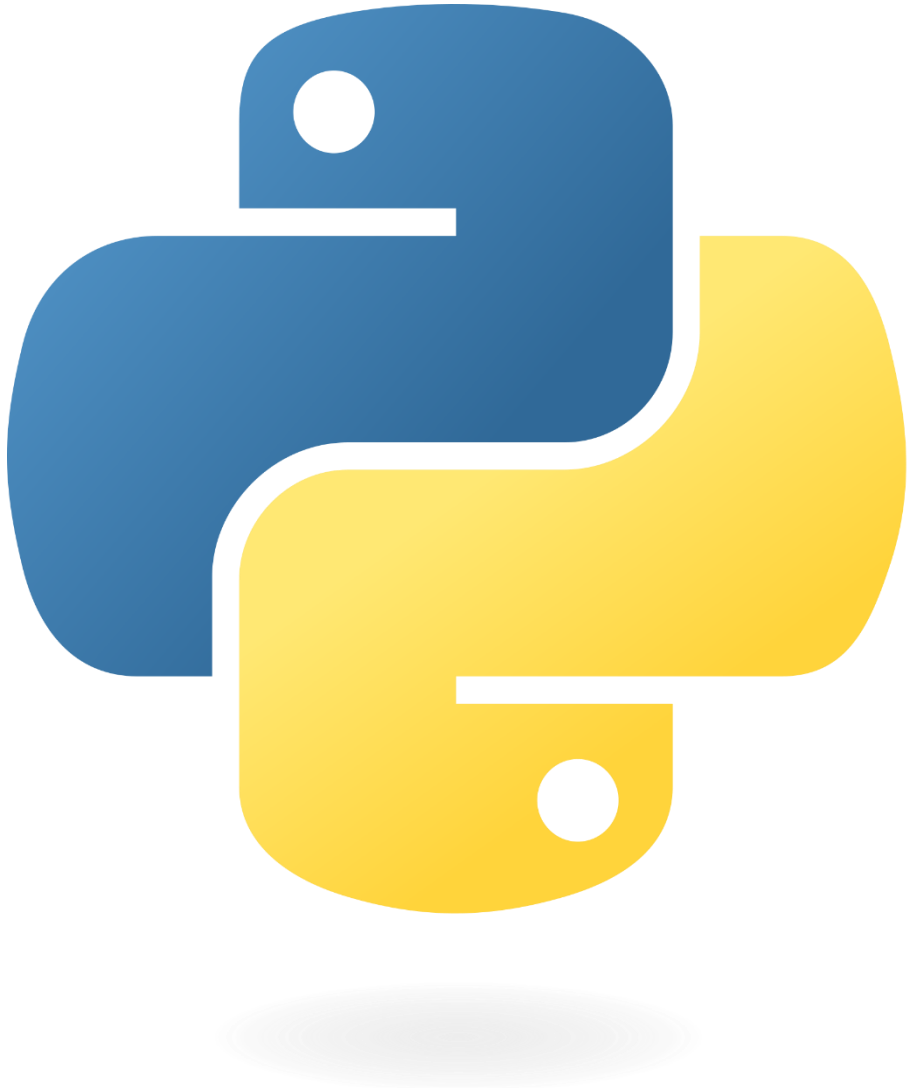
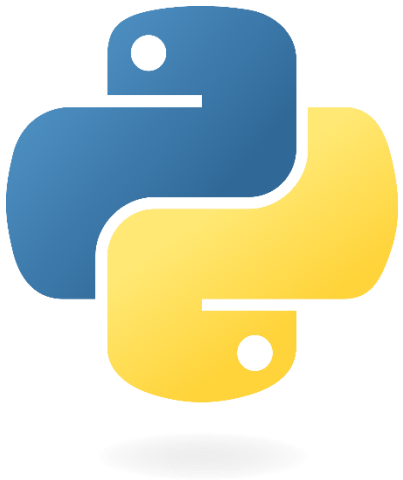
Python

- Python is a high-level, interpreted programming language that is widely used for web development, automation, data science, artificial intelligence, and more.



Tutorial Outline – Part 1

- What is Python?
- Operators
- Variables
- Functions
- Lists



Python is a popular, general-purpose programming language that's used for many tasks, including web development, data science, and machine learning. It's known for its readability and ease of use, making it a good choice for beginners.

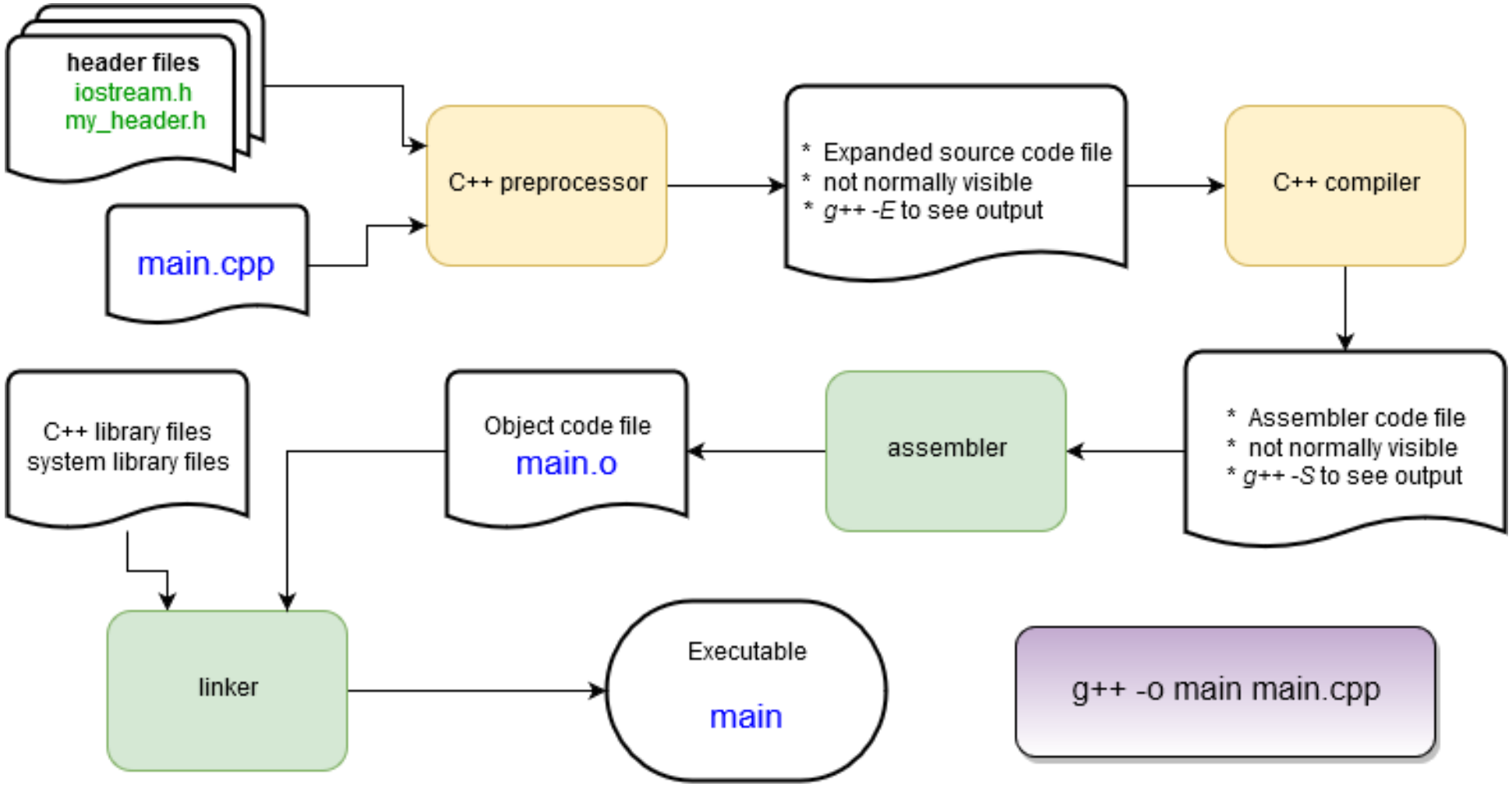
Some History

- “Over six years ago, in December 1989, I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas…I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus).”

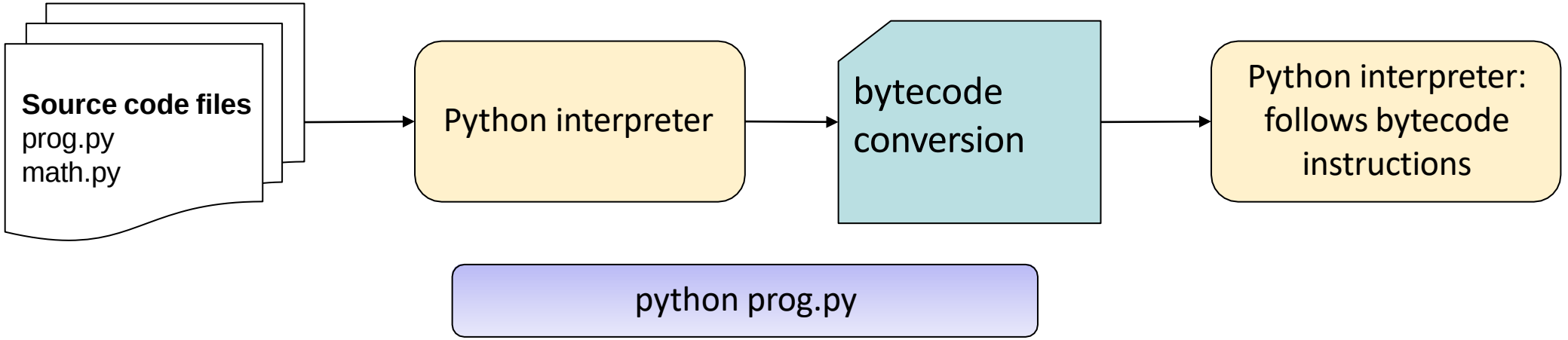
  –Python creator Guido Van Rossum, from the foreward to *Programming Python (1st ed.)*

- Goals:
  - An easy and intuitive language just as powerful as major competitors
  - Open source, so anyone can contribute to its development
  - Code that is as understandable as plain English
  - Suitability for everyday tasks, allowing for short development times

Compiled Languages (ex. C++ or Fortran)



Interpreted Languages (ex. Python or R)



- A lot less work is done to get a program to start running compared with compiled languages!
- Python programs start running immediately – no waiting for the compiler to finish.
- Bytecodes are an internal representation of the text program that can be efficiently run by the Python interpreter.
- The interpreter itself is written in C and is a compiled program.

The Python Prompt

- The standard Python prompt looks like this:

```
[bgregor@scc2 bg]$ python
Python 3.6.2 (default, Aug 30 2017, 15:46:55)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-3)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- The IPython prompt in Spyder looks like this:

```
Python 3.6.3 |Anaconda, Inc.| (default, Oct 15 2017, 03:27:45) [MSC v.1900 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 6.1.0 -- An enhanced Interactive Python.

In [1]:
```

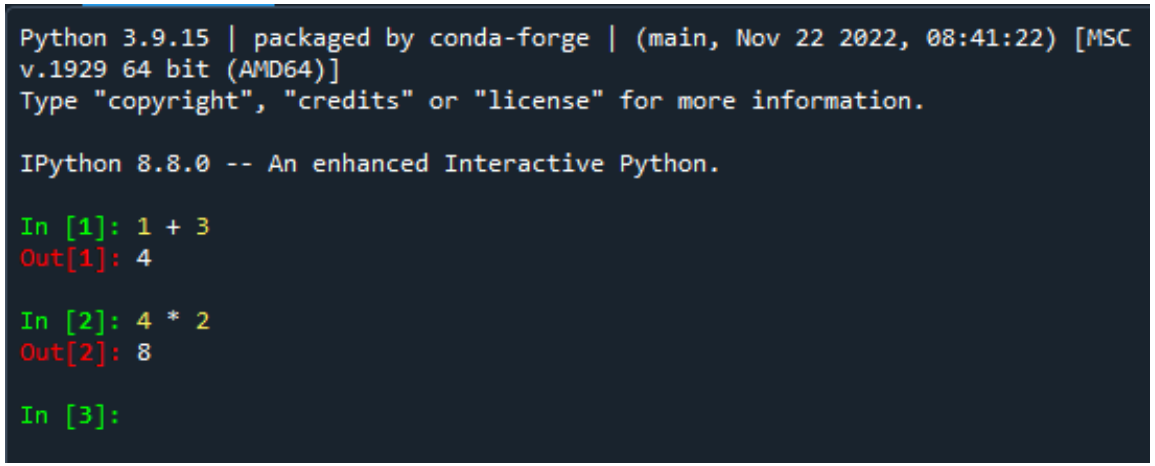
- Spyder is an open-source cross-platform integrated development environment for scientific programming in the Python language. Spyder integrates with a number of prominent packages in the scientific Python stack, as well as other open-source software.

- IPython adds some handy behavior around the standard Python prompt.

Operators

- Python supports a wide variety of operators which act like functions, i.e. they do something and return a value:
  - Arithmetic: `+ - * / // % **`
  - Logical: `and or not`
  - Comparison: `> < >= <= != ==`
  - Assignment: `=`
  - Bitwise: `& | ~ ^ >> <<`
  - Identity: `is is not`
  - Membership: `in not in`

Try Python as a calculator



- Go to the Python prompt.
- Try out some arithmetic operators:

```
+    -    *    /   //   %   **   ==   ( )  and
```

- Can you identify what they all do?

Operators

Operator	Function
+	Addition
-	Subtraction
*	Multiplication
/	Division (25 / 4 = 6.25)
//	Integer Division (25 // 4 = 6)
%	Remainder (aka *modulus*)
**	Exponentiation
==	Equals
and or not	Boolean operations
> < <= >=	Comparison

More Operators

- Try some comparisons and Boolean operators. *True* and *False* are the keywords indicating those values:

```
In [3]: 4 > 5
Out[3]: False

In [4]: 6 > 3 and 3 > 0
Out[4]: True

In [5]: not False
Out[5]: True

In [6]: True and (False or not False)
Out[6]: True
```

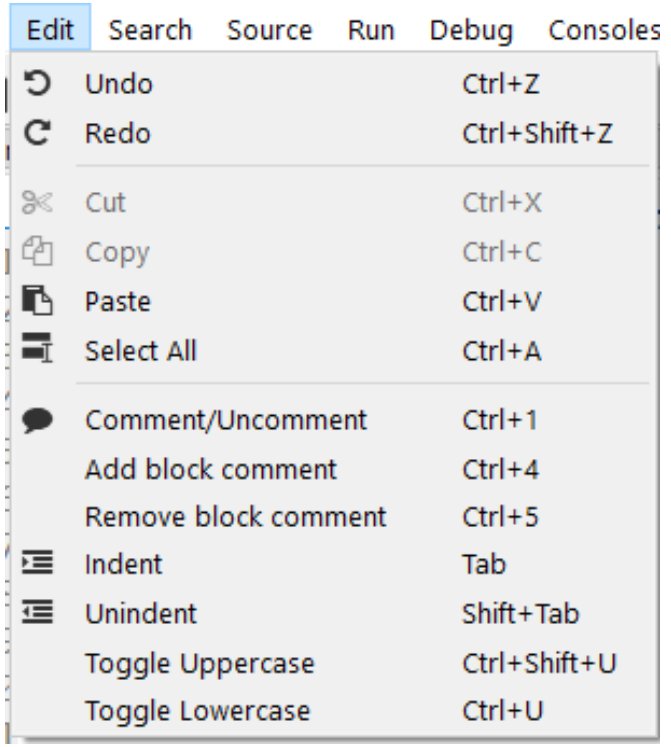
Comments

- # is the Python comment character. On any line everything after the # character is ignored by Python.

- There is no multi-line comment character as in C or C++.

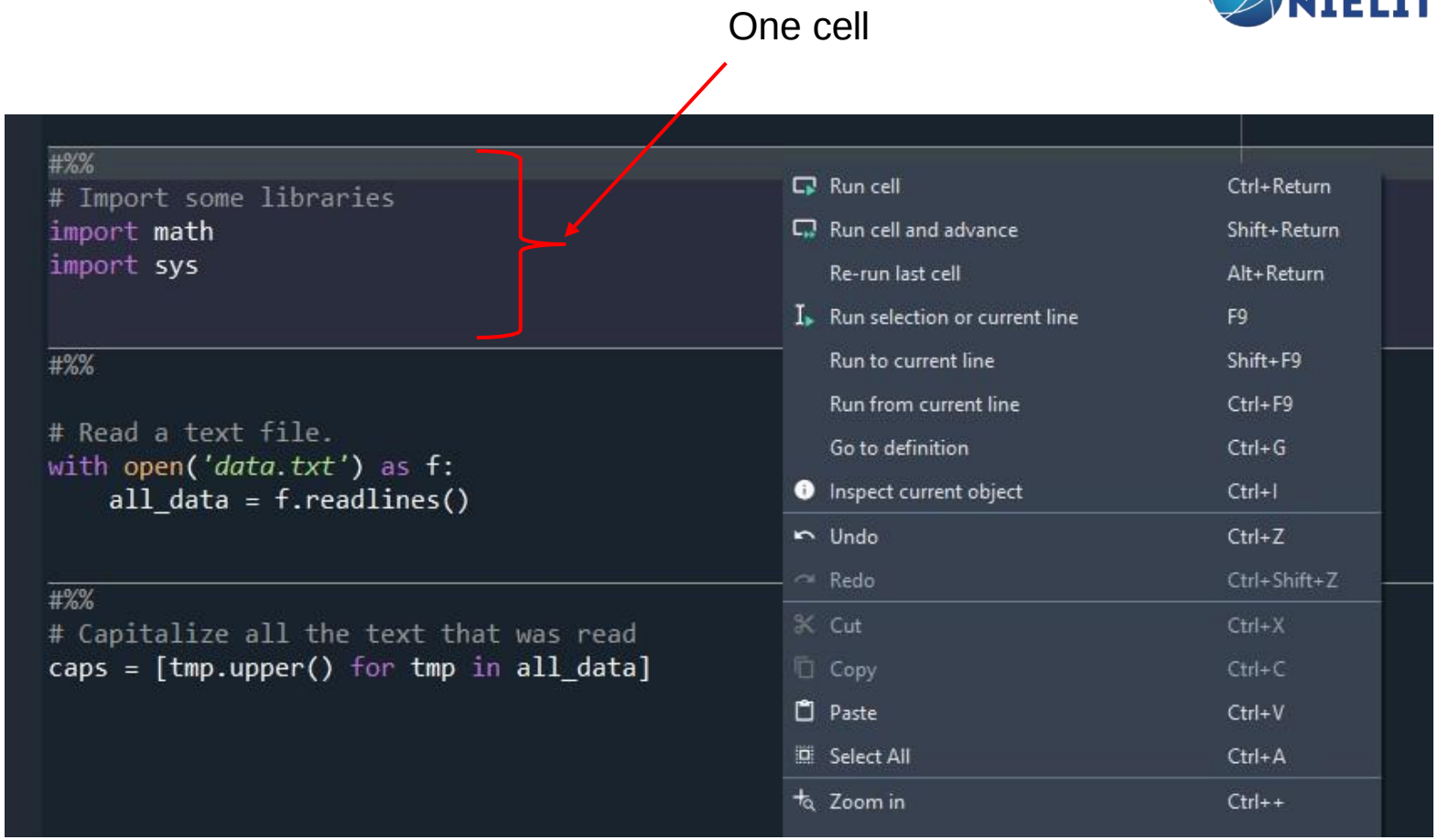
- An editor like Spyder makes it very easy to comment blocks of code or vice-versa. Check the *Edit* menu

```
a=1
b=2
# this is a comment
c=3 # this is also a comment
# this is a
# multiline comment
```



Spyder Cells

- This is a Spyder-specific tool for helping you to run snippets of code in the file editor.
- Every time the characters #%% are seen Spyder treats that section as a “cell”.
- Right-click to run a single cell.
  - Or use the keyboard shortcuts



Note: A “right-click” on a Mac is “click while holding down the Control key”

Variables

- Variables are assigned values using the = operator
- In the Python **console**, typing the name of a variable prints its value
  - **Not true in a script!**
  - Visualize Assignment

- Variables can be reassigned at any time
- Variable type is not specified
- Types can be changed with a reassignment

```
In [1]: a=1

In [2]: b=2

In [3]: a
Out[3]: 1

In [4]: b
Out[4]: 2

In [5]: a=b

In [6]: a
Out[6]: 2

In [7]: b=-0.15
```

Variables cont'd

- Variables refer to a value stored in memory and are created when first assigned
- Variable names:
  - Must begin with a letter (a - z, A - Z) or underscore _
  - Other characters can be letters, numbers or _
  - Are case sensitive: capitalization counts!
  - Can be any reasonable length
- Assignment can be done *en masse*:

```
x = y = z = 1
```

- Multiple assignments can be done on one line:

```
x, y, z = 1, 2.39, 'cat'
```

Try these out!

Variable Data Types

- Python determines data types for variables based on the context
- The type is identified when the program **runs**, using **dynamic typing**
  - Compare with compiled languages like C++ or Fortran, where types are identified by the programmer and by the compiler **before** the program is run.
- Run-time typing is very convenient and helps with rapid code development

Variable Data Types

Numbers	Integers and floating point (64-bit)
Complex numbers	`x = complex(3,1)` or `x = 3+1j`
Strings	`"cat"` or `'dog'`
Boolean	`True` or `False`
Lists, **dictionaries**, **sets**, and **tuples**	These hold collections of values
Specialty types	Files, network connections, etc.
Custom types	User- or library-defined types using Python classes

Variable modifying operators

- Some additional arithmetic operators that modify variable values:

Operator	Effect	Equivalent to…
x += y	Add the value of *y* to *x*	x = x + y
x -= y	Subtract the value of *y* from *x*	x = x - y
x *= y	Multiply the value of *x* by *y*	x = x * y
x /= y	Divide the value of *x* by *y*	x = x / y

- The += operator is by far the most used of these.

Strings

- Strings are a basic data type in Python.

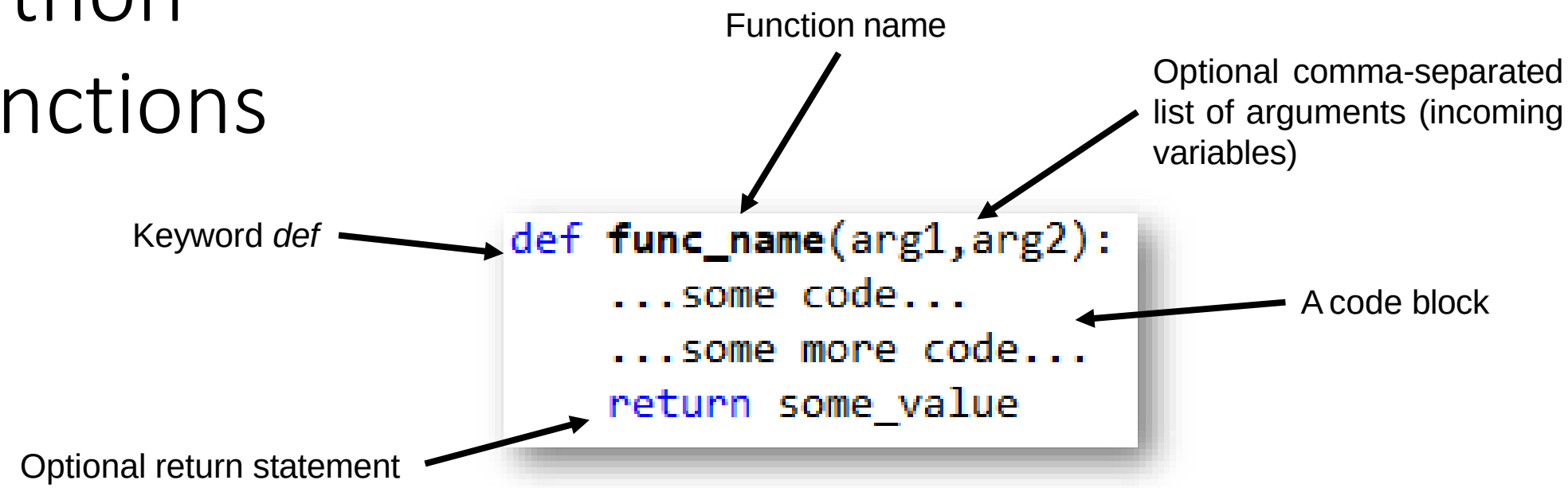
- Indicated using pairs of single '' or double "" quotes.
- Multiline strings use a triple set of quotes (single or double) to start and end them.

```
'cat'

"dog"

"What's that?"

'They said "hello"'

''' This is
    a multiline
    string '''
```

Functions

- Functions are used to create pieces of code that can be used in a program or in many programs.
- The use of functions is to logically separate a program into discrete computational steps.
- Programs that make heavy use of function definitions tend to be easier to:
  - develop
  - debug
  - maintain
  - understand

Python functions



- The return value can be any Python type
- If the return statement is omitted a special *None* value is still returned.
- The arguments are optional but the parentheses are required!
- **Functions must be defined** before they can be called.

Sample Built-In Functions

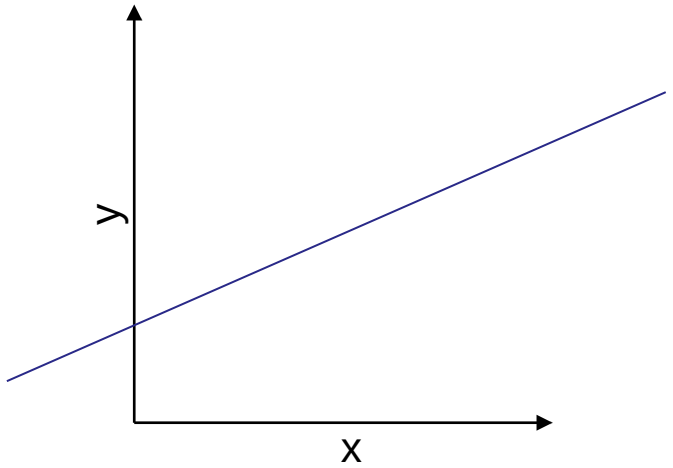
- Let’s try a few useful built-in functions…

- print()
- dir()
- type()
- help()

Visualize a Function Call

- Here's a simple function call to calculate the equation of a line.

$$y = A * x + B$$



Write a Function

```
def mathcalc( ...your args here...):
    # ... do something ...
    return ...your calculated value...

# Call your function and print the
# result
ans = mathcalc( ... args ...)
print(ans)
```

- In Spyder's editor:
  - Define a function called **mathcalc** that takes 3 numbers as arguments and returns their sum divided by their product.
  - Save the file and run it. Here's some sample output to check your result.

`mathcalc(1,2,3)` → `returns 1.0`

`mathcalc(4, -2.5, 3.0)` → `returns -0.15`

Which code sample is easier to read?

- C:

```c
float avg(int a, int b, int c){
float sum = a + b + c ;
return sum / 3.0 ;}
```

or

```c
float avg(int a, int b, int c)
{
   float sum = a + b + c ;
   return sum / 3.0 ;
}
```

- Matlab:

```
function a = avg(x,y,z)
a = x + y + z ;
a = a / 3.0 ;
end
```

or

```
function a = avg(x,y,z)
   a = x + y + z ;
   a = a / 3.0 ;
end
```

Which code sample is easier to read?

- Most languages use special characters ({ } pairs) or keywords (*end*, *endif*) to indicate sections of code that belong to:
  - Functions
  - Control statements like *if*
  - Loops like *for* or *while*
- Python instead uses the **indentation** that programmers use anyway for readability.

C

```c
float avg(int a, int b, int c)
{
    float sum = a + b + c ;
    return sum / 3.0 ;
}
```

Matlab

```matlab
function a = avg(x,y,z)
    a = x + y + z ;
    a = a / 3.0 ;
end
```

The Use of Indentation

- Python uses whitespace (spaces or tabs) to define *code blocks*.
- Code blocks are logical groupings of commands. They are **always** preceded by a colon **:**

```
def avg(x,y,z):
        all_sum = x + y + z
        return all_sum / 3.0

def mean(x,y,z):
        return (x + y + z) / 3.0
```

A code block

Another code block

- This pattern is consistently repeated throughout Python syntax.
- Spaces or tabs can be mixed in a file but **not** within a code block.

Function Return Values

- A function can return any Python value.
- Function call syntax:

```
A = some_func()   # some_func returns a value
Another_func()    # ignore return value or nothing returned
b,c = multiple_vals(x,y,z)    # return multiple values
```

- Open *function_calls.py* for some examples

Function arguments

- Function arguments can be required or optional.
- Optional arguments are given a default value

```
def my_func(a,b,c=10,d=-1):
    …some code…
```

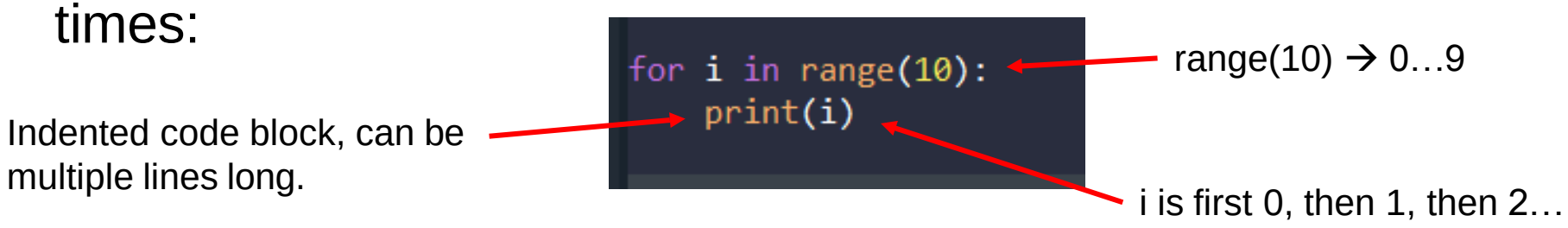
- To call a function with optional arguments:
- Optional arguments can be used in the order they're declared or out of order if their name is used.

```
my_func(x,y)              # a=x, b=y, c=10, d=-1
my_func(x,y,z)            # a=x, b=y, c=z, d=-1
my_func(x,y,d=w,c=z)      # a=x, b=y, c=z, d=w
```

For Loops

- *For* loops are used to repeat commands a specified number of times.
- Python has a built-in function to produce a sequence of numbers, *range()*
  - range(N) → numbers 0 to (N-1)
  - range(M, N) → numbers M to (N-1)
  - range(M, N, P) → numbers M to (N-1) in steps of P

- Put that together with a *for* loop and run commands a specified number of times:



Project Euler Problem 6

- Write a function that solves this problem for an arbitrary amount N of natural numbers (1,2,3,…,N)
- In Spyder's editor write a function "euler6" that takes an argument N and returns this calculation:

Sum square difference

Problem 6

The sum of the squares of the first ten natural numbers is,

$$1^2 + 2^2 + \ldots + 10^2 = 385$$

The square of the sum of the first ten natural numbers is,

$$(1 + 2 + \ldots + 10)^2 = 55^2 = 3025$$

Hence the difference between the sum of the squares of the first ten natural numbers and the square of the sum is $3025 - 385 = 2640$.

Find the difference between the sum of the squares of the first one hundred natural numbers and the square of the sum.

```python
def euler6(N):
    # do your calculations here.
    # hint: try a "for" loop...
    # don't forget to return the result.
    return ...your answer...


# This should print 2640
print(euler6(10))

# This should print 25164150
print(euler6(100))
```

Lists

- A Python list is a general purpose 1-dimensional container for variables.
  - i.e. it is a row, column, or vector of things
- Lots of things in Python act like lists or use list-style notation.
- Variables in a list can be of any type at any location, including other lists.
- Lists can change in size: elements can be added or removed

Making a list and checking it twice…

- Make a list with [ ] brackets.
- Append with the *append()* function
- Create a list with some initial elements
- Create a list with N repeated elements

Try these out yourself!
Add some print() calls to see the lists.

```
list_1 = []

list_1.append(1)
list_1.append('A string!')
list_1.append([])


list_2 = [4, 5, -23.0+4.1j, 'cat']


list_3 = 10 * [42]
```

List functions

- Try *dir(list_1)*
- List have a number of built-in functions
- Let's try out a few…
- Also try the len() function to see how many things are in the list: *len(list_1)*

```
'append',
'clear',
'copy',
'count',
'extend',
'index',
'insert',
'pop',
'remove',
'reverse',
'sort']
```

List Indexing

- Elements in a list are accessed by an index number.

- Index #’s start at 0.

- List: `x=['a', 'b', 'c', 'd' ,'e']`

- First element: `x[0]` → `'a'`
- Nth element: `x[2]` → `'c'`
- Last element: `x[-1]` → `'e'`
- Next-to-last: `x[-2]` → `'d'`

List
Slicing

```
x=['a', 'b', 'c', 'd' ,'e']
x[0:1] → ['a']
x[0:2] → ['a','b']
x[-3:] → ['c', 'd', 'e']
# Third from the end to the end
x[2:5:2] → ['c', 'e']
```

- Slice syntax: `x[start:end:step]`
  - The start value is inclusive, the end value is exclusive.
  - Start is optional and defaults to 0.
  - Step is optional and defaults to 1.
  - Leaving out the end value means “go to the end”
  - Slicing always returns a **new list copied from the existing list**

List assignments and deletions

- Lists can have their elements overwritten or deleted (with the *del)* command.
  - Note the *del* command does not use parentheses – it's sort of like a function call.

```
x=['a', 'b', 'c', 'd' ,'e']

x[0] = -3.14 → x is now [-3.14, 'b', 'c', 'd', 'e']

del x[-1] →  x is now [-3.14, 'b', 'c', 'd']
```

DIY Lists

- In the Spyder editor try the following things:

- Assign some lists to some variables. a = [1,2,3]
  - Try an empty list, repeated elements, initial set of elements
- Add two lists: a + b What happens?

b = 3*[‘xyz’]

- Try list indexing, deletion, functions from *dir(my_list)*
- Try assigning the result of a list slice to a new variable

More on Lists and Variables

- What happens when we pass a list to a function?

- Or we do an assignment with it?

```
def change_list(my_list, val):
    if len(my_list) > 0:
        first_val = my_list.pop(0)
    my_list.extend([val, first_val])
    return my_list

x=[1,2]

# call change_list, overwrite x
x = change_list(x,10)

# Do we need the return value?
change_list(x, 20)

# What about an assignment...
y = x
change_list(y,-1.5)
print(x)
```

Let's visualize it!

Copying Lists

- How to copy (2 ways…there are more!):
  - `y = x[:]` or `y=list(x)`
- Many data types in Python have this same behavior

Introduction to Python Part 2

National Institute of Electronics & Information Technology

Tutorial Outline – Part 2

- If / else
- Classes
- Loops
- Tuples and dictionaries
- Modules
- Some useful modules
- Script setup
- Development notes

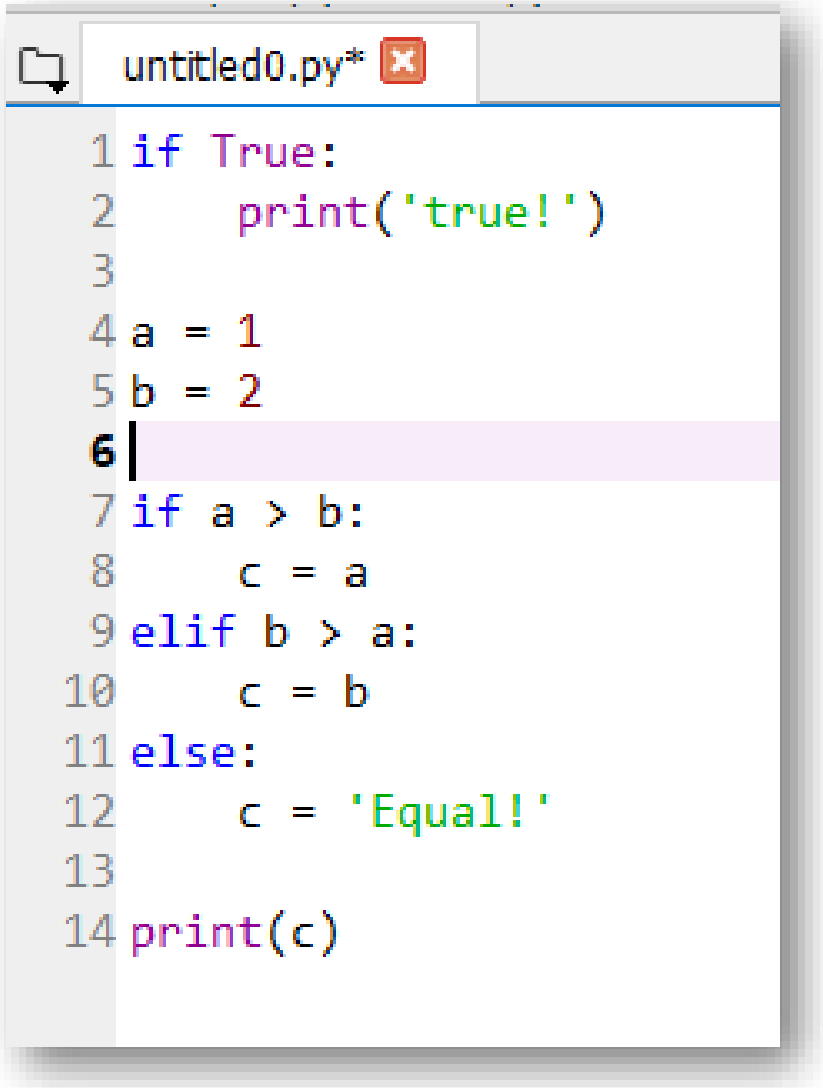
If / Else

- *If*, *elif*, and *else* statements are used to implement conditional program behavior
- Syntax:

```
if Boolean_value:
  …some code
elif Boolean_value:
  …some other code
else:
  …more code
```

- *elif* and *else* are not required – use them to chain together multiple conditional statements or provide a default case.

- Try out something like this in the Spyder editor.
- Do you get any error messages in the console?
- Try using an *elif* or *else* statement by itself without a preceding *if*. What error message comes up?



If / Else code blocks

- Python knows a code block has ended when the indentation is removed.

- Code blocks can be nested inside others therefore *if-elif-else* statements can be freely nested within others.
  - Or used in functions…

```
a = 1
b = 2
if a <= b:
    c = a
    print('a <= b')
    if c == 1:
        print('c is 1')
print('out of the if statement')
```

Project Euler Problem 1

- Let's code this!

Multiples of 3 or 5

Problem 1

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.

Find the sum of all the multiples of 3 or 5 below 1000.

```
def euler1_(N):
    ''' Sum of all natural numbers
    from 1 to N-1 that are multiples of
    3 or 5.'''
    # hint: use a for loop and the range()
    # function.
    return ...your answer...


# Prints 23
print(euler1(10))
# Prints 233168
print(euler1(1000))
```

Python Classes

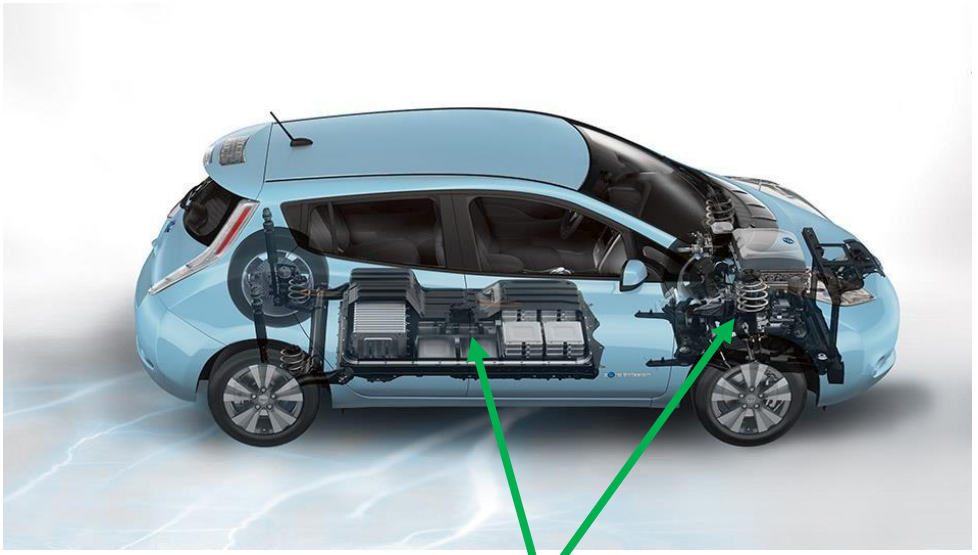
- OOP: Object Oriented Programming
- In OOP a *class* is a data structure that combines data with functions that operate on that data.
- An *object* is a variable whose type is a *class*
  - Also called an *instance* of a class
- Classes provide a lot of power to help organize a program and can improve your ability to re-use your own code.

Object-oriented programming

- Classes can contain data and methods (internal functions).
- Methods can call other code inside the class to implement complex behavior.
- This is a highly effective way of modeling real world problems inside of a computer program.

"Class Car"

public interface



internal data and methods

Object-oriented programming

- Python is a fully object oriented programming (OOP) language.
- Some familiarity with OOP is needed to understand Python data structures and libraries.
- You can write your own Python classes to define custom data types.

Track via separate variables

```
boston_pop = 685094
boston_sq_km = 232.1
boston_num_colleges = 35
```

Boston	
Population	685094
Area (km^2)	232.1
# of colleges	35

A class lets you bundle these into one variable

Writing Your Own Classes

- Define your own Python classes to:
  - Bundle together logically related pieces of data
  - Write functions that work on specific types of data
  - Improve code re-use
  - Organize your code to more closely resemble the problem it is solving.

```
class City:
    ''' A class to hold info about a city '''
    def __init__(self, name, area, pop, num_colleges):
        self.name = name
        self.area = area
        self.pop = pop
        self.num_colleges = num_colleges


    def is_best_city(self):
        return self.name == 'Boston'


boston = City('Boston', 685094, 232.1, 35)
new_york = City('New York City', 8804190, 1223.59, 120)

print(new_york.is_best_city())   # prints False
```

Syntax for using Python classes

Create an object, which is a variable whose type is a Python class.

Created by a call to the class or returned from a function.

```
# Open a file. This returns a file object.
file = open('some_file.txt')

# Read all the lines from the text file.
# Return them as a list.
lines = file.readlines()

# Get the filename
file.name # --> some_file.txt
```

Call a method for this object:

`object_name.method_name(args…)`

Access internal data for this object:

`object_name.data_name`

Classes bundle data and functions

- In Python, calculate the area of some shapes after defining some functions.

```
radius = 14.0
width_square = 14.0
a1 = area_circle(radius)          # ok
a2 = area_square(width_square)    # ok
a3 = area_circle(width_square)    # !! OOPS
```

- If we defined Circle and Rectangle classes with their own *area()* methods…it is not possible to miscalculate.

- Group data with matching functions into classes.

```
class Circle
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14.159 * self.radius**2

class Square:
    def __init__(self, width):
        self.width = width

    def area(self):
        return self.width**2

c1 = Circle(radius)
r1 = Square(width_square)

a1 = c1.area()
a2 = r1.area()
```

When to use your own class

- A class works best when you've done some planning and design work before starting your program.

- This is a topic that is best tackled after you're comfortable with solving programming problems with Python.

- Some tutorials on using Python classes:

  W3Schools: https://www.w3schools.com/python/python_classes.asp

  Python tutorial: https://docs.python.org/3.6/tutorial/classes.html

Strings Are a Class In Python

- Python defines a string class – **all strings in Python are objects.**

- This means strings have:
  - Their own internal (hidden) memory management to handle storage of the characters.
  - A variety of methods (functions) that operate on the stored string once you have a string object.
- You can't access string functions without a string – in Python the string provides its own functions.
  - C: strcat, strcmp, strlen functions
  - Matlab: strlength, isletter, etc
  - R: nchar, toupper, etc

String functions

- In the Python console, create a string variable called *mystr*

- type: *dir(mystr)*

- Try out some functions:

```
mystr = 'Hello!'

mystr.upper()

mystr.title()

mystr.isdecimal()

help(mystr.isdecimal)
```

- Need help? Try:

  help(mystr.title)

The len() function

- The len() function is not a string specific function.
- It'll return the length of any Python object that contains **any** countable thing.

```
len(mystr)→ 6
```

- In the case of strings it is the number of characters in the string.

String operators

- Try using the + and += operators with strings in the Python console.

```
a="Hello BU!"
print(a[4])
```

- + concatenates strings.
- += appends strings.
  - These are defined in the string class as functions that operate on strings.

- Index strings using square brackets, starting at 0.

String operators

- Changing elements of a string by an index is **not allowed**:

```
In [79]: a='Hello BU!'

In [80]: a[4] = 'O'
Traceback (most recent call last):

  File "<ipython-input-80-7c5733c2cb67>", line 1, in <module>
    a[4] = 'O'

TypeError: 'str' object does not support item assignment
```

- Python strings are **immutable**, i.e. they can't be changed.

Old School String Substitutions

- Python provides an easy way to stick variable values into strings called *substitutions*

%s means sub in value

variable name comes after a %

- Syntax for one variable:

```
'string with a %s' % variable
```

Variables are listed in the substitution order inside ()

- For more than one:

```
'x: %s  y: %s  z: %s' % (xval,yval,zval)
```

- Printing:

```
print('x: %s, y: %s, z:%s' % (xval,yval,2.0))
```

Recommended: f-string Substitutions

- f-strings are a more contemporary way to format strings.
- Use a lowercase *f* before the first quote.
- Put the names of variables, or function calls, in {} pairs inside the strings.

```
name = 'Boston'
school = f'{name} University'
```

```
result = f'{mathcalc(1,2,3)}'
```

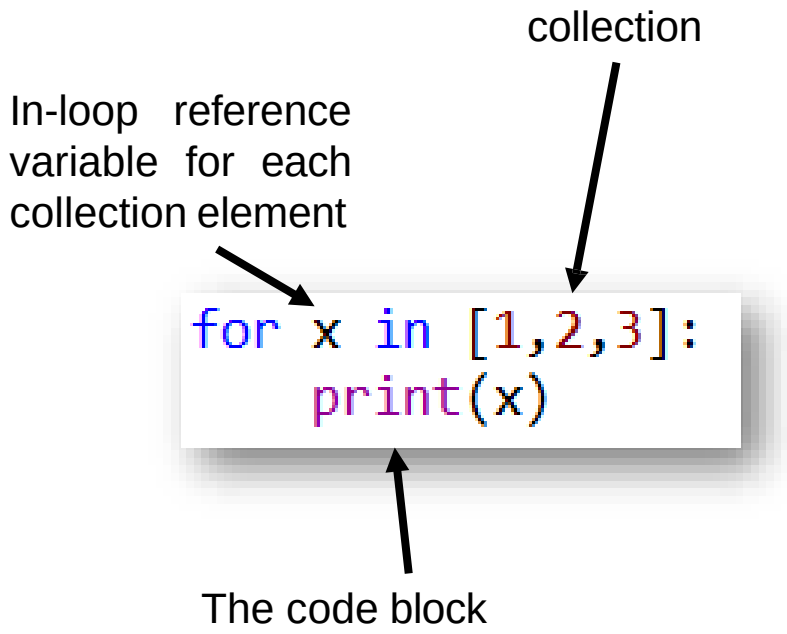
While Loops

- While loops have a condition and a code block.
  - the indentation indicates what's in the while loop.
  - The loop runs until the condition is false.
- The *break* keyword will stop a while loop running.
- In the Spyder edit enter in some loops like these. Save and run them one at a time. What happens with the 1st loop?

```
while True:
    print("looping!")


a=10
while a > 0:
    print(a)
    a -= 1

my_list=['a','b','c','d','e']
i=0
while i < len(my_list):
    print( my_list[i] )
    i += 1
    if i==3:
        break
```

For loops (again)

- *for* loops in general loop through a collection of things.
- The for loop syntax has a collection and a code block.
  - Each element in the collection is accessed in order by a reference variable
  - Each element can be used in the code block.



- The *break* keyword can be used in for loops too.

Processing lists element-by-element

- A for loop is a convenient way to process every element in a list.

- There are several ways:
  - Loop over the list elements
  - Loop over a list of index values and access the list by index
  - Do both at the same time
  - Use a shorthand syntax called a *list comprehension*

- Open the file *looping_lists.py*

Lists With Loops

- Open the file *read_a_file.py*
- This is an example of reading a file into a list. The file is shown to the right, *numbers.txt*
- We want to read the lines in the file into a list of strings (1 string for each line), then extract separate lists of the odd and even numbers.

numbers.txt

```
38,83,37,21,98
50,53,55,37,97
39,7,81,87,82
18,83,66,82,47
56,64,9,39,83
…etc…
```

- *read_a_file_low_mem.py* is a modification that uses less memory by processing the file line-by-line.

Tuples

- Tuples are lists whose elements can’t be changed.
  - Like strings they are immutable
- Indexing (including slice notation) is the same as with lists.

```
# a tuple
a = 10,20,30
# a tuple with optional parentheses
b = (10,20,30)
# a list
c = [10,20,30]
# ...turned into a tuple
d = tuple(c)

# and a tuple turned into a list
e = list(d)
```

Return multiple values from a function

- Tuples are used to return multiple values from a function.
- Python syntax can automatically unpack a tuple return value.

```
def min_max(x):
    ''' Return the maximum and minimum
        values of x '''
    minval = min(x)
    maxval = max(x)
    # a tuple return...
    return minval,maxval


a = [10,4,-2,32.1,11]

val = min_max(a)
min_a = val[0]
max_a = val[1]

# Or, easier...
min_a, max_a = min_max(a)
```

Dictionaries

- Dictionaries are another basic Python data type that are tremendously useful.

- Create a dictionary with a pair of curly braces:

```
x = {}
```

- Dictionaries store *values* and are indexed with *keys*

- Create a dictionary with some initial values:

```
x = {'a_key':55, 100:'a_value', 4.1:[5,6,7]}
```

Dictionaries

- Values can be any Python thing
- Keys can be primitive types (numbers), strings, tuples, and some custom data types
  - Basically, any data type that is **immutable**
- Lists and dictionaries cannot be keys but they can stored as values.
- Index dictionaries via keys:

```
x['a_key'] → 55
x[100] → 'a_value'
```

Try Out Dictionaries

- Create a dictionary in the Python console or Spyder editor.

```
x = {}
x[3] = -3.3
x[10.2] = []

print(x)
```

- Add some values to it just by using a new key as an index. Can you overwrite a value?

- Try *x.keys()* and *x.values()*

- Try: `del x[valid_key]` → deletes a key/value pair from the dictionary.

Modules

- Python modules, aka *libraries* or *packages*, add functionality to the core Python language.
- The Python Standard Library provides a very wide assortment of functions and data structures.
  - Check out their Brief Tour for a quick intro.
- Distributions like Anaconda provides dozens or hundreds more
- You can write your own libraries or install your own.

PyPI

- The Python Package Index is a central repository for Python software.
  - Mostly but not always written in Python.
- A tool, *pip*, can be used to install packages from it into your Python setup.
  - Anaconda provides a similar tool called *conda*
- Number of projects (as of January 2023): **430,524**
- You should always do your due diligence when using software from a place like PyPI. Make sure it does what you think it's doing!

Python Modules on the SCC

- Python modules should not be confused with the SCC *module* command.
- For the SCC there are instructions on how to install Python software for your account or project.
- Many SCC modules provide Python packages as well.
  - Example: tensorflow, pycuda, others.
- Need help on the SCC? Send us an email: help@scc.bu.edu

Importing Libraries

- The *import* command is used to load a library.
- The name of the library is prepended to function names and data structures in the module.
  - The preserves the library *namespace*
- This allows different libraries to have the same function names – when loaded the library name keeps them separate.

```
import math

z=math.sin(0.1)

print(z)

dir(math)

help(math.ceil)
```

Try these out!

Fun with *import*

- The *import* command can strip away the module name:

```
from math import *
```

- Or it can import select functions:

```
from math import cos
from math import cos,sqrt
```

- Or rename on the import:

```
from math import sin as pySin
```

Easter Eggs

```
# Try to load curly braces for Python
from __future___import braces

# Proof that Python programmers have more fun
import antigravity
```

Fun with *import*

- The *import* command can also load your **own** Python files.

- The Python file to the right can be used in another Python script:

```
# Don't use the .py ending
import myfuncs
x = [1,2,3,4]
y = myfuncs.get_odds(x)
```

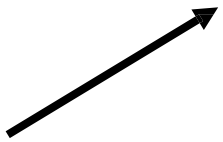
myfuncs.py

```
def get_odds(lst):
    ''' Gets the odd numbers in a list.

        lst: incoming list of integers
        return: list of odd integers '''
    odds = []
    for elem in lst:
        # Odd if there's a remainder when
        # dividing by 2.
        if elem % 2 != 0:
            odds.append(elem)
    return odds
```

- Splitting your code into multiple files helps with development and organization.

Import details

- Python reads and executes a file when the file is:
  - opened directly: `python somefile.py`
  - imported: `import somefile`
- Lines that create variables, call functions, etc. are **all executed**.
- Here these lines will run when it's imported into another script!

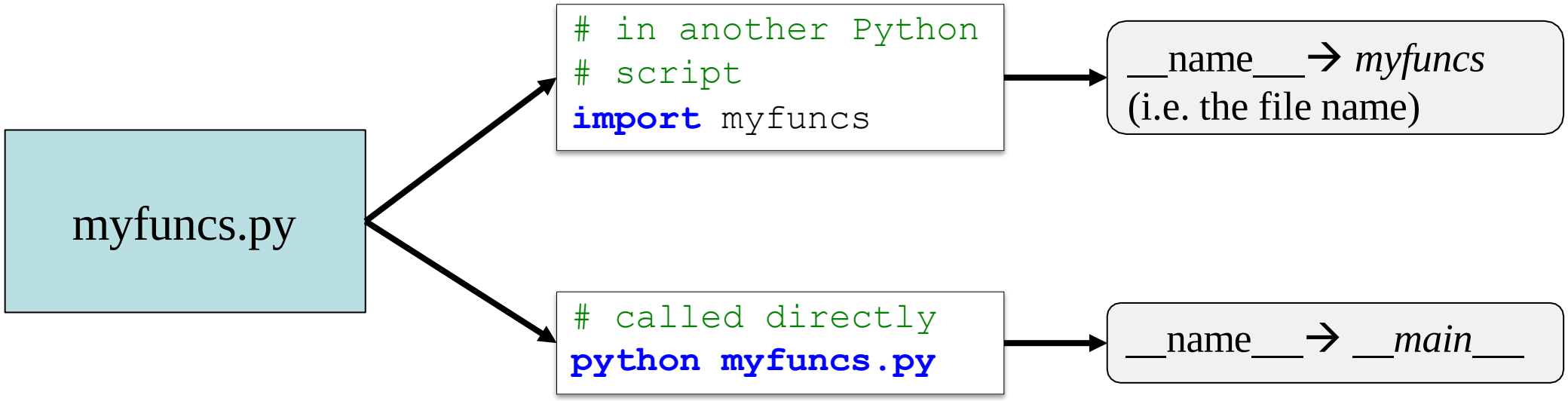


myfuncs.py

```
def get_odds(lst):
    ''' Gets the odd numbers in a list.

        lst: incoming list of integers
        return: list of odd integers '''
    odds = []
    for elem in lst:
        # Odd if there's a remainder when
        # dividing by 2.
        if elem % 2 != 0:
            odds.append(elem)
    return odds

x = [1,2,3,4]
y = get_odds(x)
print(y)
```

The __name_____attribute

- Python stores object information in hidden fields called *attributes*
- Every file has one called *__name__* whose value depends on how the file is used.



The __name__ attribute

- *__name__* can be used to make a Python scripts usable as a standalone program **and** as imported code.

- Now:
  - `python myfuncs.py` → __name__ has the value of '__main__' and the code in the *if* statement is executed.
  - `import myfuncs` → __name__ is 'myfuncs'
    - and the *if* statement does not run.

myfuncs.py

```
def get_odds(lst):
    ''' Gets the odd numbers in a list.

        lst: incoming list of integers
        return: list of odd integers '''
    odds = []
    for elem in lst:
        # Odd if there's a remainder when
        # dividing by 2.
        if elem % 2 != 0:
            odds.append(elem)
    return odds


if __name__=='__main__':
    x = [1,2,3,4]
    y = get_odds(x)
    print(y)
```

Very Useful Modules

- numpy is a Python library that provides efficient multidimensional numeric data structures
- matplotlib is a popular plotting library
  - Remarkably similar to Matlab plotting commands!
- scipy provides a wide variety of numerical algorithms:
  - Integrations, curve fitting, machine learning, optimization, root finding, etc.
  - Built on top of numpy
- pandas is used for data analysis using DataFrame structures
  - Very similar to what you find in R.

numpy

- numpy provides data structures written in compiled C code
- Many of its operations are executed in compiled C or Fortran code, not Python.

- Check out *numpy_basics.py*

numpy datatypes

- Unlike Python lists, which are generic containers, numpy arrays are *typed* and hold a single type of data.

- If you don’t specify a type, numpy will assign one automatically.

```
import numpy as np
x = np.array([1, 2])
 # Prints "int64"
print(x.dtype)

x = np.array([1.0, 2.0])
# Prints "float64"
print(x.dtype)

x = np.array([1, 2], dtype=np.uint8)
 # Prints "uint8"
print(x.dtype)
```

- A wide variety of numerical types are available.

- Proper assignment of data types can sometimes have a significant effect on memory usage and performance.

Numpy operators

- Numpy arrays will do element-wise
  - arithmetic:+ / - * **

```
import numpy as np
x = np.array([1, 2])

x = x + 1
print(x)

y=x / 2.5

print(y.dtype)
print(y)

print(y * x)
print('Dot product: %s' % y.dot(x))
```

- Matrix (or vector/matrix, etc.) multiplicatio

- Numpy has its own sin(), cos(), log(), etc. functions that will operate element- by-element on its arrays.

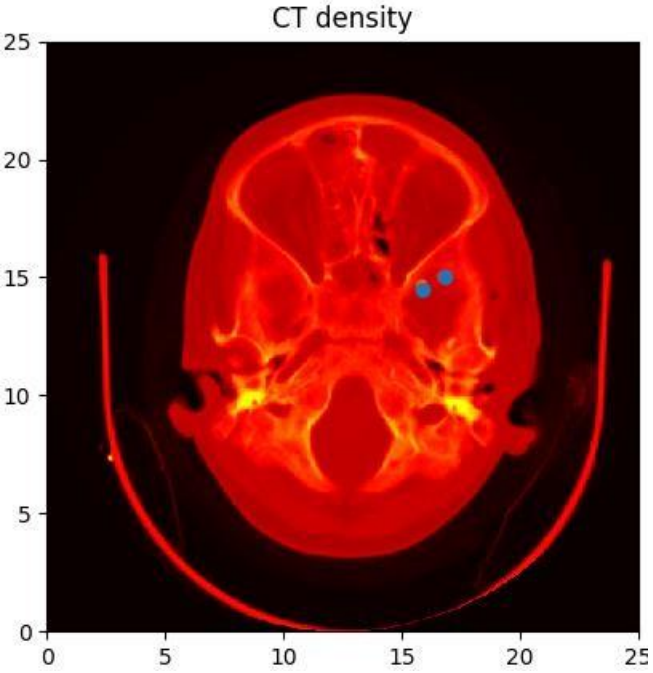
Try these out!

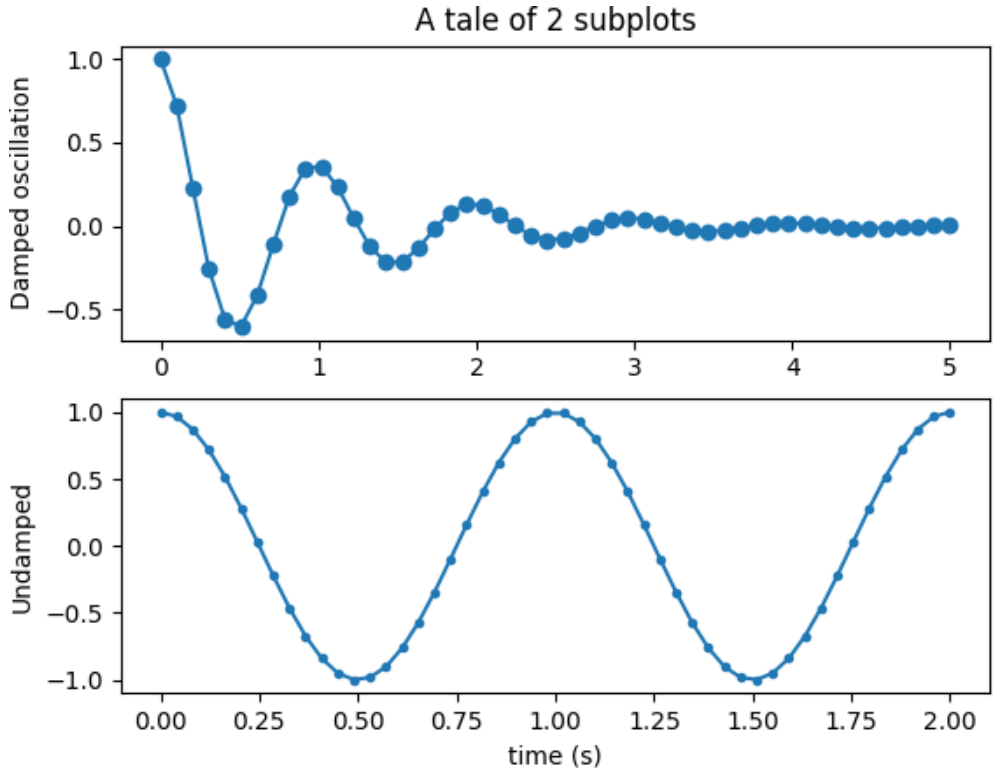
Plotting with matplotlib

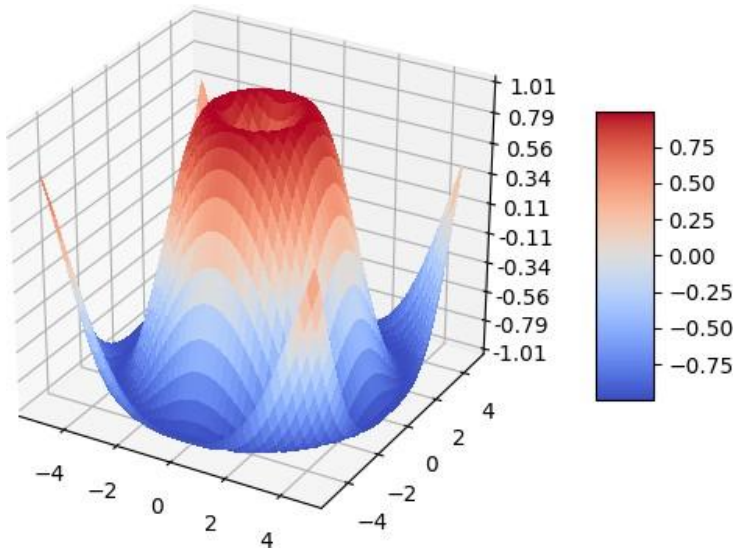
- Matplotlib is the most popular Python plotting library
  - Seaborn is another.
- Based on Matlab plotting.
- Plots can be made from lists, tuples, numpy arrays, etc.

```
import matplotlib.pyplot as plt
plt.plot([5,6,7,8])
plt.show()

import numpy as np
plt.plot(np.arange(5)+3, np.arange(5) / 10.1)
plt.show()
```

Try these out!







- Some sample images from matplotlib.org
- A vast array of plot types in 2D and 3D are available in this library.

A numpy and matplotlib example

- *numpy_matplotlib_fft.py* is a short example on using numpy and matplotlib together.
- Open *numpy_matplotlib_fft.py*
- This sample extracts signals from a noisy background.

Writing Quality Pythonic Code

- Cultivating good coding habits pays off in many ways:
  - Easier and faster to write
  - Easier and faster to edit, change, and update your code
  - Other people can understand your work

- Python lends itself to readable code
  - It's quite hard to write **completely** obfuscated code in Python.
    - Exploit language features where it makes sense
  - Contrast that with this sample of obfuscated C code.

- Here we'll go over some suggestions on how to setup a Python script, make it readable, reusable, and testable.

Compare some Python scripts

- Open up three files and let's look at them.

- A file that does…something…
  - *bad_code.py*
- Same code, re-organized:
  - *good_code.py*
- Same code, debugged, with testing code:
  - *good_code_testing.py*

Command line arguments

- Try to avoid hard-coding file paths, problem size ranges, etc. into your program.
- They can be specified at the command line.
- Look at the *argparse* module, part of the Python Standard Library.

```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max)')

args = parser.parse_args()
print(args.accumulate(args.integers))
```

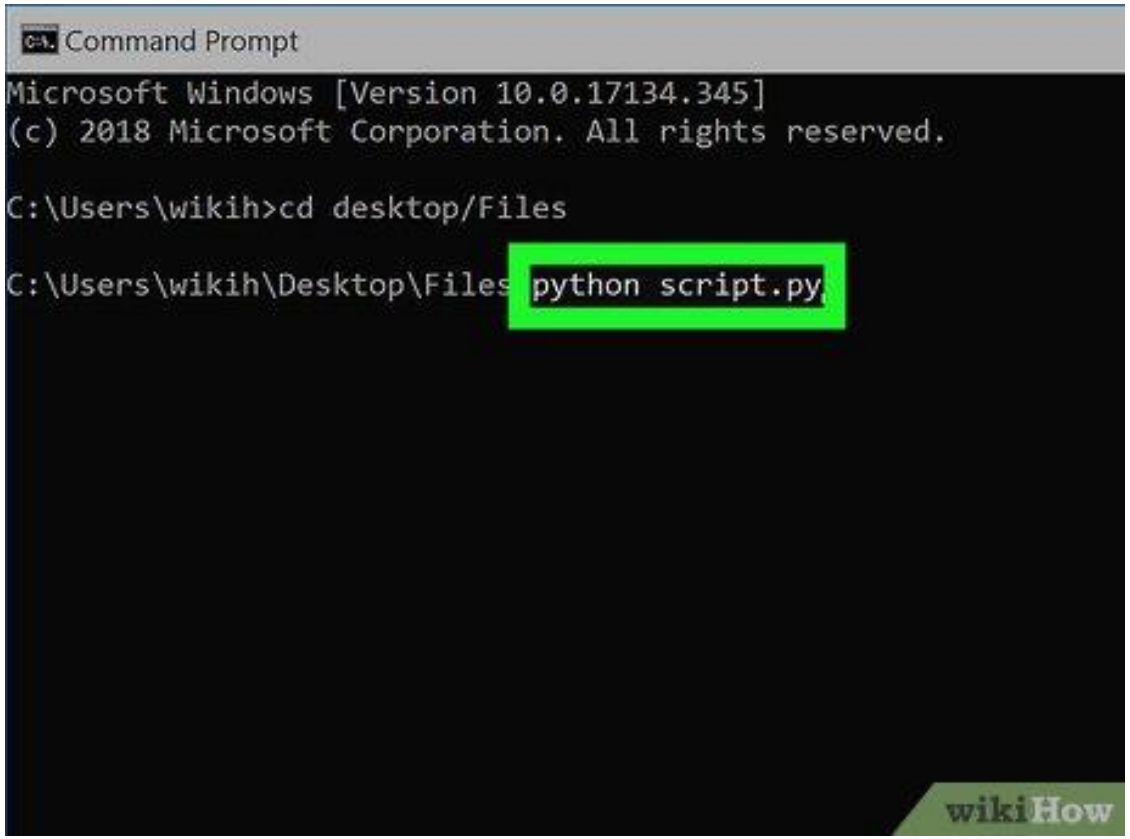
```
$ python prog.py -h
usage: prog.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:
 N           an integer for the accumulator

optional arguments:
 -h, --help  show this help message and exit
 --sum       sum the integers (default: find the max)
```

Function, class, and variable naming

- There’s no word or character limit for names.
- It’s ok to use descriptive names for things.
- An IDE (like PyCharm , VSCode) will help you fill in longer names so there’s no extra typing anyway.
- Give your functions and variables names that reflect their meaning.
  - Once a program is finished it’s easy to forget what does what where

Python from the command line

- To run Python from the command line:
- After a Python module is loaded just type *python* followed by the script name followed by script arguments.



Where to get help…

- The official Python Tutorial
- Automate the Boring Stuff with Python
  - Focuses more on doing useful things with Python, not focused on scientific computing
- Full Speed Python tutorial
- CodeWithHarry Full YouTube tutorial