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

Getting Started

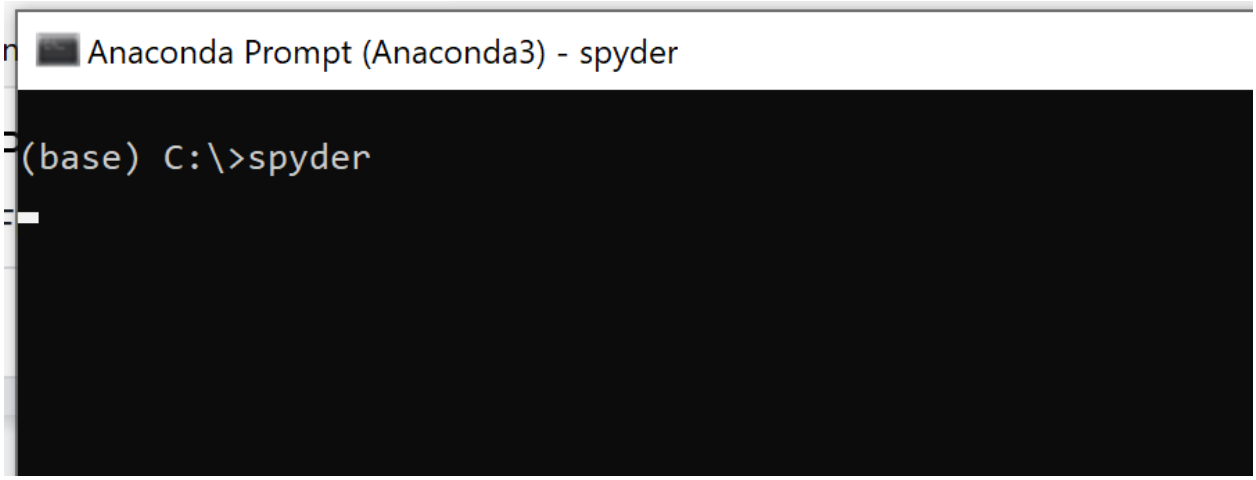
https://librarycarpentry.org/lc-python-intro/01-getting-started/index.html

Use the Spyder IDE for editing and running Python.

- The Anaconda package manager is an automated way to install the Spyder IDE.
    - See the setup instructions for Anaconda installation instructions.
- It also installs all the extra libraries it needs to run.

Once you have installed Python and the Spyder IDE requirements, open a shell and type: "Spyder".

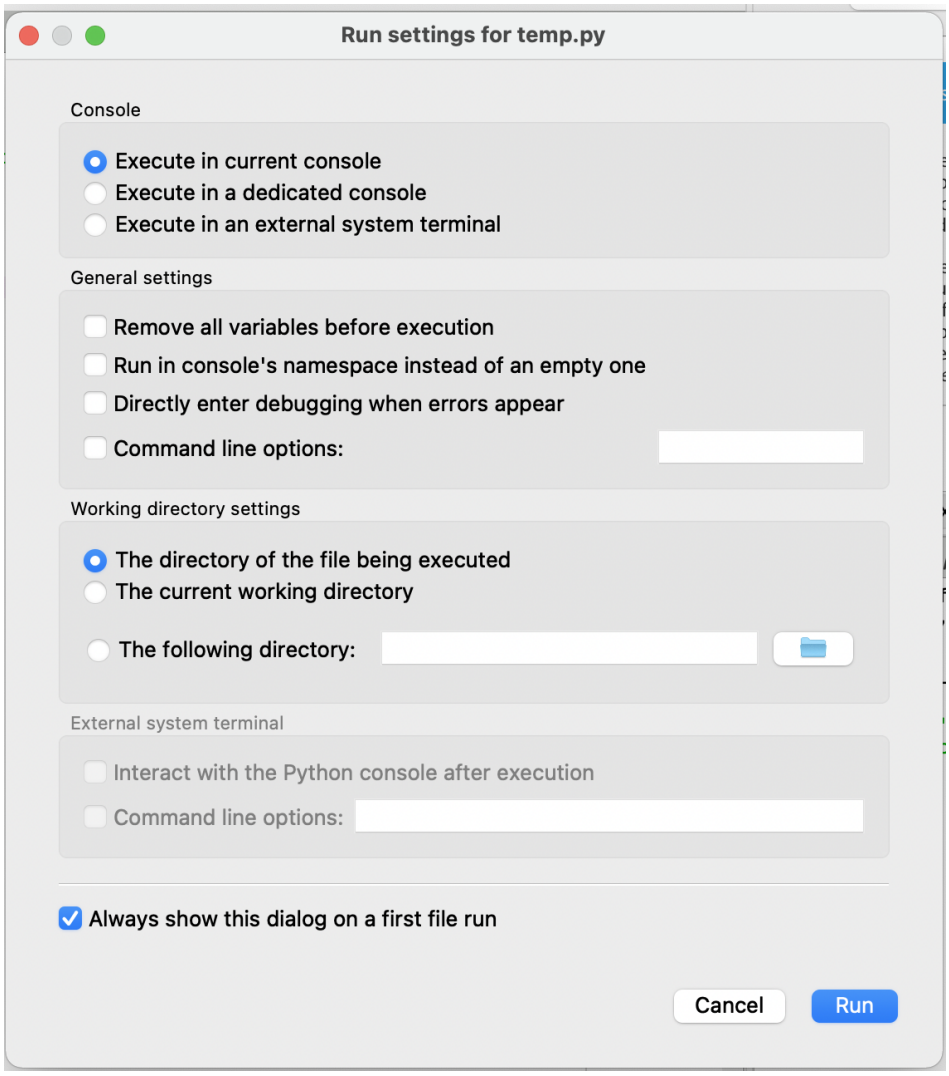
- On a Mac, use Spotlight (Command + Space Bar) to search for **Terminal** (terminal.app). Open Terminal and type "Spyder" to launch the application.
- On a PC, open Anaconda Prompt (type Anaconda in the search bar) . Open the terminal and type "Spyder" to launch the application.

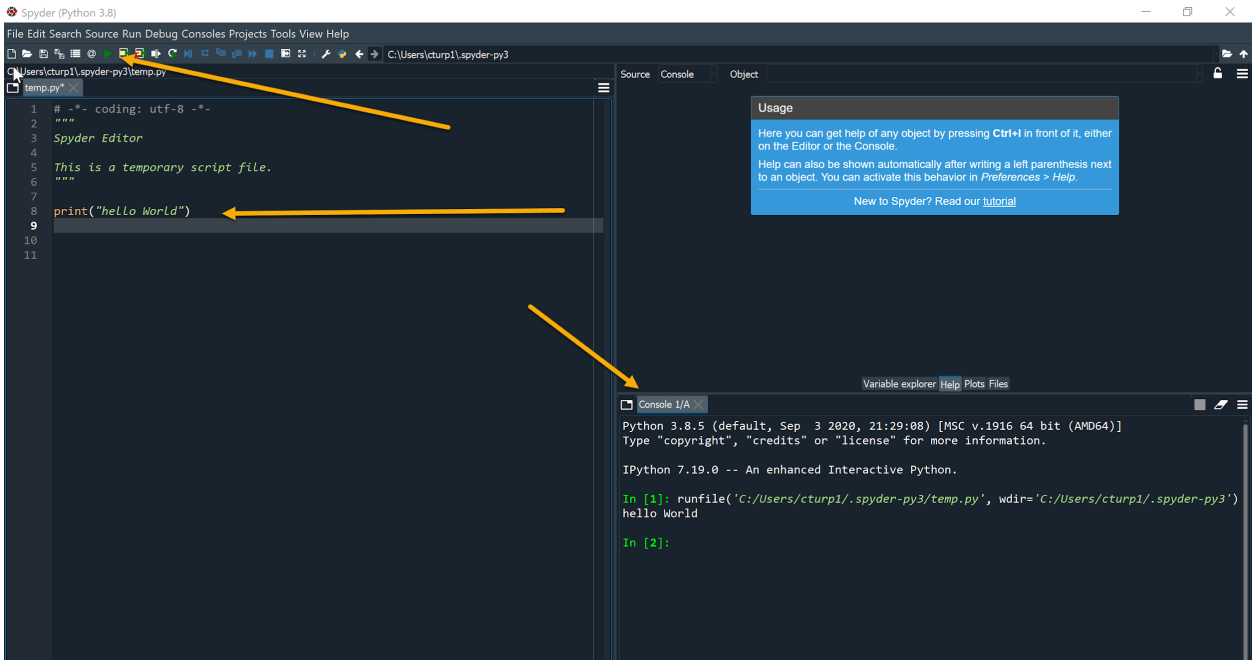


- This will start The Spyder IDE.
- This environment has several useful tools we can use, which you can see in different panels in the Spyder IDE. We will look into some of them.
- You can change the positions and sizes of these panels to your preference, as you get to know them.

Interacting with Python using Spyder

- On the left, filling half of the screen is the editor. Here you can write and edit code, which can then be saved in a file (usually with a .py extension). We can run the code we wrote here by pressing the green 'play' button on top or press F5 on your keyboard.
- On the bottom right, we find the IPython console. This is where we can talk directly to Python. It will interpret what you have typed directly when you press Enter.
- To test this type: print("hello world"), and press F5 or the Green triangle (play) button.
- The first time you press run, you mind see this pop up. You can simply click on Run.





- Print() is the first python command you learn. It prints the value in the parentheses to the console.

Saving the code

To save the code, press ‘file’ and then ‘save as’. Now give the file a name, for example ‘mycode.py’ and save it in a directory/folder where you know how to find it. You can then run the script using the command line or another shell.

Use comments to add documentation to programs.

You can add comments to document your script. 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

Some data types (more to come)

- Integers (numbers) - don't use quotation marks.
- Strings (words) - use quotation marks

If we type a number between quotation marks, what is it?

```
Title = "1984"
```

Use variables to store values.

- 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.
- 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?

```
age = 35
first_name = "Clara"
```

- 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.

```
age = 35
first_name = "Clara"

print(age)
```

- 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:

```
11
12  print("Clara is 35 years old")
13
```

Or we could use the variables we defined earlier:

```
age = 35
first_name = "Clara"


print("Clara is 35 years old")

print(first_name, "is", age, "years old")
```

Here is what appears in the console once we run the script

```
In [1]: runfile('C:/Users/cturp1/
Clara is 35 years old
Clara is 35 years old

In [2]:
```

- 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:

```
1  # -*- coding: utf-8 -*-
2  """
3  Spyder Editor
4
5  This is a temporary script file.
6  """
7
8  age = 35
9  first_name = "Clara"
10
11 print(first_name, "is", age, "years old")
12 print(eye_color)
```

```
In [2]: runfile('C:/Users/cturp1/.spyder-py3/temp.py', wdir='C:/Users/ctu
Clara is 35 years old
Traceback (most recent call last):

  File "C:\Users\cturp1\.spyder-py3\temp.py", line 12, in <module>
    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

```
print(age + 3)

age = age + 3
print(age)

age_plus_three = age + 3
print(age_plus_three)
```

In the console:

```
38
38
41
```

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])
```

```
In [1]: runfile
h

In [2]:
```

- 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])
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])
```

```
In [1]: runfi
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"))
```

```
In [2]: runfile('C:/Users/c
<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.
- 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("hello" - "o")

print("Clara" + " " + "Turp")

print("Clara" * 10)
```

```
In [1]: Traceback (most recent call last):

  File "C:\Users\cturp1\.spyder-py3\temp.py", line 8, in <module>
    print("hello" - "o")

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


In [2]: runfile('C:/Users/cturp1/.spyder-py3/temp.py', wdir='C:/Users/cturp1/.spyder-py3')
Clara Turp

In [3]: runfile('C:/Users/cturp1/.spyder-py3/temp.py', wdir='C:/Users/cturp1/.spyder-py3')
ClaraClaraClaraClaraClaraClaraClaraClaraClaraClara

In [4]:
```

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'))
print(str(1) + '2')
print(int(1.10))
```

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)
```

```
In [1]: runfile('C:/User
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))
```

```
In [2]: runfile('C:/
6

In [3]:
```

- 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)
```

```
In [1]: runfile('C:/U
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))
```

```
In [6]: run
3.8
```

```
print(round(3.798))
```

```
In [5]: runfile('C:/U
4

In [6]:
```

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

```
In [2]: runfile('C:/Users/cturp1/.spyder-py
James' favorite fruit is banana
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.

```
favorite_fruit = "James' favorite fruit is banana"
print(favorite_fruit)
new_string = favorite_fruit.replace(" ", "; ")
print(new_string)
```

```
In [3]: runfile('C:/Users/cturp1/.spyder-py3/temp
James' favorite fruit is banana
James'; favorite; fruit; is; banana
```

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

```
favorite_fruit = "James' favorite fruit is banana"
print(favorite_fruit)
new_string = favorite_fruit.replace(" ", "; ", 1)
print(new_string)
```

```
In [4]: runfile('C:/Users/cturp1/.spyder-py3/
James' favorite fruit is banana
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(1, "A"))
```

```
In [7]: runfile('C:/Users/cturp1/.spyder-py3/temp.py', wdir='C:/Users/ctur
Traceback (most recent call last):

  File "C:\Users\cturp1\.spyder-py3\temp.py", line 8, in <module>
    print(max(1, "A"))

TypeError: '>' not supported between instances of 'str' and 'int'


In [8]:
```

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)
```

```
In [8]: runfile('C:/Users/cturp1/.spyder-py3/temp.py', wdir='C:/Users/cturp1/.spyder
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

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:', temperatures)
```

```
print('length:', len(temperatures))
```

```
In [2]: runfile('C:/Users/cturp1/.spyder-py3/temp
temperatures: [17.3, 17.5, 17.7, 17.5, 17.6]

In [3]: runfile('C:/Users/cturp1/.spyder-py3/temp
length: 5
```

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

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

```
temperatures = [17.3, 17.5, 17.7, 17.5, 17.6]
print(temperatures[0])
print(temperatures[4])
```

```
In [4]: ru
17.3
17.6
```

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

```
In [5]: runfile('C:/Users/cturp1/.spyder-py
Traceback (most recent call last):

  File "C:\Users\cturp1\.spyder-py3\temp.py
    print(temperatures[5])

IndexError: list index out of range
```

Lists' values can be replaced by assigning 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)
```

```
In [7]: runfile('C:/Users/cturp1/.spyder-py3/t
[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.append(19.7)
print(temperatures)
```

```
[17.3, 17.5, 17.7, 17.5, 17.6]
list is now:  [14.7, 17.5, 17.7, 17.5, 17.6]
[14.7, 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 item:', primes)
```

```
In [1]: runfile('C:/Users/cturp1/.spyder-py3/temp.py',
primes before removing last item: [2, 3, 5, 7, 9]
primes after removing last item: [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"]
print(goals)
print(values)
```

```
In [2]: runfile('C:/Users/cturp1/.spyder-py3/temp.py', wdir='C:/Users/cturp1/.
[1, 'Create lists.', 2.0, 'Extract items from lists.', 3, 'Modify lists.']
[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
if mass > 3.0:
    print(mass, 'is larger')

mass = 2.07
if mass > 3.0:
    print (mass, 'is larger')
```

```
In [3]: runfile(
3.54 is larger
```

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

mass = 2.07
if mass == 3.0:
    print (mass, 'is 3.0')
```

```
In [5]: runfile('C:
3.54 is not 3.0
```

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

```
firstName="Jon"
 lastName="Smith"
```

IndentationError: unexpected indent

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')
```

```
In [6]: runfile('C:/
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')
```

```
In [10]: runfile(
py3')
9.7 is HUGE

In [11]: runfile(
py3')
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.

```
grade = 85

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

```
In [12]: runfile('
py3')
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, 21.1]:
    print(temperature)
```

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

```
temperatures = [17.5, 15.9, 19.4, 21.1]

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

- The output of all these is:

```
In [3]: runf
17.5
15.9
19.4
21.1
```

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, 21.1]

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

```
In [5]: runfile('C:/Users/cturp1/.spyder-py3/temp.
  File "C:\Users\cturp1\.spyder-py3\temp.py", line
    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”.

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.

```
primes = [2, 3, 5]

for p in primes:
    squared = p ** 2
    cubed = p ** 3
    print(p, squared, cubed)
```

```
In [1]: runf
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
    - Exactly the legal indices of a list or character string of length N

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

```
In [2]:
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!")
```

```
In [3]: r
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 the first 10 integers.

total = 0
for number in range(10):
   total = total + (number + 1)

print(total)
```

```
In [4]
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')
```

```
In [5]: runfile('C:
3.54 is larger
9.22 is larger
```

- You can use loops and conditionals to “evolve” the values of variables.

```python
velocity = 10.0

for i in range(5): # execute the loop 5 times
    print(i, ':', velocity)

    if velocity > 20.0:
        print('moving too fast')
        velocity = velocity - 5.0

    else:
        print('moving too slow')
        velocity = velocity + 10.0

print('final velocity:', velocity)
```

```
In [6]: runfile('C:/Users/
0 : 10.0
moving too slow
1 : 20.0
moving too slow
2 : 30.0
moving too fast
3 : 25.0
moving too fast
4 : 20.0
moving too slow
final velocity: 30.0
```

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()
```

```
In [7]: run

In [8]: run
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)
```

```
In [9]: runfile('C:/
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 actual values:', a)

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

```
In [10]: runfile('C:/Users/cturp1/.spyder-py3/t
py3')
average of actual values: 2.6666666666666665
average of empty list: None
```

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

Exercise:

- Transform this loop into a function that returns the final velocity.
- This function should have one parameter, the value of the velocity variable.

```
velocity = 10.0

for i in range(5): # execute the loop 5 times
    print(i, ':', velocity)

    if velocity > 20.0:
        print('moving too fast')
        velocity = velocity - 5.0

    else:
        print('moving too slow')
        velocity = velocity + 10.0

print('final velocity:', velocity)
```

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".
- Using string, one of the modules in the standard library:

```
import string

print(string.capwords('capitalise this sentence please.'))
```

```
In [11]: runfile('C:/Users/cturp1/
py3')
Capitalise This Sentence Please.
```

- You have to refer to each item with the module's name.

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

- Works just like help for a function.

```
help(string)
```

Help on module string:

```
NAME
    string - A collection of string constants.

MODULE REFERENCE
    https://docs.python.org/3.8/library/string

    The following documentation is automatically generated from the Python
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the
    location listed above.

DESCRIPTION
    Public module variables:

    whitespace -- a string containing all ASCII whitespace
    ascii_lowercase -- a string containing all ASCII lowercase letters
    ascii_uppercase -- a string containing all ASCII uppercase letters
    ascii_letters -- a string containing all ASCII letters
    digits -- a string containing all ASCII decimal digits
    hexdigits -- a string containing all ASCII hexadecimal digits
    octdigits -- a string containing all ASCII octal digits
    punctuation -- a string containing all ASCII punctuation characters
    printable -- a string containing all ASCII characters considered printable
```

Import specific items from a library module to shorten programs.

- Use from ... import ... to load only specific items from a library module.
- Then refer to them directly without library name as prefix.

```
from string import ascii_letters

print('The ASCII letters are', ascii_letters)
```

```
In [13]: runfile('C:/Users/cturp1/.spyder-py3/temp.py', wdir='C:/Users/cturp1/
py3')
The ASCII letters are abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
```

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 string as s

print(s.capwords('capitalise this sentence again please.'))
```

- 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.

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