Python Programming

INFO.2820.061

Instructor: Andy Garside

Data Types, Expressions, Variables, Assignments, and using Functions and Modules

Data Types

In a written or spoken language, numbers and letters are treated similarly. For example, in English, we write or speak the statement 3 + 9 = 12 the same way we do the statement "Today is the day!" However, in computing, numbers and strings are typically stored and processed differently. These different forms are called **data types** and each data type has its own set of permissible values and allowable operations.

Note: Python is a **strongly typed** programming language. This means Python expects compatible data types whenever an operation is performed and it does not automatically convert, for example, the string "3.145" to a number if you attempt to use it in an arithmetic expression. In Python, the interpreter validates the data types of each operand before an operator is applied to those operands. If the operand type is not appropriate for the operation, execution halts and an error message is displayed.

Numeric data types

Numeric data types represent numbers. In Python, the two numeric data types we'll see are integers (**int**) and floating-point numbers (**float**).

Integers (int)

Integers include positive and negative whole numbers and the number 0.

In many programming languages, integers are stored as 32-bit or 64-bit quantities and the size of an integer is limited by what can be stored in either

32 or 64 bits. In Python, integers are stored as arrays of digits and are limited only by the amount of memory on the computer. As a result, Python integers can be much larger in magnitude than those of many programming languages.

What about performance? To achieve the highest performance possible, Python integers that fit into 32 or 64-bit hardware values are stored and treated similar to traditional 32-bit and 64-bit integers. This gives Python great performance on traditional integer sizes while allowing it to also handle much larger integers when necessary.

For example, the largest signed integer that can be represented on a computer with a 64-bit hardware integer is $2^{64} - 1$ = 9,223,372,036,854,775,807. However, Python can represent significantly larger integers. Using the IDLE IDE, below, the integer 123456789 is raised to the 100th power.

Note: In Python, the ** operator is the exponentiation operator and it is defined as A ** B = A^B.

```
>>> 123456789 ** 100
14174172601035587702142524239761426685023098432892168330190482375
94757708238986182489372231899746980921982728329402793285767462862
88246412163586040073071625403994235108484654701851813111412522017
07343655197746818256635550809600884481877003066625910338354975470
65849829393393851336850513516716654954594842407071059129565467264
69283110832013048361267258779772354758935887424049533897842706489
17007984590282408898177399292436203902925002036796208649715339142
60827834601579209314189120626901904458486936727622905582367388818
32546715962674705459956895378670356212279994168084514114818989630
51046413448394572238305079017627185288673969817759651765547006983
56765830690713630912519126290583362303892345035739308972274807594
10337695934859367858714793296706039210143078982981706101059862119
667407317346718937443597566001
```

Floating-point numbers (float)

Floating-point numbers (also called “**real**” numbers) consist of a whole number, a decimal point, and a fractional part. For example, 3.14159 is a floating-point number in **decimal notation**. 1.2×10^{12} is a floating-point number in **scientific notation** or **exponential notation**.

Decimal and scientific notation

As a reminder, any decimal number can be written in scientific notation and vice versa. Below are some examples of decimal numbers, their equivalent scientific values, and their mathematical meanings.

Decimal Notation	***Scientific Notation***	***Meaning***
3.78	*3.78e0*	*3.78×10^{0}*
37.8	*3.78e1*	*3.78×10^{1}*
3780.0	*3.78e3*	*3.78×10^{3}*
0.378	*3.78e-1*	*3.78×10^{-1}*
0.00378	*3.78e-3*	*3.78×10^{-3}*

In mathematics, floating-point numbers have unlimited precision because their fractional part can continue forever and their exponents are unbounded. However, in programming languages, these numbers have limits.

In Python, floating-point numbers are stored as 64-bit quantities. This is common in most modern programming languages. For example, it is equivalent to the C language data type **double**.

A relationship between Python and C. When you execute Python source code your program is compiled into bytecode (automatically) and the bytecode is then interpreted in an evaluation loop. The compiler/interpreter that does this work is called CPython (there are multiple such implementations but CPython is the original reference implementation). This compiler/interpreter called CPython is written in C.

To represent floating-point numbers using 64-bit quantities, the fractional part of the number is represented using 52-bits and the exponent is represented using 11 bits (the “sign” of the number is represented with the remaining bit). As a result, the magnitude of a floating-point number (**float**) in Python is slightly less than 1.8×10^{308}.

If you try to create floating point values that are too large, Python will give it the value “**inf**”, which is short for **infinity**. The value **inf** is intended to convey that the value is too large for Python to compute. For example,

```
>>> 1.7 * 10 ** 308
1.7e+308
>>> 1.9 * 10 ** 308
inf
```

Arithmetic Expressions

An **arithmetic expression** is a collection of operands and operators combined to form a numerical result. In Python, this collection consists of a familiar set of mathematical operators including those in the table below.

In the table below, the dash (-) is used to represent two operations, negation and subtraction. When the dash (-) directly precedes a literal number or numeric variable it implies negation. When it separates two values which are either literal numbers or numeric variables, it implies subtraction.

Operator	Meaning	Example
-	Negation	>>> x = -3
-	Subtraction	>>> 5 - 3 2
+	Addition	>>> 5 + 3 8
*	Multiplication	>>> 5 * 3 15
/	Floating-point Division	>>> 5 / 3 1.6666666666666667
//	Quotient (whole part of division)	>>> 5.2 // 3 1.0
%	Modulus (Fractional remainder of division)	>>> 5.2 % 3 2.2
**	Exponentiation	>>> 5 ** 3 125

Python's **arithmetic expressions** follow the same **order of operations** (**precedence rules)** found in algebra. The **order of operations** are

1. Exponentiation
2. Unary negation occurs before performing any multiplication, division or remainder.
3. Multiplication, Division (Floating-point and Quotient) and Modulus
4. Addition and Subtraction
5. Assignment

In Python, operations are evaluated left to right except for Exponentiation and assignment operations which are evaluated right to left. As in algebra, you can use parenthesis to change the order of evaluation.

Below are some Python arithmetic expressions and their evaluations

Expression	Evaluation	Value
5 + 4 * 2	5 + 8	>>> 5 + 4 * 2 13
(5 + 4) * 2	9 * 2	>>> (5 + 4) * 2 18
2 * 5 ** 3	2 * 125	>>> 2 * 5 ** 3 250
-5 ** 3	-(5 ** 3)	>>> -5 ** 3 -125
2 ** 3 ** 2	2 ** 9	>>> 2 ** 3 ** 2 512
45 / 0	Division by zero error	>>> 45 / 0 Traceback (most recent call last): File "<pyshell#66>", line 1, in <module> 45 / 0 ZeroDivisionError: division by zero

In all cases other than exact division, when both operands of an arithmetic expression are the same type (**int** or **float**), the resulting value is also of that type. When the operands are of different types, the resulting value is of the more general data type. For example,

```
>>> type(5 * 2)
<class 'int'>
>>> type(5.0 * 2.0)
<class 'float'>
>>> type(5 * 2.0)
<class 'float'>
```

Syntax and semantic (runtime) errors

Python generates a **syntax error** when a statement is malformed. These kinds of errors are discovered and displayed to the user when the source code is compiled into byte code. For example,

```
>>> x = 45 + * 3
SyntaxError: invalid syntax
```

A **runtime** or **semantic error** is generated when a **syntactically** correct program statement is executed and the result is **invalid**. Division by 0 is a good example,

```
>>> x = 2 - 2
>>> 5 / x
Traceback (most recent call last):
  File "<pyshell#70>", line 1, in <module>
    5 / x
ZeroDivisionError: division by zero
```

Mixed-Mode Arithmetic and Type Conversions

Above, we noted that when each numeric operand is of a different type, the resulting value is of the more general type. Performing calculations when the operands are mixed in this way (a mix of **int** and **float** types) is called **mixed-mode arithmetic**. When Python sees a mixed-mode equation, it temporarily converts the less general values, the integers, to the more general values, the floats, so it can do the computation.

Note that Python does not automatically convert strings to integers or floats. In our first lecture we learned that the **input** function prompts the user for input and returns the text entered by the user as a string. If we intend to use this returned value as a number, it must be explicitly converted to the desired type using either the **int**() or **float**() conversion function. For example,

```
# Get a “number” from the user.
>>> radius = input("Enter the radius: ")
Enter the radius: 3.2

# Notice radius is a string containing the characters ‘3.2’
>>> type(radius)
<class 'str'>
>>> radius
'3.2'

# Notice exponentiation fails using a string.
>>> radius ** 2 * 3.14
Traceback (most recent call last):
  File "<pyshell#78>", line 1, in <module>
    radius ** 2 * 3.14
TypeError: unsupported operand type(s) for ** or pow(): 'str' and
'int'

# If we explicitly convert radius to a float, everything works.
>>> float(radius) ** 2 * 3.14
32.153600000000004
```

Note: The **int**() conversion function always **truncates** a value. For example, **int**(3.75) is 3. If desired, you can **round** a decimal number to an integer, rather than **truncate** it, using the **round**() function. For example, **round**(3.75) is 4.

The String data type

The built-in data types of low-level programming languages like C are typically simple, often designed to correspond directly to a data type supported natively on the hardware. For example, the C data type ***int*** maps to either a 32-bit or 64-bit hardware integer, depending on the computer’s architecture. Similarly, the C data type ***float*** maps to a 32-bit, single-precision hardware floating-point data type and the C data type ***double*** maps to a 64-bit, double-precision hardware floating-point data type (if it exists on the hardware). More complex data types like **strings** (a sequence of characters treated as a unit) are not native to the C language. Instead, C strings are implemented as arrays and you must use functions in software **libraries** to manipulate them.

High-level languages, like Python, support more complex, built-in data types, like **strings**.

Literal Strings

In programming, a **literal** value is a text-based representation of a data type. For example, **literal** values of type **int** would be -1, 0, 1, 2, etc, and literal values of type **float** would be -0.21, .3333, 3.14, 6.0, etc.

In Python, a **literal string** takes the form of a character sequence enclosed in single or double quotes. For example **'Hello World!'** or **"Hello World"**.

Note: A **literal string** must always be enclosed in **matching** quotes. One advantage of this is it allows double quotes to be used to create a literal string containing a single quote. For example, **"Today's the day!"** You can also use single quotes to create a literal string containing double quotes like, **'He shouted, "Get out of the way!"'**

Below are examples of various strings and their representation in the IDLE shell.

```
>>> "Hello World!"          # enclosing double quotes
'Hello World!'
>>> 'Hello World!'          # enclosing single quotes
'Hello World!'
>>> "Hello World!'          # mismatched quotes
SyntaxError: EOL while scanning string literal
>>> "Today's the day"      # single quote within double quotes
"Today's the day"
>>> 'He shouted, "Get out of the way!"'  # double quotes within single quotes
'He shouted, "Get out of the way!"'
```

Note that it is possible to have a string that doesn't contain any characters at all. This is called the **empty string** and is denoted as **""** or **''**. We will see that the empty string is used fairly frequently in programming, including when you are determining if a user provided a meaningful string to the **input** function.

Note: an **empty string** is different than a **blank string**. An **empty string** contains no characters and is created by entering a set of matching quotes with no intervening characters, while a **blank string** contains one or more spaces or tabs between the matching quotes.

Below are some empty and blank literal string examples

```
>>> “”       # empty string
''
>>> ''       # empty string
''
>>> " "      # blank string
' '
```

Recall from our first lecture that output from the Python **print** function has enclosing quotes removed from the string. For example,

```
>>> print("Today's the day!")
Today's the day!
```

When viewing the output of **print** this makes it impossible to visually see the difference between an **empty string** and a **blank string**.

```
>>> print("")

>>> print(" ")
```

Line Continuation and Multi-line Literal Strings

When you create a **literal string** using single or double quotes the entire string is, by default, displayed on one line. For example,

```
>>> print("This is a very, very, very, very, very long line!")
This is a very, very, very, very, very long line!
```

If a string is long and you'd like to break it across multiple lines in your source code yet still have it displayed as a single line of output, you can use Python's **line continuation** character, the backslash, \. The **line continuation** character must appear as the last character in the line (without any following spaces or tabs). For example,

```
>>> print("This is a very, very, very,\
 very, very long line!")
This is a very, very, very, very, very long line!
```

In a program, it is sometimes desirable to write a string on one line and have it displayed as output on multiple lines. This can be done using the **escape sequence, \n,** which represents a **Newline** in a string.

```
>>> print("This is a very, very,\nvery, very, very long line!")
This is a very, very,
very, very, very long line!
```

In Python, newlines can also be enforced without using the **\n escape sequence** by enclosing the entire string literal, line breaks and all, within **three consecutive quotes** (either single or double quotes). For example,

```
>>> print("""This is a very, very,
very, very, very long line!""")
"This is a very, very,
very, very, very long line!
```

Note: Unlike the **print** function, if you use **\n** in a string within the IDLE shell, or you use the **three quotes** technique, the Newline is displayed as a **literal \n**, as seen below.

```
>>> """This is a very, very,
very, very, very long line!"""
'This is a very, very,\nvery, very, very long line!'
```

Escape Sequences

In most programming languages, special characters within a string, like the **tab**, **backspace** or **newline,** are represented by a two-character sequence called an **escape sequence**. For example, **\n** is used to represent a **newline** character.

Below is a list of some of the more commonly used escape sequences and their meanings in Python.

Escape Sequence	Meaning
\b	Backspace (delete key)
\n	Newline
\t	Tab (horizontal)
\’	Literal Single quote
\”	Literal Double quote
\\	Literal Backslash

Note: in a string, the characters **b**, **n** and **t**, are literal characters that are made special by preceded them with a backslash. In an inverse way, the **single quote**, **double quote** and **backslash** usually have special meaning and are made literal characters by preceding them with a backslash.

String Operators

When we say a data type is built-in we mean the data type is recognized by the language and there are a set of built-in operators that can be applied to the data type.

For example, the built-in data type **int** (integer) includes native operators for addition, subtraction, multiplication, division, and so on. In Python, the built-in data type **string** also has a set of operators. We discuss two of these operators below.

Concatenation: the + operator

In Python, the operator **+** is used to join two or more strings to form a new string. For example,

```
>>> "This is a very, very," + " very, very, very long line!"
'This is a very, very, very, very, very long line!'
```

Note: The **+** operator cannot be the last character in a line. Just as the integer operation 3 + 5 must appear on one line, the string operation “str1” + “str2” must appear on one line.

```
>>> "This is a very, very," +
SyntaxError: invalid syntax
```

Repetition: the * operator

In Python, the ***** operator is used to repeat a preceding string a number of times. For example, **“very “ * 5** means to repeat the string “very “ five times.

Below the * operator is used with the + operator to create our very long line using a much smaller expression.

```
>>> "This is a " + "very " * 5 + "long line!"
'This is a very very very very very long line!'
```

Converting numbers to strings

As stated earlier, Python does not automatically convert strings to numbers. Conversely, it does not automatically convert numbers to strings. To convert a number to a string, you use the **str**() function.

Consider you have a floating-point variable named *cost* that holds the cost of an item in a store and you want to print this value in a string preceded by a dollar sign ($), similar to the following: “The cost is $<cost>”. As shown below, if you attempt to concatenate the floating-point value cost with the string containing the dollar sign ($) using the **+** operator, Python will generate a Traceback error indicating that you cannot concatenate a string and a float.

```
>>> cost = 5.32

>>> print("The cost is $" + cost)
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    print("The cost is $" + cost)
TypeError: can only concatenate str (not "float") to str
```

In order to get the desired result, you must explicitly convert the floating-point cost to a string using the function call **str(cost)**,

```
>>> print("The cost is $" + str(cost))
The cost is $5.32
```

Strings verses Characters

In some programing languages, strings and individual characters are viewed differently. In Python, strings and individual characters are both the same data type, **str**. That is, a character in Python is simply a string containing a single character.

Our textbook uses the convention of single quotes around a character, like ‘A’, and double quotes around a string, like “Apple”. However, the two objects are both strings and both are of the data type **str**. In our course notes we will follow this convention, but realize this is simply to make it visually clear that we’re discussing a single character string verses a multi-character string.

Variables and Assignments

Variables

We briefly introduced Variables and assignments in our first lecture. Let’s look at them in more detail here.

In our first lecture we learned that a Python **variable** is a name that refers to an object that holds information. The name is used within Python code 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	

Python programmers typically use **lower-case** characters for variable names. For example, **name**, **radius**, and **date**.

When a variable name contains two or more words, they are typically either separated by underscores, as in **user_name**, **circle_radius**, and **original_closing_date**, or they are concatenated together with the first letter of each word capitalized (other than the first word) as in **userName**, **circleRadius**, and **originalClosingDate**. This last format is also known as **camelcase**, as it reflects a series of humps in the variable name caused by the intermittent capital letters.

Unlike some programming languages, Python does not support a **constant variable**. A constant variable (or **symbolic constant**) is a variable that is given an initial value which cannot be changed. Instead, by convention, if a Python programmer wants to create a variable and request that it never be changed, the variable is given a name consisting only of **upper-case** characters, numbers and underscores. For example, SPEED_OF_LIGHT or CONST_GRAVITY.

The assignment statement

Variables are given their **initial** value and can be reset to a new value using the **assignment** statement. In its simplest form, the assignment statement looks as follows

<variable name> = *<expression>*

where <variable name> represents any valid Python variable name and <expression> represents any valid Python expression. For example

```
>>> x = 5

>>> name = "Stephen Hawking"
```

Note: In Python, a single equals sign, **=**, is used for assignment while the double equals sign, **==**, is used for comparison. For example,

```
>>> x = 5
>>> y = 6
>>> x == y
False
```

Unlike some languages, for example Visual Basic, variables in Python must first be explicitly assigned values before they can be used. When the Python interpreter encounters a variable name it looks up the variable's value. If the variable is not yet **defined** (not yet **bound** to a value), a Traceback is displayed with a NameError.

```
>>> print(closing_date)
Traceback (most recent call last):
  File "<pyshell#49>", line 1, in <module>
    print(closing_date)
NameError: name 'closing_date' is not defined
```

The **type** of a variable (for example, a str, int, float, etc) is **inferred** by the context of its use. The programmer does not declare a Python variable's **type**.

The example below creates a Python variable named *saying* and assigns it the value "Hello World!". In the example, Python considers the variable *saying* as a string type (str) because it is assigned a series of characters enclosed in quotes.

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

While it sounds trivial, choosing descriptive variable names is important. Well-chosen variable names makes source code much easier to read and, at the same time, reduces the need for additional comments.

Program comments and Docstrings

Our previous lecture introduced the concept of a one-line comment in Python which is any sequence of characters preceded by the # character. For example

```
# This line is a comment!

area = 2 * radius * 3.14      # Calculate the circumference of a circle
```

Any Python script you write should make good use of comments, particularly to explain areas of a program that are complex and/or difficult to understand.

Docstrings

In any programming language, it is convention that the program and any of its functions include an initial description of their purpose so that readers of the source code can understand the program without needing to read and understand all the lines of the source code. Python follows this same convention and introduces an enhanced documentation feature called a **docstring** (documentation string)

A **docstring** is a comment enclosed in **three consecutive quotes** that appears either at the top of a program module or immediately following a function's definition. For example, below is a docstring that might appear at the top of a program module that calculates the circumference of a circle.

```
"""
Program: circumference.py
Author: Jim Foster
Date: 5/13/2020

This program calculates the circumference of a circle.
Input: The radius of a circle as an int or floating-point number.
Output: The circumference of a circle with the supplied radius.
"""
```

In Python, when the initial comment in a file is a **three consecutive quote** comment, the comment is also automatically stored in an attribute with the name __**doc**__ which can be accessed from within a running python program, making it unnecessary to view the original source file to access the comment.

For example, when the comment above is at the top of a python script, using the following **print** function call within the script will display the comment.

```
print(__doc__)
```

```
Program: circumference.py
Author: Andy Garside
Date: 1/13/2020

This program calculates the circumference of a circle.
Input: The radius of a circle as an int or floating-point number.
Output: The circumference of a circle with the supplied radius.
```

When we discuss creating Python functions and using existing Python modules we will see how this capability can be useful.

Using Functions and Modules

We have, at a number of points in our first two lectures, made use of **functions**. Now we'll cover them in more detail and discuss the incredibly rich set of functions available to Python programmers in what are known as Python **modules**.

Function calling, Arguments and Return Values

The concept of a **function** is common to all high-level programming languages. The purpose of a **function** is to gather a set of statements together so they can be called from one or more locations in a program, thereby making it possible to reuse code.

An advantage of a function, besides eliminating code redundancy, is that all locations that call the function benefit from any future improvements made to the function itself. For example, performance improvements, bug fixes, etcetera.

In Python, functions **return** a result and may have one or more **required arguments**. The arguments themselves can be any **expression** that evaluates to the data type expected by the function. For example, the Python **round** function requires a number as an **argument** and **returns** the rounded value as a result. The argument can be a numeric **literal value**, a numeric **variable**, or an **arithmetic expression**. For example,

```
>>> round(3.14159)    # round 3.14159 to the nearest whole number
3

>>> x = 3.14159    # assign the variable x the value 3.14159
>>> round(x)       # return the rounded value of x.
3

>>> round(3 + 0.14149)   # Add 3 + 0.14159 and round the result
3
```

It's important that function arguments be of the correct data type. When an incorrect data type is passed, the result is a **Traceback** with a **TypeError**. For example, if the **round** function is passed a **string** rather than a number, the following error occurs.

```
>>> round("wrong!")
Traceback (most recent call last):
  File "<pyshell#105>", line 1, in <module>
    round("wrong!")
TypeError: type str doesn't define __round__ method
```

Since the **return value** of a function also has a data type, you can use a function as an expression as long as the function returns the required data type.

Below, the variable *pi_str* is created as the string “3.14159”. Since the **round** function requires a number rather than a string, *pi_str* is first passed as an argument to the **float** function, which converts the *pi_str* string to a floating-point value. The floating-point value, 3.14159, is then returned from the **float** function and passed as an argument to the **round** function. Finally, the **round** function rounds the value to the nearest integer, returning the number 3.

```
>>> pi_str = "3.14159"
>>> round(float(pi_str))
3
```

Required and Optional Parameters

Unlike some programming languages, in Python, it is common to have both **required** and **optional** parameters to a function.

For the **round** function, the number to be rounded is a **required** parameter. The **default behavior** of the function is to round this parameter’s value to the nearest whole number. However, there is an **optional** second parameter that indicates the number of significant digits to round to. Below, the **optional** second argument, 3, indicates that the **required** first argument, 3.14159, should be rounded to 3 significant digits after the decimal point.

```
>>> round(3.14159, 3)
3.142
```

Optional arguments greatly simplify calling Python functions since the programmer only needs to provide the **required** arguments and can specify **optional** arguments only when necessary.

Function help

As a programmer, you can do a Web search to find the definition of a built-in or widely used Python function. However, since functions change over time (a perfect example being the **print** function in Python 2 versus Python 3) a good way to know the definition of a function on your current platform is to use the Python **help** function. The **help** function takes the name of a function and returns its **docstring**, which typically includes both the required and optional parameters.

Below, the **help(round)** statement returns the **docstring** of the round function. The text shows that the first parameter, *number*, is a number, and the second parameter, *ndigits=None*, is the number of digits to round to. The component "=None" indicates that *ndigits* is given the default value *None* if no value is specified, meaning the parameter is optional.

```
>>> help(round)
Help on built-in function round in module builtins:

round(number, ndigits=None)
    Round a number to a given precision in decimal digits.

    The return value is an integer if ndigits is omitted or None.
    Otherwise the return value has the same type as the number.
    ndigits may be negative.
```

Modules

In modern programming languages it is common to gather similar functions and other resources into groups that can be conveniently included in a program. In Python, this is done with **modules**.

Some functions, like **round**, are part of a module called **__builtin__**, which is automatically imported into all Python programs and, thus, always available for the programmer to use. Most other modules must be explicitly imported into a program using the **import** key word.

The math Module

In order to perform most complex mathematical functions in Python you must **import** the **math** module. Once imported, you can access the desired math functions and associated resources by writing the name of the module, followed by a **dot** (**.**), and the name of the function or resource.

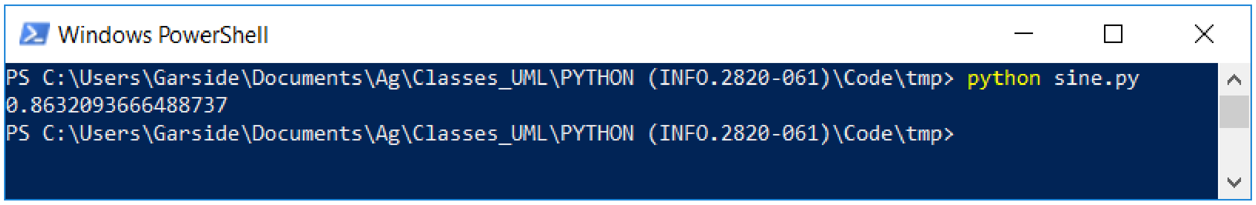
Below is a simple Python script called ***sine.py***. The script imports the math module, calculates the sine of 2.1 and displays the result.

```
import math         # import the math functions
y = math.sin(2.1)  # call the sine function with the argument 2.1
print(y)           # print the result
```

When the script is run (either within IDLE or at the Terminal Command Prompt) it displays the result:

```
0.8632093666488737
```

For example:



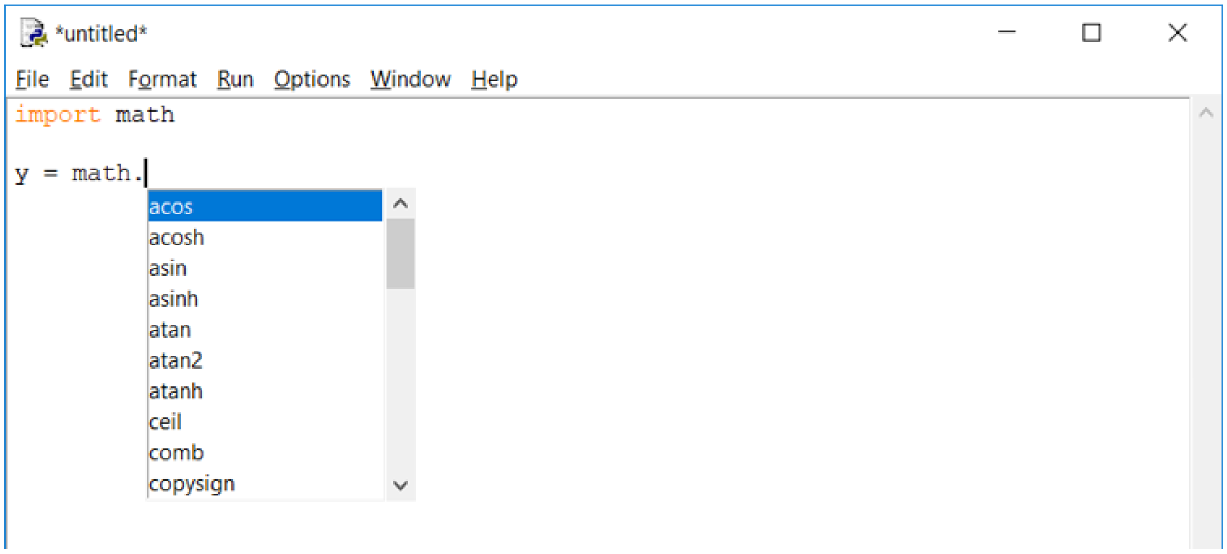
```
PS C:\Users\Garside\Documents\Ag\Classes_UML\PYTHON (INFO.2820-061)\Code\tmp> python sine.py
0.8632093666488737
PS C:\Users\Garside\Documents\Ag\Classes_UML\PYTHON (INFO.2820-061)\Code\tmp>
```

Viewing the functions and resources of a module

Once a module is imported, you can view the list of functions and resources it supports using the **dir** function, as shown below. Of course, you can find a similar list by doing a Web search. However, since modules can vary over time, the best way to know which functions exist for your current platform is to use the **dir** function.

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos', 'cosh',
'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod',
'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan',
'isqrt', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'perm', 'pi',
'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

Note: If you're using IDLE to create your Python scripts, you can type Ctrl+backspace after you type a module name and the dot(.) to view a list of the functions available in the module, as seen below:



```python
import math

y = math.
```

Importing individual resources from a module

If you are only using a few functions from a module, you can import just those functions using the following syntax

```
    from <module> import <func1>, <func2>…
```

When you do this, you no longer have to specify the name of the module and a **dot** (**.**) to call a function in the module. For example, the ***sine.py*** script can be rewritten to import just the sin module, eliminating the need to write **math.sin(2.1)** and replacing it with simply **sin(2.1)**.

```python
from math import sin
y = sin(2.1)
print(y)
```

In a similar manner, you can import all of the functions of a module using the following syntax:

```
    from <module> import *
```

Note: there are advantages and disadvantages to using `import *`. The biggest advantage is that it imports all of the functionality of the module making it easy for the programmer to call any of the module's functions. However, it also adds all of the module's functions to the program's name space. As a programming practice, it's usually best to import just those functions that are being used.

The Main module

In Python, the **main module** is the script that is passed to the Python interpreter from which all other modules are imported. For example, the **sine.py** module we created earlier is a main module. To run **sine.py** from the Terminal Command Prompt we would type,

```
python sine.py
```

Like any module, a **main module** can be imported. Below, the **sine.py** module is imported into the IDLE IDE. This results in the statements within the module being read and evaluated.

```
>>> import sine
0.8632093666488737
```

Putting it all together

Below is a Python script that uses many of the concepts we've discussed to date. The script calculates the hypotenuse (long side) of a right-triangle given the length of the triangle's base and height. The script…

1. Begins with a docstring header explaining the program.
2. imports the **math** module.
3. Prompts the user for the length of the triangle's base and height.
4. Calculates the hypotenuse using the formula hypotenuse = sqrt(base^2 + height^2).
5. Displays the results.

```python
"""
Program: Pythagorean
Author: Andy Garside

This program calculates the hypotenuse of a right triangle.
It prompts the user for the length of the base and height.

Input: The base and height of a right triangle
Output: The length of the long side of the triangle (hypotenuse)
"""

import math

# Get the length of the base and height of the triangle
base = float(input("Enter the base of the triangle: "))
height = float(input("Enter the height of the triangle: "))

# Calculate the hypotenuse = sqrt(base^2 + height^2)
hypotenuse = math.sqrt(base**2 + height**2)

# Display the results
print("The length of the hypotenuse is", hypotenuse)
```

Below is the output of the script using the values 3.0 and 4.0 for the triangle base and height, respectively. (This a classic, 3, 4, 5 right-triangle)

```
Enter the base of the triangle: 3.0

Enter the height of the triangle: 4.0

The length of the hypotenuse is 5.0

>>>
```

Conclusion

In this lecture we've looked at data types (including **string**, **int** and **float**), **arithmetic expressions**, **variables**, **assignments**, **calling functions** and **importing modules**. With these tools we can put together an almost infinite number of simple Python programs.