# Python Basics

Aidan Hunt, University of Washington

---

## Learning Objectives

After this lesson students will be able to:

- understand and interpret the basics of Python syntax (i.e., how to write text that Python can interpret)
- use mathematical operators to conduct basic math operations in Python
- identify basic data types used by Python
- construct strings and print them to the console

## Resources for this lesson

- Python documentation
  - **Section 3.1: Numbers**
  - **Section 3.2: Strings**
- A Whirlwind Tour of Python:
  - **Basic Python Syntax**
  - **Python Semantics: Variables**
  - **Python Semantics: Operators**
  - **Built-In Scalar Types**

---

## Quick intro to Jupyter Notebook

This is a Jupyter notebook. A Jupyter notebook is an interactive Python program that consists of several "cells" that can be run individually. Cells can contain code or plain text (also known as "markdown" cells). The text you are reading right now is in a markdown cell. The cell below is a code cell. To run it, click on it and press the "Run" button in the toolbar. Alternatively, click on it and then press Enter+Shift.

In [1]:
```
print("Hello world!")
```

```
Hello world!
```

When you run a code cell, the statements in the cell excecute. The above cell executes the print command and prints the enclosed text below. Cell outputs are also sometimes printed automatically if not assigned to a variable.

In [2]:
```
2 + 3
```

Out[2]:
```
5
```

A few comments on Jupyter Notebooks:

- Since cells are run individually, sometimes you might get errors if you run cells out of order (that is, if code in one cell depends on the outputs of another cell.
- Jupyter notebooks are great for demonstrating concepts (as in this notebook) or testing small chunks of code. However, for larger programs (scripts), I would recommend using Spyder or another IDE.

## Basic Syntax

"Syntax" are the basic rules that we use for writing code "statements" that Python can interpet and execute. For example, consider the code block below, which consists two statements:

In [3]:
```
x = 2 + 3

print(x)
```

```
5
```

In the first statement, the value of `2 + 5` is assigned to the "variable" (i.e., a container for storing data) `x` using an equal sign. In the second statement, the data contained in `x` is printed to the console. The statements are separated by a line break.

The amount of whitespace in a statement does not matter in Python. Similarly, the amount of blank lines between statements does not matter. For example, the following code is identical to that above.

In [4]:
```
x              =      2+3


print(x)
```

```
5
```

(However, note that the leading indentation of a statement DOES matter, which we will see later when discussing functions.)

If we want, we can put multiple statements on a single line using a semicolon ( `;` ) to indicate the end of the first statement and the start of the next. If the semicolon is omitted, an error is thrown.

In [5]:
```
x = 2+3; print(x) # Semicolon used: no error
```

```
5
```

In [6]:
```
x = 2+3 print(x) # No semicolon: syntax error!
```

```
  File "C:\Users\Aidan Hunt\AppData\Local\Temp\ipykernel_3908\2964206796.py", line 1
    x = 2+3 print(x) # No semicolon: syntax error!
            ^
SyntaxError: invalid syntax
```

---

## Comments

We can also add text that is not to be executed using the pound symbol ( `#` ). Importantly, this allows us to write comments that describe code. We can also "comment out" lines of code that we want to keep, but not execute for now.

In [ ]:
```
# The code below assigns the value of 2+3 to the variable x.
```

```
x = 2+3 # I can also write a comment after a statement

# Code that is commented out does not execute
# print(x)
```

---

# Operators

We can perform mathematical operations on the data stored in variables using various operators.

(*Side note*: In any IPython console like the cells below, if the output of a statement is not assigned to a variable, then the output is automatically printed).

In [ ]:
```
# Addition
5 + 10
```

In [ ]:
```
# Subtraction
5 - 2
```

In [ ]:
```
# Multiplication
4 * 3
```

In [ ]:
```
# Division
6 / 4
```

In [ ]:
```
# Exponential
3 ** 2
```

In [ ]:
```
# Standard order of operations still applies here
(3 + 2)**2 + 7
```

While the operations above are likely familiar, there are two other operators which may come in handy.

## Floor Division

Floor division or "integer division" divides a dividend by the divisor and returns the result rounded down to the nearest whole number. In other words, any decimal remainders are ignored.

(If you have programmed in other languages, like Java, this is how integer division works).

In [ ]:
```
10 // 4 # 4 goes into 10 2 times
```

## Modulo

The modulo operator ("mod for short") gives the remainder that is left out by a floor division.

In [ ]:
```
# The remainder of 10 divided by 4 is 2
10 % 4
```

One especially useful aspect of mod is the ability to programmatically distinguish even numbers from odd numbers:

```
64 % 2 # Even numbers have a remainder of 0 when divided by 2
```

In [ ]:

In [ ]:
```
9 % 2 # Odd numbers have a remainder of 1 when divided by 2
```

---

## Variables and Assignment Operations

Recall that we can assign a **value** to a **variable** using the `=` operator:

In [ ]:
```
x = 2 + 3
```

Here, `x` is a variable that stores the value of `5`. You can think of this as a box contains the value of "5" that has a label "x" on it. When we call `print(x)`, Python finds that data in x and executes the `print()` statement using that data.

Variables may be named anything so long as the name adheres the following rules:

- The name must start with a letter or underscore.
- The name must contain only numbers, letters, and underscores.
- The name cannot be a Python keyword.

Variables can also be used on the right-hand side of assignment statements. Here, the values in `x` and `y` are used to compute the value that should be stored in `z`.

In [ ]:
```
x = 5
y = 7
z = x + y
print(z)
```

**Question**: *What happens to `z` if I change the value in `x`?*

In [ ]:
```
# Print the current value of z
print(z)

# Change the values of x and y
x = 8
y = 402

# Print the current value of z
print(z)
```

**Answer**: Note that variables are, generally, independent of each other. Once `z` is assigned it has no connection to `x` or `y`.

(Later, we will see that there are certain types of data that this is not true for. However, for now, this can be accepted as true).

You can even set the value of an existing variable based on its current value by using it on both the left and right side of a statement:

In [ ]:
```
# Set the value of x and print it out
x = 5
print(x)
```

```
# Add 10 to the current value of x and print it out
x = x + 10
print(x)

# A shortcut for this kind of assignment
x += 10
print(x)
```

---

# Basic Data Types

All values that flow through a Python program have a "type", which describes what kind of data is contained in them. Python has several built-in data types. To determine the type associated with a variable or value, you can use the `type()` command.

For now, we will focus on three of the basic data types.

## Floats

Floats (short for floating-point values) represent real numbers, and are the most general type of numeric data in Python. Floats can be specified via decimal numbers, fractions, exponentials, etc.

In [ ]:
```
floatData = 1.625
print(floatData)
type(floatData)
```

In [ ]:
```
floatData = 1 + 5/8
print(floatData)
type(floatData)
```

In [ ]:
```
floatData = 2e3 # Same as 2 * 10**3
print(floatData)
type(floatData)
```

## Integers

Integers ( `int` data type) represent whole numbers (e.g., numbers without decimals):

In [ ]:
```
intData = 4
type(intData)
```

Note that even if a number is *mathematically* an integer, it will not be automatically represented by an integer if a decimal point is included.

In [ ]:
```
intData = 4.0 # 4.0 is still an integer, right?
print(intData)
type(intData) # Actually stored as a float
```

Thankfully, Python provides flexibility here and will convert integers to floats as necessary to perform mathematical operations.

## Strings

Strings ( `str` data type) store text data and can be created using either single quotes ( `'` ) or double quotes ( `"` ).

In [ ]:
```
someText = "This is a string"
type(someText)
```

In [ ]:
```
moreText = 'This is also a string'
type(moreText)
```

Strings can be concatenated together by using the `+` operator:

In [ ]:
```
someText + moreText # Note that whitespace is not automatically included.
```

In [ ]:
```
someText + '. ' + moreText + "." # Adding whitespace, mixing and matching quotes.
```

If you want to know how many characters are in a string (including whitespace), you can use the `len()` command. Note that unlike other programming languages, there is no `char` data type, only `str` .

In [ ]:
```
len(someText) # There are 16 characters in "This is a string", including whitespace.
```

## Type conversion

Sometimes we would like to convert data that is stored in one data type to a different data type. Python includes special commands to do this:

In [ ]:
```
# Example: convert a number to a string
str(5.0)
```

Note that this is what happens when we use a `print()` command with a number. The number is converted to text and printed to the console!

In [ ]:
```
numberData = 5.0
print(numberData)

textData = str(numberData)
print(textData)
```

We can convert text to numbers, too, but only if the string is valid:

In [ ]:
```
textData = 5.1275
float(textData)
```

In [ ]:
```
textData = 'This string is clearly not a number'
float(textData)
```

---

# Printing to the Console

Now that we have an understanding of operators, variables, and data types, we can dive into more of how to use the `print()` command. So far, we have used `print()` to print the values in our variables, but what if we wanted to be more descriptive?

In [7]:
```
x = 3
```

```
y = 5
z = x + y
```

We would like to print something like:

x = 3
y = 5

z = x + y = 8

We can do this by providing both "string literals" (plain text) and variables to the print function. Separating them by commas will concatenate them together automatically and add whitespace.

In [8]:
```
print('x =', x)
print('y =', y)
print()
print('z = x + y =', z)
```

```
x = 3
y = 5

z = x + y = 8
```

While we have mostly been printing the values contained in variables, note that a variable is not necessary to use `print()`: we can print any value directly.

In [9]:
```
print(1+2) # Print the sum of 1+2
print('Hello there') # Print literal text
```

```
3
Hello there
```

## Example:

Consider a program that prints someone's birthday given their name, the birthday month and the day of the month

In [10]:
```
name = 'Aidan'
month = 'September'
day = 27

# Desired output:
# Aidan's birthday is on September 27!
```

**In Groups**: Write a line of code that prints this output. Note different approaches.

Using the print function, we can combine the values of these variables, as well as some extra text that is not stored in a variable (these are called "string literals"):

In [11]:
```
# One solution, use the print command to generate the whitespace
# Note that print automatically performs the type conversion on the day
print(name + "'s", 'birthday is on', month, day)
```

```
Aidan's birthday is on September 27
```

In [12]:
```
# Another solution: Create one big string with concatenation (+), then print it
# Note that concatenating int to str with "+" requires type conversion
textToPrint = name + "'s" + " birthday is on " + month + " "+ str(day)
print(textToPrint)
```

```
Aidan's birthday is on September 27
```

In [13]:
```
# Yet another solution: use the .format function
textToPrint = "{}'s birthday is on {} {}".format(name, month, day)
print(textToPrint)
```

```
Aidan's birthday is on September 27
```

### Why do we care about printing things?

Printing to the console can be useful for:

- Printing the status/diagnostics of a program
- Printing values of variables for debugging

---

## Next Time

Now that we are familiar with basic python syntax we can start writing functions: blocks of code that perform distinct, repeatable tasks.