Getting Started with Python

A workshop by Clara Turp, for McGill University Libraries

This workshop was built using this: https://librarycarpentry.org/lc-python-intro/

Please leave feedback:
https://forms.office.com/Pages/ResponsePage.aspx?id=cZYxzedSaEqvqfz4-J8J6jyp0g6_kyVDhNJTXJrpgkRUNkRUVU1IMzc5RElTUlZEWUtBM01aSUZLUi4u

Pre-Workshop: Downloading Anaconda

Installing Python Using Anaconda

https://librarycarpentry.org/lc-python-intro/setup.html

Python is great for general-purpose programming and is a popular language for scientific computing as well. Installing all of the packages required for this lessons individually can be a bit difficult, however, so we recommend the all-in-one installer Anaconda.

Regardless of how you choose to install it, please make sure you install Python version 3.x (e.g., Python 3.6 version). Also, please set up your Python environment at least a day in advance of the workshop. If you encounter problems with the installation procedure, ask your workshop organizers via e-mail for assistance so you are ready to go as soon as the workshop begins.

Windows - Video tutorial

1. Open anaconda.com/download with your web browser.
2. Download the Python 3 installer for Windows.
3. Double-click the executable and install Python 3 using *MOST* of the default settings. The only exception is to check the **Make Anaconda the default Python** option.

macOS - Video tutorial

1. Open anaconda.com/download with your web browser.
2. Download the Python 3 installer for macOS.
3. I recommend choosing the Graphical Installer
4. Install Python 3 using all of the defaults for installation.

Workshop 1

NOTES: Build a script that goes through all files in a directory and read lines. Extract the files with words. Or count the words? Build the script slowly during the two workshops and finish with a demo. Include the libraries imported earlier??

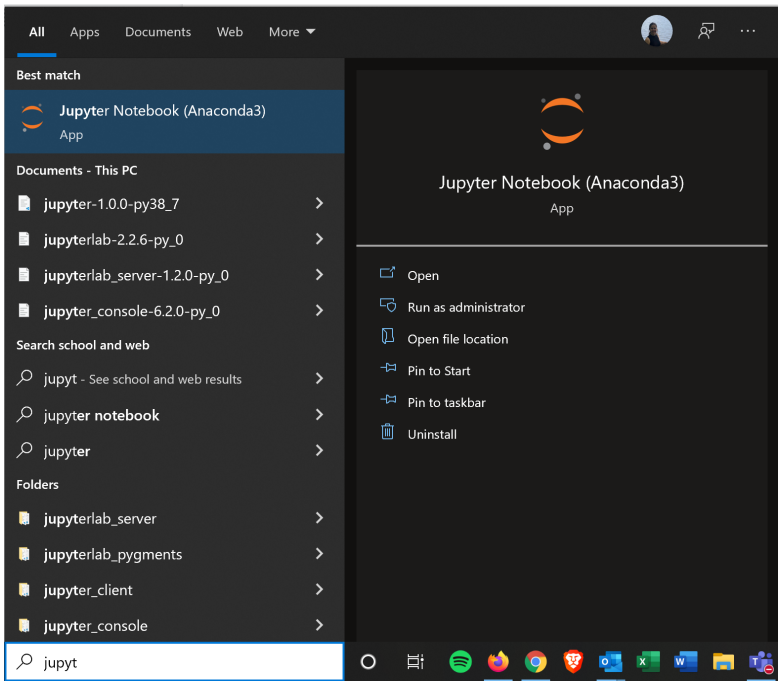
Getting Started

https://librarycarpentry.org/lc-python-intro/01-getting-started/index.html

Use Jupyter

- The Anaconda package manager is an automated way to install Jupyter Notebooks..
    - See the setup instructions for Anaconda installation instructions.

Once you have installed Python, you can search your computer for the Jupyter app.

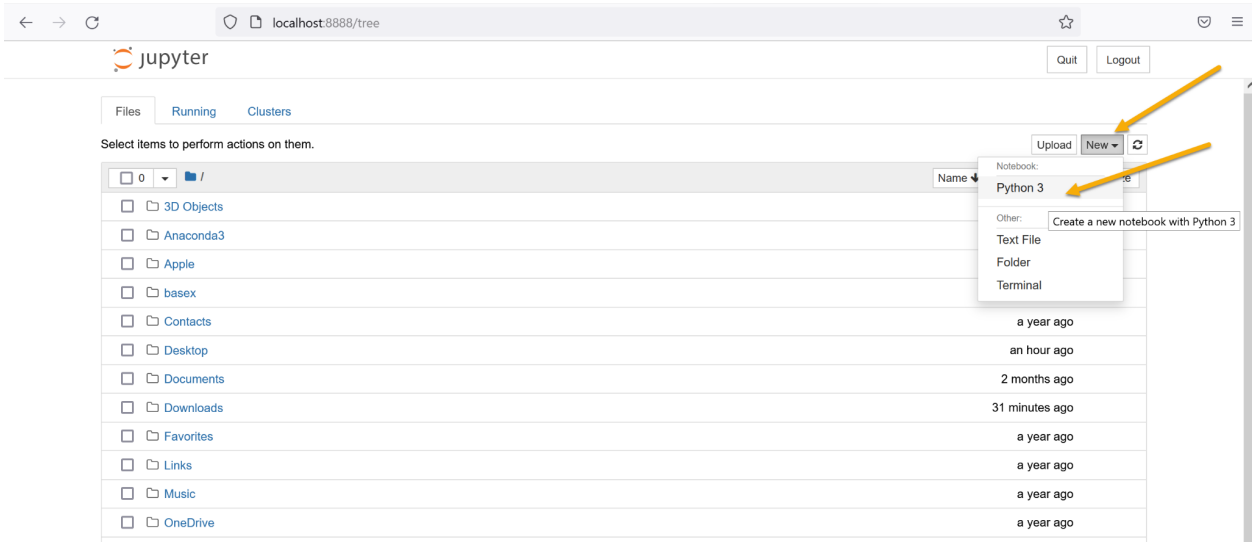


- This will start Jupyter Notebooks in your web browser. You will also see the application run in a terminal.

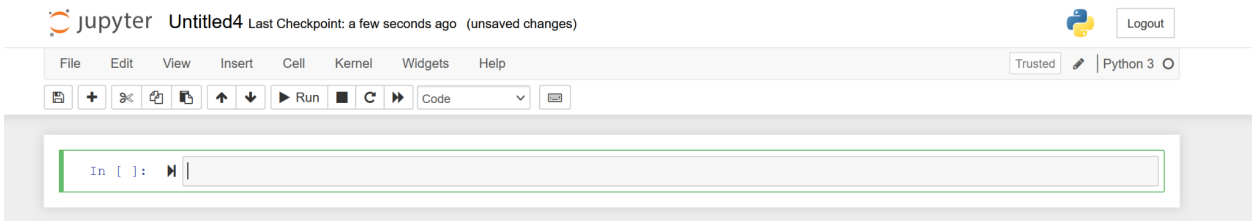
The advantages of Jupyter is that you can create a notebook with code that you can run and save the results of your code directly under it. You can also add notes.

When I run scripts, I prefer using Visual Studio Code, which is also part of the Anaconda Package. I, then, run scripts using the Anaconda Prompt. I will show you later, however, Jupyter Notebooks is a better learning environment.

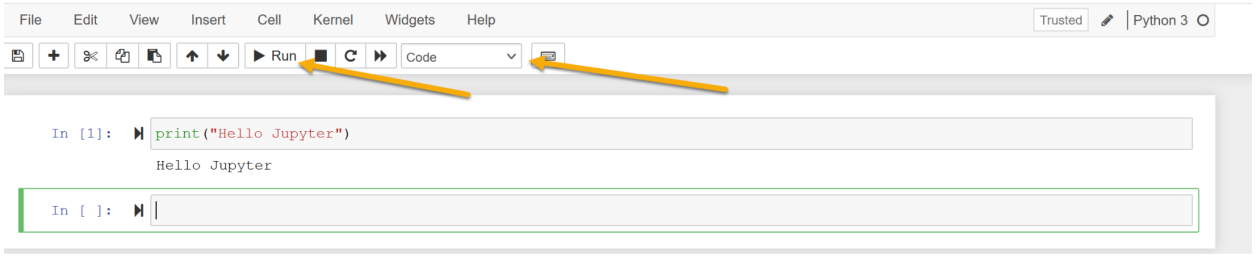
In Jupyter, click on **New** and then on **Python 3 Notebook**



A new notebook will be created in a new tab.

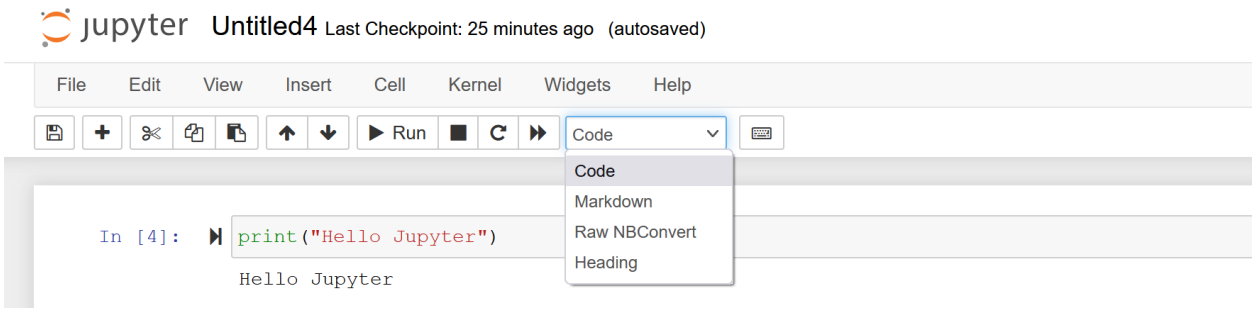


Try to type some Python code in the top box, next to the arrow, such as **print("Hello Jupyter")**. You can then run your code by clicking on the **Run button**, or with a keyboard shortcut **Shift-Enter.** You will then see the code you wrote and the result. A new cell will appear underneath. If you made a mistake, you can still edit the cell above.



You can also add new cells by clicking on the **+** sign, which is to insert a cell below. You can also select Insert and choose whether you want to insert a cell above or below your current cell.

The great part of Jupyter Notebooks is that it combines different types of cells, you can add **code** cells or **Markdown** cells. This means you can include all your workshop notes directly with the code.

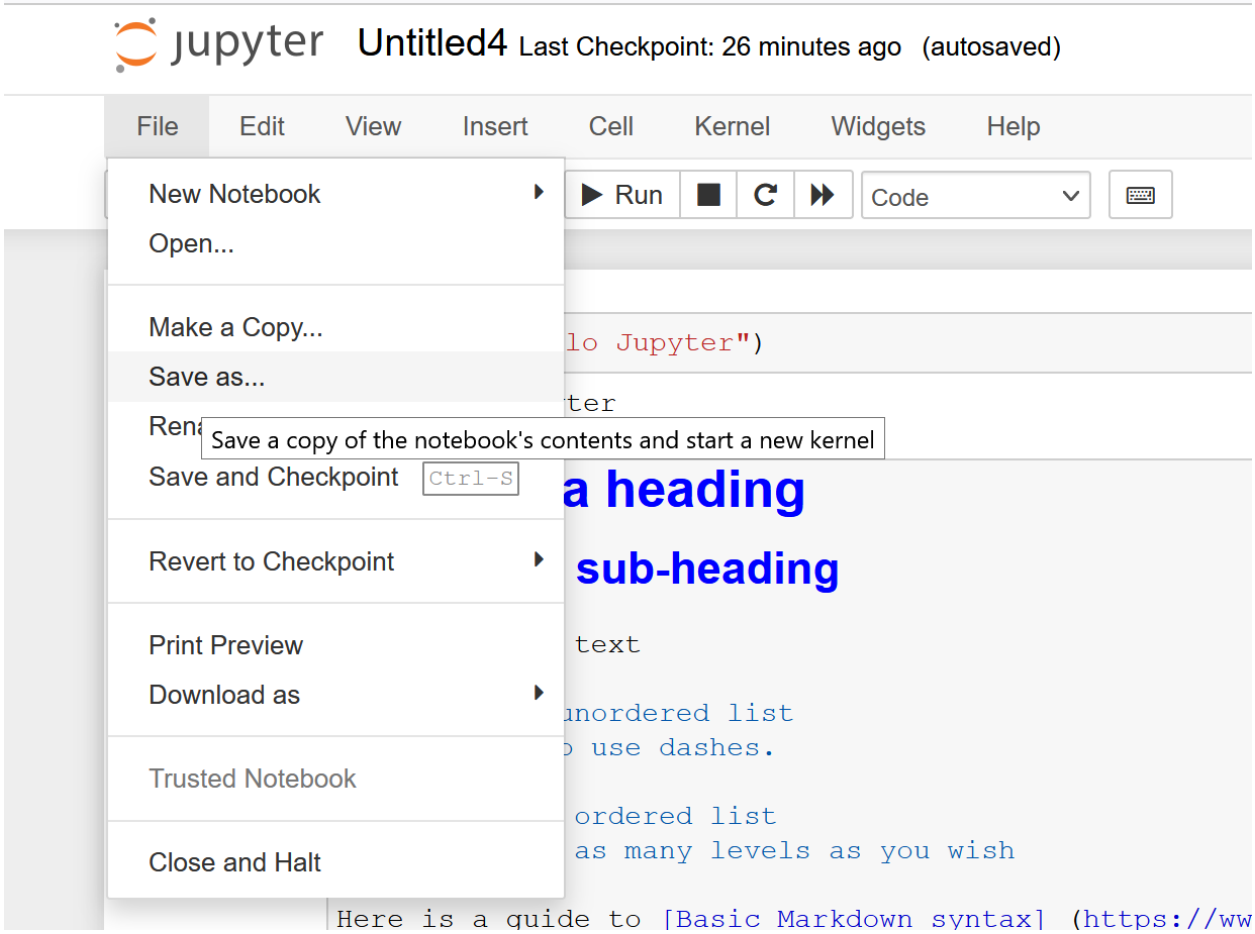


Here are some basics of Markdown language.

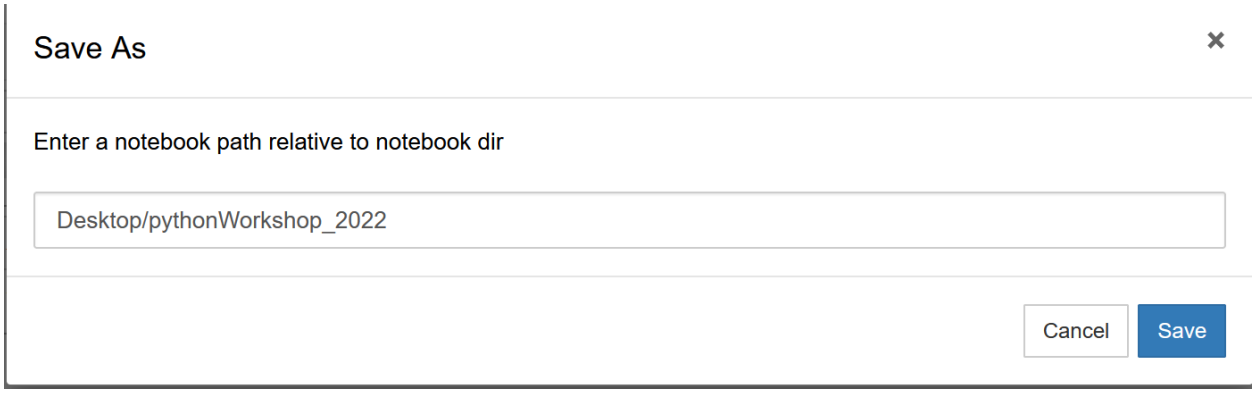
```
# This is a heading

## this is a sub-heading

this is plain text

* This is an unordered list
- You can also use dashes.

1. This is an ordered list
    1.1. With as many levels as you wish

Here is a guide to [Basic Markdown syntax] (https://www.markdownguide.org/basic-syntax/#overview)
```

Saving the code

To save the code, press ‘file’ and then ‘save as’. This saves your notebook (using the ipynb extension). This saves the notebook at your current directory in the Jupyter home page.



If I wanted to save it on my **Desktop**, I would have to type Desktop/PythonWorkshop_2022. This is called a relative path.



Use comments to add documentation to programs.

You can add comments to document your script, this works in Jupyter, but also wherever you are writing Python. Usually comments are used to describe what the script is doing. General documentation that might help your future-you.

- For one-line comments, use hash (#). The commented line will be ignored when you run a script.
- For multi-line comments, use three quotation marks (“””).
- Use comments to debug

```
#This is a comment

"""
This is a long comment

Usually when you have a multi-line comment

"""
```

Variables and Assignment

https://librarycarpentry.org/lc-python-intro/02-variables/index.html

Use variables to store values, and some data types.

- Variables are names for values.
- In Python the = symbol assigns the value on the right to the name on the left.
- The variable is created when a value is assigned to it.

Some data types:

- Integers (numbers) - don't use quotation marks.
- Strings (words) - use quotation marks
- If we type a number between quotation marks, such as Title = "1984" , what is it?

- Here, Python assigns an age to a variable age and a name in quotation marks to a variable first_name.
- Can you see the different colors for the different data types?

In [ ]:
```
age = 35              #Integers (numbers)
first_name = "Clara" #Strings (words), use quotation marks.
title = "1984"        #What would this be?
```

- Now if we called a variable, it would represent the value stored in it. So print(age) will print out the value assigned to the variable age, which is 35. To call the variable, you type the variable name without any quotation marks.

In [1]:
```
age = 35              #Integers (numbers)
first_name = "Clara" #Strings (words), use quotation marks.
title = "1984"        #What would this be?

print(age)
```

```
35
```

- Variable names:
  - Should have a meaning
  - cannot start with a digit
  - cannot contain spaces, quotation marks, or other punctuation

    - Use underscore or CamelCase (typically used to separate words in long variable names)
        - Underscores at the start like __alistairs_real_age have a special meaning so don't use it.

Python is case-sensitive.

- Python thinks that upper- and lower-case letters are different, so Name and name are different variables.
- There are conventions for using upper-case letters at the start of variable names so we will use lower-case letters for now.

Use meaningful variable names.

- Python doesn't care what you call variables as long as they obey the rules (alphanumeric characters and the underscore).

```
flabadab = 42
ewr_422_yY = 'Ahmed'
```

- Use meaningful variable names to help other people understand what the program does.
- The most important "other person" is your future self.

Use print to display values.

- As we saw, Python has a built-in function called print that prints things as text.
- Call the function (i.e., tell Python to run it) by using its name.
- Provide values to the function (i.e., the things to print) in parentheses.
- To add a string to the printout, wrap the string in quotations.
- The values passed to the function are called 'arguments'
    - So Print is a function (we will explore this later)
    - Print expects at least one argument
    - The arguments can be a variable, an integer, a string.
    - The arguments will be printed to the console.

If we want to print Clara is 35 years old, we could do: print("Clara is 35 years old")

Or we could use the variables we defined earlier. Please note that Python remembers the variables, even if they were created earlier. Jupyter reads all the cells, not just the current one.

```
In [2]: # Print is a built-in function that prints the function's argument as text.
        print("Clara is 35 years old")
        print(first_name, "is", age, "years old")

        Clara is 35 years old
        Clara is 35 years old
```

Here is what happens once we run the script:

- print automatically puts a single space between items to separate them; items (arguments) are separated by a comma.
- And wraps around to a new line at the end.

Variables must be created before they are used.

- If a variable doesn't exist yet, or if the name has been misspelled, Python reports an error.
- Once created, the variable can be used anywhere in your script.

If we add a print function that calls the variable eye_color without first defining the variable, we get an error:

```
In [4]: print(eye_color)

        ---------------------------------------------------------------------
        NameError                                 Traceback (most recent call last)
        <ipython-input-4-28e7666e71cd> in <module>
        ----> 1 print(eye_color)

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

- The last line of an error message is usually the most informative.

Variables can be used in calculations.

- We can use variables in calculations just as if they were values.
    - Remember, we assigned 35 to age.
- There are multiple ways to add 3 to Clara's age.
    - The first one (age + 3) prints out 38, however that value will not be saved.

- The second one (age = age + 3) replaces the value of age with 38.
- The third one (age_plus_three = age + 3) assigns the value 38 to a new variable. You can see here that it prints 41, because we changed the value of age.

The three different options for calculating age + 3

```
In [2]: age = 35

        print(age + 3)
```

```
38
```

```
In [3]: age = 35
        age = age + 3

        print(age)
```

```
38
```

```
In [5]: age = 35
        age_plus_three = age + 3

        print(age_plus_three, age)
```

```
38 35
```

Use an index to get a single character from a string.

- The characters (individual letters, numbers, and so on) in a string are ordered. For example, the string ‘AB’ is not the same as ‘BA’. Because of this ordering, we can treat the string as a list of characters.
- Each position in the string (first, second, etc.) is given a number. This number is called an index or sometimes a subscript.
- Indices are numbered from 0 rather than 1.
- Use the position’s index in square brackets to get the character at that position.

```
element = "helium"
print(element[0])
```

```
h
```

- What happens if we change the “0” with another number? What letter would be returned if we used 1?
- What happens if we use -1? Using a negative number starts at the end of the word (starting with 1). -1 will return the last letter of the word.

```
element = "helium"
print(element[0])
```

```
h
```

```
print(element[1])
print(element[-1])
```

Use a slice to get a substring.

- A part of a string is called a substring. A substring can be as short as a single character.
- A slice is a part of a string.
- We take a slice by using [start:stop], where start is replaced with the index of the first element we want and stop is replaced with the index of the element just after the last element we want.
- Mathematically, you might say that a slice selects [start:stop].
- The difference between stop and start is the slice's length.
- Taking a slice does not change the contents of the original string. Instead, the slice is a copy of part of the original string.

```
element = "helium"
print(element[0:3])
```

```
hel
```

Data types

https://librarycarpentry.org/lc-python-intro/03-types-conversion/index.html

Every value has a type.

- Every value in a program has a specific type.
- Integer (int): whole numbers like 3 or -512.
- Floating point number (float): fractional numbers like 3.14159 or -2.5.
- Whole numbers may also be stored as floats, e.g. 1.0.
- Character string (usually called “string”, str): text.
    - Written in either single quotes or double quotes (as long as they match).
    - The quotation marks aren’t printed using print(), but may appear when viewing a value in the Jupyter Notebook or other Python interpreter.
- Booleans: Their value is either True or False (with an Uppercase letter)
    - Some methods or built-in functions return a boolean.

Use the built-in function type to find the type of a value.

- Use the built-in function type to find out what type a value has.
- This works on variables as well.
    - But remember: the *value* has the type — the *variable* is just a label.
    - When you change the value of a variable to a new data type, the results of print(type(your_variable)) will change accordingly.

```
print(type(52))
print(type(True))
print(type(3.7964))
print(type("hello world"))
```

```
<class 'int'>
<class 'bool'>
<class 'float'>
<class 'str'>
```

Types control what operations (or methods) can be performed on a given value.

- A value’s type determines what the program can do to it.
- You can use - with numbers, but not strings.

```
In [2]: print("hello" - "o")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-c2c419fea538> in <module>
----> 1 print("hello" - "o")

TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

- You can use on + and * on strings:
  - “Adding” character strings concatenates them.
  - Multiplying a character string by an integer *N* creates a new string that consists of that character string repeated *N* times.
    - Since multiplication is repeated addition.

```
print("Clara" + " " + "Turp")
print("Clara" * 10)
```

```
Clara Turp
ClaraClaraClaraClaraClaraClaraClaraClaraClaraClara
```

Must convert numbers to strings or vice versa when operating on them.

- Cannot add numbers and strings.
- Not allowed because it’s ambiguous: should 1 + '2' be 3 or '12'?
- Some types can be converted to other types by using the type name as a function.
- str(1) converts the integer 1 into the string “1”
- int(“10”) converts the string into an integer.
- int(1.10) converts the float to an integer.

```
print(1 + int("2"))
```

```
3
```

```
print(str(1) + "2")
```

```
12
```

```
print(int(1.10))
```

```
1
```

Prediction exercise

What do you think will happen if you convert 1.889 to an integer?

Can mix integers and floats freely in operations.

- Integers and floating-point numbers can be mixed in arithmetic.
    - Python automatically converts integers to floats as needed.

```
print("half is", 1 / 2.0)
print("three squared is", 3.0 ** 2)
```

```
half is 0.5
three squared is 9.0
```

Some built-in functions and methods

https://librarycarpentry.org/lc-python-intro/04-built-in/index.html

https://www.w3schools.com/python/python_ref_string.asp

Built-in functions

Functions are central to programming. You can write functions or use some of the built-in functions provided. Functions allow you to take an action (algorithm) on an element you link it to. Functions are powerful because you can use them repeatedly.

A function may take zero or more arguments.

- We have seen some functions already — now let's take a closer look.
- An *argument* is a value passed into a function.
- Any arguments you want to pass into a function must go into the ()
    - print("I am an argument and must go here.")
- You must always use parentheses, because this is how Python knows you are calling a function.
    - You leave them empty if you don't want or need to pass any arguments in.
- print takes zero or more.

Every function returns something.

- Every function call produces some result.
- If the function doesn't have a useful result to return, it usually returns the special value None.

Use the built-in function len to find the length of a string.

- Len() is a built-in function. It returns the length of an objectL
    - How many characters are in a string
    - How many items are in a list
    - Length function doesn't work on numbers.
- For now, let's worry about the length of a string.
- The function takes one argument in the parentheses.

```
element = "Helium"
print(len(element))
```

```
6
```

- Nested functions are evaluated from the inside out, just like in mathematics.
    - Finding out the length of the variable element, before you print.

Exercise

- Try to return the last letter of the string using len(). Note: If Python starts counting from zero, and len returns the number of characters in a string, what index expression will get the last character in the string name?
    - name[len(name) - 1]

Use string methods - upper() and lower() - to transform a string

- A method is similar to functions, but they are tied to an object (i.e. a string)
    - Use object_name.method_name to call methods.
- These methods take a string and return the same string with all uppercase or lowercase.

```
element = "helium"
upper_element = element.upper()
print(upper_element)
```

```
HELIUM
```

Use the built-in function max and min to find the largest / smallest value.

- Both work on character strings as well as numbers.
    - “Larger” and “smaller” use (0-9, A-Z, a-z) to compare letters.
    - This means that:
        - 'a' is smaller than 'b'
        - 'A' is smaller than 'a'
        - '0' is smaller than 'a'

Code predictions:

What will these lines print to the console:

```
print(max(1,2,5,78,56))
print(min(0.5, 5.75, -3, 587))
print(max("a", "total", "g"))
print(min("science", "total", "hebrew", "10"))
```

- What happens if we combine strings and numbers in the list?
    - It returns an error. Min and max compare two values and it can't compare strings and integers.

Use the built-in function round to round off numbers

- Reminder: What are floating-point numbers?
- The round() function takes two arguments. The second argument is the specified number of decimals.
- The default number of decimals is 0, meaning that the function will return the nearest integer.

```
print(round(3.798, 2))
print(round(3.798))
```

```
3.8
4
```

Use string method replace() to find and replace an element in a string

- The replace() method takes a string, finds a specific element and replaces it by another element.
- string.replace(oldvalue, newvalue, count)
    - Count is optional.

```
favorite_fruit = "James' favorite fruit is banana"
print(favorite_fruit)
```

```
James' favorite fruit is banana
```

```
new_string = favorite_fruit.replace("banana", "orange")
print(new_string)
```

```
James' favorite fruit is orange
```

- By adding a count, only *n* occurrences will change. If you write “1”, only the first iteration will change. The default is to change all occurrences.
- First, replace all spaces with a semicolon and a space. You will see all spaces changed.

```
new_string = favorite_fruit.replace(" ", "; ")
print(new_string)
```

```
James'; favorite; fruit; is; banana
```

- Now if we add a count, only the first space will be replaced.

```
new_string = favorite_fruit.replace(" ", "; ", 1)
print(new_string)
```

```
James'; favorite fruit is banana
```

Functions may only work for certain (combinations of) arguments.

- max and min must be given at least one argument.
- And they must be given things that can meaningfully be compared.
- What might be the problem with this line:

```
print(max("science", "total", 10, "hebrew"))
```

```
---------------------------------------------------------------
-----------------
TypeError                                 Traceback (most
recent call last)
<ipython-input-9-111a46402a3e> in <module>
----> 1 print(max("science", "total", 10, "hebrew"))

TypeError: '>' not supported between instances of 'int' an
d 'str'
```

Use the built-in function help to get help for a function.

- Every built-in function has online documentation.
- Gives you information about the function and the expected arguments.

```
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.
```

Exercises:

Nesting exercise:

- What will be printed out to the console if you type this code

```
color = "Orange"
fruit = "Tangerine"

print(max(len(color), len(fruit)))
```

Lower exercise:

- Create a variable “python” and assign it the phrase “Hello World, I AM learning Python.”
- Use the lower method to transform the phrase to only lowercase.
- Print the result to the console.

    python = “Hello World, I AM learning Python”
    lower_python = python.lower()
    print(lower_python)

Replace exercises:

- Create a sentence that contains a list of ingredients required to make pizza.
- Use commas between each ingredient.
- Replace all commas with semicolons.

Some Slicing Exercises

1. What does thing[low:high] do?
2. What does thing[low:] (without a value after the colon) do?
3. What does thing[:high] (without a value before the colon) do?
4. What does thing[:] (just a colon) do?
5. What does thing[number:negative-number] do?

Workshop 2

Activity

We would like to create a script that reads files from a folder and returns ”Match found” with the filename when a match is found for a specific word.
We will build this script slowly during this workshop. The first thing is to add files to a new folder you created in the folder for this workshop.

Let’s create a variable called numberOfMatches and assign it the value 0.

```
numberOfMatches = 0
```

Lists

https://librarycarpentry.org/lc-python-intro/11-lists/index.html

A list stores many values in a single structure.

- Scenario: You want to keep track of many temperatures.
- Doing calculations with a hundred variables called temperature_001, temperature_002, etc., would be at least as slow as doing them by hand.
- You can create a list by assigning brackets ([]) to a list name. You can create it with values or without.
    - Use [] on its own to represent a list that doesn’t contain any values.
        - “The zero of lists.”
- Use a *list* to store many values together.
    - Contained within square brackets [...].
    - Values separated by commas ,.
- Use len to find out how many values are in a list.

```
temperatures = []
print(temperatures)
```

```
[]
```

```
temperatures = [17.3, 17.5, 17.7, 17.5, 17.6]
print(temperatures)
print("length of temperatures list", len(temperatures))
```

```
[17.3, 17.5, 17.7, 17.5, 17.6]
length of temperatures list 5
```

Use an item's index to fetch it from a list.

- Just like strings.
- The count starts at 0.

```
print(temperatures[0])
print(temperatures[4])
```

```
17.3
17.6
```

- What happens if you use an index that isn't linked to an item? For example: print(temperatures[5])

```
print(temperatures[5])
```

```
---------------------------------------------------------------------
IndexError                                Traceback (most
recent call last)
<ipython-input-4-ee631108fda6> in <module>
----> 1 print(temperatures[5])

IndexError: list index out of range
```

Lists' values can be replaced by assigning new values to them.

- You can assign a value to lists similarly to the way you assign values to a variable, however you want to use the lists' name and index.
- This works to replace a value in a list. You can't use that to add values to an empty list.

```
temperatures = [17.3, 17.5, 17.7, 17.5, 17.6]
print(temperatures)

temperatures[0] = 14.7
print("list is now: ", temperatures)
```

```
[17.3, 17.5, 17.7, 17.5, 17.6]
list is now:  [14.7, 17.5, 17.7, 17.5, 17.6]
```

Appending items to a list lengthens it.

- Use list_name.append to add items to the end of a list.

```
temperatures = [17.3, 17.5, 17.7, 17.5, 17.6]

temperatures.append(19.7)
print(temperatures)
```

```
[17.3, 17.5, 17.7, 17.5, 17.6, 19.7]
```

- append is a *method* of lists.
    - Like a function, but tied to a particular object.
- We will meet other methods of lists as we go along.
    - Use help(list) for a preview.

Use del to remove items from a list entirely.

- del list_name[index] removes an item from a list and shortens the list.

```
primes = [2, 3, 5, 7, 9]
print("primes before removing last item:", primes)
del primes[4]
print("primes after removing last irem:", primes)
```

```
primes before removing last item: [2, 3, 5, 7, 9]
primes after removing last irem: [2, 3, 5, 7]
```

Lists may contain values of different types.

- A single list may contain numbers, strings, and anything else.
- A list can even contain other lists.

```
goals = [1, "Create Lists", 2.0, "Extract items from lists", 3, "Modify lists"]
values = [12, [3, 4, 5], 87, "Hello World"]
```

Conditionals

https://librarycarpentry.org/lc-python-intro/17-conditionals/index.html

Use if statements to control whether or not a block of code is executed.

- An if statement (more properly called a *conditional* statement) controls whether some block of code is executed or not.
- The idea is:
  - If x then:
    - Do a
  - If y then:
    - Do b
  - A is only executed if the conditions of x are met.
- Structure is similar to a for statement:
  - If variable conditions :
    - Block of code
    - This block of code is indented (usually by 4 spaces)
  - The conditions often use <, >, ==, !=

```
#mass = 3.54
mass = 2.97
if mass > 3.0:
    print(mass, "is larger")
if mass < 3.0:
    print(mass, "is smaller")
```

```
2.97 is smaller
```

```
mass = 3.54
if mass != 3.0:
    print(mass, "is not 3.0")
if mass == 3.0:
    print(mass, "is 3.0")
```

```
3.54 is not 3.0
```

- Indentation is always meaningful in Python. Expected indented block missing or unexpected indent will return an error.

```
firstName = "John"
 lastName = "Smith"
```

```
  File "<ipython-input-8-da09491f9801>", line 2
    lastName = "Smith"
    ^
IndentationError: unexpected indent
```

Activity:
Create an if statement that evaluates whether the numberOfMatches variable is greater than 0, meaning that there would be at least one match.

```
if numberOfMatches > 0:
    print("match found")
```

Use else to execute a block of code when an if condition is *not* true.

- else can be used following an if.
- Allows us to specify an alternative to execute when the if *branch* isn't taken.
- The structure is:
    - If variable condition:
        - Block of code (indented)
    - Else:
        - Block of code (indented)

```
mass = 3.78
if mass > 3.0:
    print(mass, "is larger")
else:
    print(mass, "is smaller")
```

```
3.78 is larger
```

Use elif to specify additional tests.

- May want to provide several alternative choices, each with its own test.
- Use elif (short for "else if") and a condition to specify these.
- Always associated with an if.

- Once the condition is met, the code stops. That is why 9.7 isn't printed twice, once with is HUGE and once with is larger.
- Must come before the else (which is the "catch all").

```
mass = 4.9
if mass > 9.0:
    print(mass, "is huge")
elif mass > 3.0:
    print(mass, "is larger")
else:
    print(mass, "is smaller")
```

```
4.9 is larger
```

Conditions are tested once, in order.

- Python steps through the branches of the conditional in order, testing each in turn.
- So ordering matters.
- How can we fix this error?

```
grade = 85

if grade >= 70:
    print("Grade is C")
elif grade >= 80:
    print("Grade is B")
elif grade >= 90:
    print("Grade is A")
```

```
Grade is C
```

For Loops

https://librarycarpentry.org/lc-python-intro/12-for-loops/index.html

A *for loop* executes commands once for each value in a collection.

- Doing calculations on the values in a list one by one is as painful as working with temperature_001, temperature_002, etc.
- Let's say you want to print out every temperature recorded. You could do it this way:

```
temperature_01 = 17.5
temperature_02 = 15.9
temperature_03 = 19.4
temperature_04 = 21.1

print(temperature_01)
print(temperature_02)
print(temperature_03)
print(temperature_04)
```

- You can also use a for loop to do this systematically.
- A *for loop* tells Python to execute some statements once for each value in a list, a character string, or some other collection.
- "for each thing in this group, do these operations"

```
for temperature in [17.5, 15.9, 19.4, 21.2]:
    print(temperature)
```

- You can also do this with a list assigned to a variable:

```
temperatures = [17.5, 15.9, 19.4, 21.2]

for current_temperature in temperatures:
    print(current_temperature)
```

```
17.5
15.9
19.4
21.2
```

- The output of all these should be the same.:

The first line of the for loop must end with a colon, and the body must be indented.

- Just like conditionals, the indentation is essential.
- The colon at the end of the first line signals the start of a *block* of statements.
- Python uses indentation rather than {} or begin/end to show *nesting*.
    - Any consistent indentation is legal, but almost everyone uses four spaces.

```
temperatures = [17.5, 15.9, 19.4, 21.2]

for current_temperature in temperatures:
print(current_temperature)
```

```
  File "<ipython-input-7-424393157208>", line 4
    print(current_temperature)
    ^
IndentationError: expected an indented block
```

- This error can be fixed by removing the extra spaces at the beginning of the second line.

A for loop is made up of a collection, a loop variable, and a body.

```
for number in [2, 3, 5]:
    print(number)
```

- The collection, [2, 3, 5], is what the loop is being run on.
- The body, print(number), specifies what to do for each value in the collection.
- The loop variable, number, is what changes for each *iteration* of the loop.
    - The “current thing”.

```
for number in [2,3,5]: # number is the loop variable. [2,3,5] is the collection, what the loop is running on.
    print(number) # The body specifies what to do for each value in the collection.
```

Loop variable names follow the normal variable name conventions.

- Loop variables will:
    - Be created on demand during the course of each loop.
    - Persist after the loop finishes.
        - Use a new variable name to avoid overwriting a data collection you need to keep for later
    - Often be used in the course of the loop

- So give them a meaningful name you'll understand as the body code in your loop grows.

Example:

```
for single_letter in ['A', 'B', 'C', 'D']:

instead of

for asdf in ['A', 'B', 'C', 'D']:
for kitten in [2, 3, 5]:
```

The body of a loop can contain many statements.

- But no loop should be more than a few lines long.
- ** is exponential

```
primes = [2,3,5]
for number in primes:
    squared = number ** 2
    cubed = number ** 3
    print(number, squared, cubed)
```

```
2 4 8
3 9 27
5 25 125
```

Use range to iterate over a sequence of numbers.

- The built-in function range produces a sequence of numbers.
    - *Not* a list: the numbers are produced on demand to make looping over large ranges more efficient.
- range(N) is the numbers 0..N-1

```
for number in range(0,3):
    print(number)
```

```
0
1
2
```

Or use range to repeat an action an arbitrary number of times.

- You don't actually have to use the iterable variable's value.
- Use this structure to simply repeat an action some number of times.
    - That number of times goes into the range function.

```
for number in range(5):
    print("Again!")
```

```
Again!
Again!
Again!
Again!
Again!
```

The Accumulator pattern turns many values into one.

- A common pattern in programs is to:
    1. Initialize an *accumulator* variable to zero, the empty string, or the empty list.
    2. Update the variable with values from a collection.

```
# sum of the first 10 integers.

total = 0 # Accumulator variable
for number in range(10):
    total = total + (number + 1) # Remember, the range will start at 0.

print(total)
```

```
55
```

- Read total = total + (number + 1) as:
    - Add 1 to the current value of the loop variable number.
    - Add that to the current value of the accumulator variable total.
    - Assign that to total, replacing the current value.
- We have to add number + 1 because range produces 0..9, not 1..10.

Conditionals are often used inside loops.

- Not much point using a conditional when we know the value (as above).
- But useful when we have a collection to process.

```
masses = [3.54, 2.07, 9.22, 1.86, 1.71]
for m in masses:
    if m > 3.0:
        print(m, "is larger")
```

```
3.54 is larger
9.22 is larger
```

Activity

- You can use loops and conditionals to “evolve” the values of variables. In the activity notebook, create a for loop that reads all lines in the reader variable. Then make the line lower case. With each line, create a first if statement that looks for the word sleep in line.
- Then If the word is found, add 1 to the variable numberOfMatches, using the accumulator pattern.

```
for line in reader:
    line = line.lower()
    if "sleep" in line:
        numberOfMatches = numberOfMatches + 1
```

Writing Functions

https://librarycarpentry.org/lc-python-intro/14-writing-functions/index.html

Break programs down into functions to make them easier to understand.

- Human beings can only keep a few items in working memory at a time.
- Understand larger/more complicated ideas by understanding and combining pieces.
    - Components in a machine.
    - Lemmas when proving theorems.
- Functions serve the same purpose in programs.
    - *Encapsulate* complexity so that we can treat it as a single “thing”.
- Also enables *re-use*.
    - Write one time, use many times.

Define a function using def with a name, parameters, and a block of code.

- Begin the definition of a new function with def.
- Followed by the name of the function.
    - Must obey the same rules as variable names.
- Then *parameters* in parentheses.
    - Empty parentheses if the function doesn’t take any inputs.
    - We will discuss this in detail in a moment.
- Then a colon.
- Then an indented block of code.

```
def print_greeting():
    print('Hello!')
```

Defining a function does not run it.

- Defining a function does not run it.
    - Like assigning a value to a variable.
- Must call the function to execute the code it contains.
- The commands for the function are read and stored after the def block, but not actually executed until the function is called later on.
    - Imagine getting a recipe card and keeping it in your kitchen. You can cook it anytime, but you haven’t completed any of the steps until you start that cooking process.

- This means that Python won't complain about problems until you call the function. More specifically, just because the definition of a function runs without error doesn't mean that there won't be errors when it executes later.

```
def print_greeting():
    print('Hello!')

print_greeting()
```

```
Hello!
```

Arguments in call are matched to parameters in definition.

- Functions are most useful when they can operate on different data.
- Specify *parameters* when defining a function.
  - These become variables when the function is executed.
  - Are assigned the arguments in the call (i.e., the values passed to the function).

```
def print_date(year, month, day):
    joined = str(year) + "/" + str(month) + "/" + str(day)
    print(joined)

print_date(1871, 3, 19)
```

```
1871/3/19
```

Functions may return a result to their caller using return.

- Use return ... to give a value back to the caller.
- May occur anywhere in the function.
- But functions are easier to understand if return occurs:
  - At the start to handle special cases.
  - At the very end, with a final result.

```
def average(values):
    if len(values) == 0:
        return None

    return sum(values)/ len(values)

a = average([1,3,4])
print("average of values:", a)

av = averag([])
print("average of empty list:", av)
```

```
average of values: 2.6666666666666665

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-18-1d65eb42a7cc> in <module>
      8 print("average of values:", a)
      9
---> 10 av = averag([])
     11 print("average of empty list:", av)

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

- Remember: every function returns something
- A function that doesn't explicitly return a value automatically returns None.

Activity:

- Transform the activity loop and conditional statement into a function..
- This function should have three parameters: text, word, and accumulatorVar
- Return the accumulatorVar

```
def findMatch(text, word, accumulatorVar):
    for line in text:
        line = line.lower()
        if word in line:
            accumulatorVar = accumulatorVar + 1

    return accumulatorVar
```

Using libraries

https://librarycarpentry.org/lc-python-intro/06-libraries/index.html

Most of the power of a programming language is in its (software) libraries.

- A *(software) library* is a collection of files (called *modules*) that contains functions for use by other programs.
    - May also contain data values (e.g., numerical constants) and other things.
    - Library's contents are supposed to be related, but there's no way to enforce that.
- The Python standard library is an extensive suite of modules that comes with Python itself.
- Many additional libraries are available from PyPI (the Python Package Index).
- Libraries allow you to not reinvent the wheel.

Libraries and modules

A library is a collection of modules, but the terms are often used interchangeably, especially since many libraries only consist of a single module, so don't worry if you mix them.

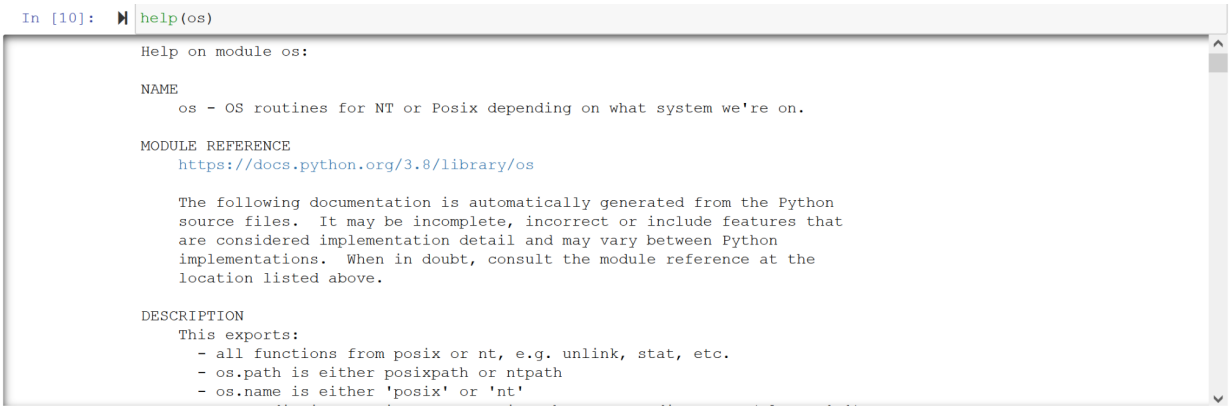
A program must import a library module before using it.

- Use import to load a library module into a program's memory.
- Then refer to things from the module as module_name.thing_name.
    - Python uses . to mean "part of".
- For the activity, we will import a library: os

```
# import a library
import os
```

Use help to learn about the contents of a library module.

- Works just like help for a function.



Use the library's name to use modules available in the library

```
currentDir = os.getcwd()
print(currentDir)
```

```
C:\Users\cturp1\Desktop\PythonWorkshop
```

Create an alias for a library module when importing it to shorten programs.

- Use import ... as ... to give a library a short *alias* while importing it.
- Then refer to items in the library using that shortened name.

```
import os as opSys

currentDir = opSys.getcwd() #cwd stands for Current Working Directory
print(currentDir)
```

```
C:\Users\cturp1\Desktop\PythonWorkshop
```

- Commonly used for libraries that are frequently used or have long names.
- But can make programs harder to understand, since readers must learn your program's aliases.

Activity

Let's walk through the rest of the script, using the function we already wrote, the variable with created, and the if statement.

```
# import a library
import os

## function:

def findMatch(text, word, accumulatorVar):
    for line in text:
        line = line.lower()
        if word in line:
            accumulatorVar = accumulatorVar + 1

    return accumulatorVar

## Script:

currentDir = os.getcwd()
filesDir = currentDir + "\\Grimm_fairyTales\\"

#Python method listdir() returns a list of files / folders in the directory given.
for filename in os.listdir(filesDir):

    numberOfMatches = 0
    currentFile = filesDir + filename

    if filename != ".ipynb_checkpoints":
        myFile = open(currentFile, "r", encoding = "utf-8", errors = "ignore")
        reader = myFile.readlines()
        numberOfMatches = findMatch(reader, "sleep", numberOfMatches)

    if numberOfMatches > 0:
        print("match found", numberOfMatches, "times in:", filename)
```

Using GitHub to find libraries

- You can find many libraries that relate to a specific domain.
- Most of those libraries are available through GitHub
    - https://github.com/pandas-dev/pandas
    - https://requests.readthedocs.io/en/master/user/install/#install
- They can be downloaded and then you can use them like any library
    - Pip and conda are easy ways to download libraries