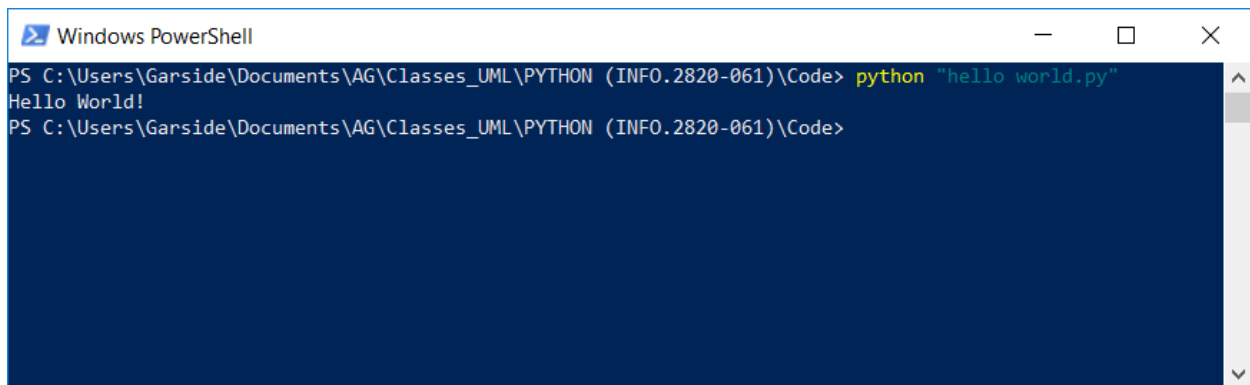


The program is then executed and the results are shown in the Python Shell, as shown below.



```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/Garside/Documents/AG/Classes_UML/PYTHON (INFO.2820-061)/Code/hello world.py
Hello World!
>>> |
```

You can also run your Python script from a Terminal Command Window like the Windows PowerShell by changing into the folder that contains your script and typing **python "hello world.py"**



```
Windows PowerShell
PS C:\Users\Garside\Documents\AG\Classes_UML\PYTHON (INFO.2820-061)\Code> python "hello world.py"
Hello World!
PS C:\Users\Garside\Documents\AG\Classes_UML\PYTHON (INFO.2820-061)\Code>
```

Note: you can run the Python script by opening a file explorer window and double-clicking on the file name, **hello world.py**. However, when you do so, the Python program will be run in a new Terminal Window and the Terminal Window will immediately close upon completion of the program. This makes it difficult to view any program output. For this reason, it's usually best to run the program from within IDLE or from a Terminal Command Window.

Comments in Python Scripts

When you write a script it's always a good idea to add **comments** to the source code describing what the script does. Comments allow the next reader of the script to understand the intentions of the source code and make it easier for future readers to modify the code.

In Python, all characters following a **#** are treated as a comment. As we saw earlier when we discussed IDLE and the color-coded elements, comments in IDLE are displayed in **red**.

In your source code, you can make an entire line a comment by doing something like

```
# This line is a comment!
```

Or you can add a comment to the end of a source code line by doing something like this

```
saying = input("Type anything: ") # Gather user input
```

Syntax Errors

No one is perfect, and when you write Python code, you will sometimes write statements that are invalid. These kinds of errors are referred to as **syntax** errors.

Below are two Python statements we used earlier. The only difference here is the letter 'a' is missing from the word *saying* within the **print** command. Since Python expects a valid expression to be passed to the **print** function, it prints a **syntax** error in the form of a **Traceback** message indicating that the name (variable) '**syng**' is not defined. Command execution is then halted.

```
>>> saying = input("Type anything: ")
Type anything: Hello World!
>>> print(syng)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print(syng)
NameError: name 'syng' is not defined
```

Syntax Errors due to indentation

Unlike most programming languages, in Python, **line indentation** is critical. I can't stress this enough. I promise there will be many times when you'll receive a Python syntax error due to incorrect line **indentation**.

Below, the first statement is syntactically correct and produces the output `Yes!`. The second statement is exactly the same except for a single space added before the **print**. As a result, the second statement produces a **SyntaxError: unexpected indent** and execution halts.

```
>>> print("Yes!")
Yes!
>>> print("Yes!")
SyntaxError: unexpected indent
```

There are rules to indentation that we will discuss in future lectures. For now, unless you know when you need to do it, avoid adding initial spaces to source code lines to avoid Python syntax issues.

Why does Python care about indentation? Unlike other programming languages, Python does not use special characters, like curly braces `{ }`, to define blocks of code. Instead, to define code blocks, Python requires that you indent the code and it uses the indentation to know what code is in the block. We will discuss more on indentation and when to use it soon.

Conclusion

This concludes our introduction to Python. We've compared Python to other programming languages, looked at Python **scripting** and the **Interactive Shell**, examined simple **expressions**, **variables**, and functions, including the **type**, **int**, **float**, **print** and **input** functions, and we've introduced the **IDLE** IDE and some common **syntax** errors. Not a bad start!