**Numbers**

1.) Types of Numbers in Python

2.) Basic Arithmetic

3.) Differences between classic division and floor division

4.) Object Assignment in Python

**Types of numbers**

Python has various "types" of numbers (numeric literals). We'll mainly focus on integers and floating point numbers.

Integers are just whole numbers, positive or negative. For example: 2 and -2 are examples of integers.

Floating point numbers in Python are notable because they have a decimal point in them, or use an exponential (e) to define the number. For example 2.0 and -2.1 are examples of floating point numbers. 4E2 (4 times 10 to the power of 2) is also an example of a floating point number in Python.

**Rules for variable names**

•

The names you use when creating these labels need to follow a few rules:

1. Names can not start with a number.

2. There can be no spaces in the name, use \_ instead.

3. Can't use any of these symbols :'",<>/?|\()!@#$%^&\*~-+

4. It's considered best practice (PEP8) that names are lowercase.

5. Avoid using the characters 'l' (lowercase letter el), 'O' (uppercase letter oh),

or 'I' (uppercase letter eye) as single character variable names.

6. Avoid using words that have special meaning in Python like "list" and "str"

**Dynamic Typing**

Python uses *dynamic typing*, meaning you can reassign variables to different data types. This makes Python very flexible in assigning data types; it differs from other languages that are *statically typed*.

**Eg :**

my\_dogs **=** 2

print(my\_dogs)

Output:2

my\_dogs **=** ['Sammy', 'Frankie']

print(my\_dogs)

Output: ['Sammy', 'Frankie']

**Pros and Cons of Dynamic Typing**

Pros of Dynamic Typing

• very easy to work with

• faster development time

Cons of Dynamic Typing

• may result in unexpected bugs!

• you need to be aware of type()

**Assigning Variables**

Variable assignment follows name = object, where a single equals sign = is an *assignment operator*

Eg :

a **=** 5

print(a)

Output:5

**Reassigning Variables**

Python lets you reassign variables with a reference to the same object.

Eg :

a **=** a **+** 10

print(a)

Output:20

There's actually a shortcut for this. Python lets you add, subtract, multiply and divide numbers with reassignment using +=, -=, \*=, and /=.

Eg:

a **+=** 10

print(a)

Output:30

**Determining variable type with type()**

You can check what type of object is assigned to a variable using Python's built-in type() function. Common data types include:

**• int** (for integer)

**• float**

**• str** (for string)

**• list**

**• tuple**

**• dict** (for dictionary)

**• set**

**• bool** (for Boolean True/False)

Eg :

a=10

b=type(a)

Output:int

a **=** (1,2)

b=type(a)

print(b)

Output:tuple

**Simple Exercise**

**Eg :**

my\_income **=** 100

tax\_rate **=** 0.1

my\_taxes **=** my\_income **\*** tax\_rate

**print(**my\_taxes)

Eg:

10.0

**Strings**

Strings are used in Python to record text information, such as

names. Strings in Python are actually a *sequence*, which basically means Python keeps track of every element in the string as a sequence. For example, Python understands the string "hello' to be a sequence of letters in a specific order. This means we will be able to use indexing to grab particular letters (like the first letter, or the last letter).

1.) Creating Strings

2.) Printing Strings

3.) String Indexing and Slicing

4.) String Properties

5.) String Methods

6.) Print Formatting

**Creating a String**

To create a string in Python you need to use either single quotes or double quotes. For example:

Eg:

b=“hello”

print(b)

Output:hello

v='This is also a string'

print(v)

Output:This is also a string

**Printing a String**

print('Hello World 1')

print('Hello World 2')

print('Use \n to print a new line')

print('\n')

print('See what I mean?')

Output:

Hello World 1

Hello World 2

Use

to print a new line

See what I mean?

**String Basics**

We can also use a function called len() to check the length of a string!

**Eg:**

len('Hello World')

**Output:11**

Python's built-in len() function counts all of the characters in the string, including spaces and punctuation

**String Indexing**

We know strings are a sequence, which means Python can use indexes to call parts of the sequence

In Python, we use brackets [] after an object to call its index. We should also note that indexing starts at 0 for Python. Let's create a new object called s and then walk through a few examples of indexing.

Eg:

s **=** 'Hello World'

print(s)

first=s[0]

print(first)

second=s[1]

print(second)

We can also use index and slice notation to grab elements of a sequence by a specified step size (the default is 1). For instance we can use two colons in a row and then a number specifying the frequency to grab elements. For example:

print(s[::1])

Output:Hello World

print(s[1:5])

Output:ello

**String Properties**

It's important to note that strings have an important property known as *immutability*. This means that once a string is created, the elements within it can not be changed or replaced. For example:

Eg:

s=’Hello World'

print(s)

Output:

'Hello World'

Eg:

s=’Hello World'

s[0] **=** 'x' ##error occurs

b=s **+** ' concatenate me!'

Print(b)

Output:

'Hello World concatenate me!'

Eg:

letter **=** 'z'

print(letter**\***10)

Output:'zzzzzzzzzz'

**Basic Built-in String methods**

Objects in Python usually have built-in methods. These methods are functions inside the object that can perform actions or commands on the object itself.

We call methods with a period and then the method name. Methods are in the form:

object.method(parameters)

Eg:s='Hello World concatenate me!'

u=s.upper()

print(u)

l=s**.**lower()

print(l)

d=s**.**split()

print(d)

f=s**.**split('W')

print(f)

Out:['Hello ', 'orld concatenate me!']

**String Formatting**

String Formatting

String formatting lets you inject items into a string rather than trying to chain items together using commas or string concatenation. As a quick comparison, consider:

player = 'Thomas'

points = 33

'Last night, '+player+' scored '+str(points)+' points.' # concatenation

f'Last night, {player} scored {points} points.' # string formatting

There are three ways to perform string formatting.

• The oldest method involves placeholders using the modulo % character.

• An improved technique uses the .format() string method.

• The newest method, introduced with Python 3.6, uses formatted string literals, called *f-strings*.

**Print Formatting**

We can use the .format() method to add formatted objects to printed string statements.

'Insert another string with curly brackets: {}'**.**format('The inserted string')

'Insert another string with curly brackets: The inserted string'

**Formatting with placeholders**

You can use %s to inject strings into your print statements. The modulo % is referred to as a "string formatting operator".

In:

print("I'm going to inject %s here." **%**'something')

Out:

I'm going to inject something here.

You can pass multiple items by placing them inside a tuple after the % operator.

in: print("I'm going to inject %s text here, and %s text here." **%**('some','more'))

Out:

I'm going to inject some text here, and more text here.

You can also pass variable names:

Eg:

x, y **=** 'some', 'more'

print("I'm going to inject %s text here, and %s text here."**%**(x,y))

Out:I'm going to inject some text here, and more text here.

**Format conversion methods.**

It should be noted that two methods %s and %r convert any python object to a string using two separate methods: str() and repr(). We will learn more about these functions later on in the course, but you should note that %r and repr() deliver the *string representation* of the object, including quotation marks and any escape characters.

Eg:

print('He said his name was %s.' **%**'Fred')

print('He said his name was %r.' **%**'Fred')

**Out:**

He said his name was Fred.

He said his name was 'Fred'.

As another example, \t inserts a tab into a string.

int:print('I once caught a fish %s.' **%**'this \tbig')

print('I once caught a fish %r.' **%**'this \tbig')

Eg:print('I once caught a fish %s.' **%**'this \tbig')

print('I once caught a fish %r.' **%**'this \tbig')

Out:I once caught a fish this big.

I once caught a fish 'this \tbig'.

The %s operator converts whatever it sees into a string, including integers and floats. The %d operator converts numbers to integers first, without rounding. Note the difference below:

Int:print('I wrote %s programs today.' **%3**.75)

print('I wrote %d programs today.' **%3**.75)

Out:

I wrote 3.75 programs today.

I wrote 3 programs today.

**Padding and Precision of Floating Point Numbers**

Floating point numbers use the format %5.2f. Here, 5 would be the minimum number of characters the string should contain; these may be padded with whitespace if the entire number does not have this many digits. Next to this, .2f stands for how many numbers to show past the decimal point. Let's see some examples:

Inp:print('Floating point numbers: %5.2f' **%**(13.144))

Out:Floating point numbers: 13.14

Inp:print('Floating point numbers: %1.0f' **%**(13.144))

Out:Floating point numbers: 13

Inp:print('Floating point numbers: %1.5f' **%**(13.144))

Out:print('Floating point numbers: %1.5f' **%**(13.144))

**Multiple Formatting**

Nothing prohibits using more than one conversion tool in the same print statement:

**Inp:**print('First: %s, Second: %5.2f, Third: %r' **%**('hi!',3.1415,'bye!'))

**Out:**First: hi!, Second: 3.14, Third: 'bye!'

**Formatting with the .format() method**

A better way to format objects into your strings for print statements is with the string .format() method. The syntax is:

'String here {} then also {}'.format('something1','something2')

For example:

Inp:print('This is a string with an {}'**.**format('insert'))

Out:This is a string with an insert

**The .format() method has several advantages over the %s placeholder method:**

1. Inserted objects can be called by index position:

Inp:print('The {2} {1} {0}'**.**format('fox','brown','quick'))

Out:The quick brown fox

2. Inserted objects can be assigned keywords:

inp:print('First Object: {a}, Second Object: {b}, Third Object: {c}'**.**format(a**=**1,b**=**'Two',c**=**12.3))

Out:First Object: 1, Second Object: Two, Third Object: 12.3

3. Inserted objects can be reused, avoiding duplication:

inp:print('A %s saved is a %s earned.' **%**('penny','penny'))

print('A {p} saved is a {p} earned.'**.**format(p**=**'penny'))

out:A penny saved is a penny earned.

A penny saved is a penny earned.

**Alignment, padding and precision with .format()**

Within the curly braces you can assign field lengths, left/right alignments, rounding parameters and more

inp:print('{0:8} | {1:9}'**.**format('Fruit', 'Quantity'))

print('{0:8} | {1:9}'**.**format('Apples', 3.))

print('{0:8} | {1:9}'**.**format('Oranges', 10))

Output:

Fruit | Quantity

Apples | 3.0

Oranges | 10

By default, .format() aligns text to the left, numbers to the right. You can pass an optional <,^, or > to set a left, center or right alignment:

inp:print('{0:<8} | {1:^8} | {2:>8}'**.**format('Left','Center','Right'))

print('{0:<8} | {1:^8} | {2:>8}'**.**format(11,22,33))

out:Left | Center | Right

11 | 22 | 33

You can precede the aligment operator with a padding character

Inp:

print('{0:=<8} | {1:-^8} | {2:.>8}'**.**format('Left','Center','Right'))

print('{0:=<8} | {1:-^8} | {2:.>8}'**.**format(11,22,33))

out:Left==== | -Center- | ...Right

11====== | ---22--- | ......33

Field widths and float precision are handled in a way similar to placeholders. The following two print statements are equivalent:

Inp: print('This is my ten-character, two-decimal number:%10.2f' **%13**.579)

print('This is my ten-character, two-decimal number:{0:10.2f}'**.**format(13.579))

Out:

This is my ten-character, two-decimal number: 13.58

This is my ten-character, two-decimal number: 13.58

Note that there are 5 spaces following the colon, and 5 characters taken up by 13.58, for a total of ten characters.

**Formatted String Literals (f-strings)**

Introduced in Python 3.6, f-strings offer several benefits over the older .format() string method described above. For one, you can bring outside variables immediately into to the string rather than pass them as arguments through .format(var).

inp:name **=** 'Fred'

print(f"He said his name is {name}.")

Out:He said his name is Fred.

**Pass !r to get the string representation:**

Inp:print(f"He said his name is {name!r}")

Out:

He said his name is 'Fred'

**Float formatting follows "result: {value:{width}.{precision}}"**

Where with the .format() method you might see {value:10.4f}, with f-strings this can become {value:{10}.{6}}

Eg:

Inp:

num **=** 23.45678

print("My 10 character, four decimal number is:{0:10.4f}"**.**format(num))

print(f"My 10 character, four decimal number is:{num:{10}.{6}}")

**Out:**

My 10 character, four decimal number is: 23.4568

My 10 character, four decimal number is: 23.4568

**Lists**

 Lists can be thought of the most general version of a *sequence* in Python. Unlike strings, they are mutable, meaning the elements inside a list can be changed!.Lists are constructed with brackets [] and commas separating every element in the list.

1.) Creating lists

2.) Indexing and Slicing Lists

3.) Basic List Methods

4.) Nesting Lists

5.) Introduction to List Comprehensions

Eg:my\_list **=** [1,2,3]

my\_list **=** ['A string',23,100.232,'o']

Just like strings, the len() function will tell you how many items are in the sequence of the list.

Inp:len(my\_list)

Out:4

**Indexing and Slicing**

Indexing and slicing work just like in strings. Let's make a new list to remind ourselves of how this works:

Eg:

my\_list **=** ['one','two','three',4,5]

print(my\_list[0])

Out:'one'

inp:

print(my\_list[1:])

Out:['two', 'three', 4, 5]

inp:print(my\_list[:3])

Out:

['one', 'two', 'three']

We can also use + to concatenate lists, just like we did for strings.

Eg:print(my\_list **+** ['new item’])

Out:['one', 'two', 'three', 4, 5, 'new item']

Note: This doesn't actually change the original list!

print(my\_list)

['one', 'two', 'three', 4, 5]

You would have to reassign the list to make the change permanent.

Inp:

my\_list **=** my\_list **+** ['add new item permanently']

print(my\_list)

out:[‘one', 'two', 'three', 4, 5, 'add new item permanently']

We can also use the \* for a duplication method similar to strings:

eg: my\_list **\*** 2

Out:

['one',

'two',

'three',

4,

5,

'add new item permanently',

'one',

'two',

'three',

4,

5,

'add new item permanently']

**Basic List Methods**

Lists in Python however, tend to be more flexible than arrays in other languages for a two good reasons: they have no fixed size (meaning we don't have to specify how big a list will be), and they have no fixed type constraint (like we've seen above).

Eg:

list1 **=** [1,2,3]

Use the **append** method to permanently add an item to the end of a list:

list1**.**append('append me!')

print(list1)

Out:

[1, 2, 3, 'append me!']

Use **pop** to "pop off" an item from the list. By default pop takes off the last index, but you can also specify which index to pop off. Let's see an example:

inp:print(list1**.**pop(0))

Out:1

print(list1)

Out:

[2, 3, 'append me!']

popped\_item **=** list1**.**pop()

print(popped\_item)

Out:'append me!'

print(list1)

Out:[2, 3]

We can use the **sort** method and the **reverse** methods to also effect your lists:

new\_list **=** ['a','e','x','b','c']

print(new\_list)

Out:

['a', 'e', 'x', 'b', 'c']

*# Use reverse to reverse order (this is permanent!)*

Inp:

new\_list**.**reverse()

print(new\_list)

Out:['c', 'b', 'x', 'e', 'a']

*# Use sort to sort the list (in this case alphabetical order, but for numbers it will go ascending)*

new\_list**.**sort()

print(new\_list)

Output:['a', 'b', 'c', 'e', 'x']

**Nesting Lists**

A great feature of of Python data structures is that they support *nesting*. This means we can have data structures within data structures. For example: A list inside a list.

inp:

*# Let's make three lists*

lst\_1**=**[1,2,3]

lst\_2**=**[4,5,6]

lst\_3**=**[7,8,9]

*# Make a list of lists to form a matrix*

matrix **=** [lst\_1,lst\_2,lst\_3]

print(matrix)

Out:[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

We can again use indexing to grab elements, but now there are two levels for the index. The items in the matrix object, and then the items inside that list!

*# Grab first item in matrix object*

inp:print(matrix[0])

Out:[1, 2, 3]

*# Grab first item of the first item in the matrix object*

eg:print(matrix[0][0])

Out:1

**List Comprehensions**

Python has an advanced feature called list comprehensions. They allow for quick construction of lists. To fully understand list comprehensions we need to understand for loops.

*# Build a list comprehension by deconstructing a for loop within a []*

first\_col **=** [row[0] **for** row **in** matrix]

print**(**first\_col)

Out:[1, 4, 7]

We used a list comprehension here to grab the first element of every row in the matrix object.

**Dictionaries**

1.) Constructing a Dictionary

2.) Accessing objects from a dictionary

3.) Nesting Dictionaries

4.) Basic Dictionary Methods

Mappings are a collection of objects that are stored by a *key*, unlike a sequence that stored objects by their relative position. This is an important distinction, since mappings won't retain order since they have objects defined by a key.

A Python dictionary consists of a key and then an associated value. That value can be almost any Python object.

**Constructing a Dictionary**

*# Make a dictionary with {} and : to signify a key and a value*

my\_dict **=** {'key1':'value1','key2':'value2'}

*# Call values by their key*

print(my\_dict['key2’])

Out:'value2'

Its important to note that dictionaries are very flexible in the data types they can hold. For example:

my\_dict **=** {'key1':123,'key2':[12,23,33],'key3':['item0','item1','item2']}

*# Let's call items from the dictionary*

my\_dict['key3']

Out:['item0', 'item1', 'item2']

*# Can call an index on that value*

print(my\_dict['key3’][0])

Out:'item0'

*# Can then even call methods on that value*

my\_dict['key3'][0]**.**upper()

Out:'ITEM0'

We can affect the values of a key as well. For instance:

print(my\_dict['key1’])

Out: 123

*# Subtract 123 from the value*

my\_dict['key1'] **=** my\_dict['key1'] **-** 123

*#Check*

print(my\_dict['key1’])

Out:

0

A quick note, Python has a built-in method of doing a self subtraction or addition (or multiplication or division). We could have also used += or -= for the above statement. For example:

*# Set the object equal to itself minus 123*

my\_dict['key1'] **-=** 123

my\_dict['key1']

Out:-123

We can also create keys by assignment. For instance if we started off with an empty dictionary, we could continually add to it:

*# Create a new dictionary*

d **=** {}

*# Create a new key through assignment*

d['animal'] **=** 'Dog'

*# Can do this with any object*

d['answer'] **=** 42

print(d)

Output:{'animal': 'Dog', 'answer': 42}

**Nesting with Dictionaries**

Hopefully you're starting to see how powerful Python is with its flexibility of nesting objects and calling methods on them. Let's see a dictionary nested inside a dictionary:

*# Dictionary nested inside a dictionary nested inside a dictionary*

d **=** {'key1':{'nestkey':{'subnestkey':'value'}}}

Wow! That's a quite the inception of dictionaries! Let's see how we can grab that value:

print(*# Keep calling the keys*

d['key1']['nestkey']['subnestkey’])

Out:

'value'

**A few Dictionary Methods**

*# Create a typical dictionary*

d **=** {'key1':1,'key2':2,'key3':3}

*# Method to return a list of all keys*

print(d**.**keys())

Out:dict\_keys(['key1', 'key2', 'key3'])

*# Method to grab all values*

print(d**.**values())

Out:dict\_values([1, 2, 3])

*# Method to return tuples of all items (we'll learn about tuples soon)*

print(d**.**items())

Out:

dict\_items([('key1', 1), ('key2', 2), ('key3', 3)])

**Tuples**

In Python tuples are very similar to lists, however, unlike lists they are *immutable* meaning they can not be changed. You would use tuples to present things that shouldn't be changed, such as days of the week, or dates on a calendar.

1.) Constructing Tuples

2.) Basic Tuple Methods

3.) Immutability

4.) When to Use Tuples

**Constructing Tuples**

The construction of a tuples use () with elements separated by commas. For example:

*# Create a tuple*

t **=** (1,2,3)

*# Check len just like a list*

len(t)

Out:3

*# Can also mix object types*

t **=** ('one',2)

*# Show*

print(t)

Out:('one', 2)

*# Use indexing just like we did in lists*

print(t[0])

Out:'one'

*# Slicing just like a list*

print(t[**-**1])

Out:2

**Basic Tuple Methods**

Tuples have built-in methods, but not as many as lists do. Let's look at two of them:

*# Use .index to enter a value and return the index*

print(t**.**index('one’))

Out:0

*# Use .count to count the number of times a value appears*

print(t**.**count('one’))

Out:1

**Set and Booleans**

**Sets**

Sets are an unordered collection of *unique* elements. We can construct them by using the set() function.

x **=** set()

*# We add to sets with the add() method*

x**.**add(1)

print(x)

Out:

{1}

*# Add a different element*

x**.**add(2)

print(x)

Out:

{1, 2}

*# Try to add the same element*

x**.**add(1)

print(x)

Out:{1, 2}

We can cast a list with multiple repeat elements to a set to get the unique elements. For example:

*# Create a list with repeats*

list1 **=** [1,1,2,2,3,4,5,6,1,1]

*# Cast as set to get unique values*

print(set(list1))

Out:{1, 2, 3, 4, 5, 6}

**Booleans**

Python comes with Booleans (with predefined True and False displays that are basically just the integers 1 and 0).It also has a placeholder object called None.

*# Set object to be a boolean*

a **=** **True**

print(a)

Out:true

We can also use comparison operators to create booleans

*# Output is boolean*

1 **>** 2

Out:False

We can use None as a placeholder for an object that we don't want to reassign yet:

*# None placeholder*

b **=** **None**

print(b)

Out:none

**Files**

Python uses file objects to interact with external files on your computer. These file objects can be any sort of file you have on your computer, whether it be an audio file, a text file, emails, Excel documents, etc. Note: You will probably need to install certain libraries or modules to interact with those various file types, but they are easily available

Python has a built-in open function that allows us to open and play with basic file types. First we will need a file though.

**Python Opening a file**

myfile = open("/Users/YouUserName/Folder/myfile.txt")

*# Open the text.txt we made earlier*

my\_file **=** open('test.txt')

*# We can now read the file*

my\_file**.**read()

Output:'Hello, this is a quick test file.'

*# But what happens if we try to read it again?*

print(my\_file**.**read())

Output:''

This happens because you can imagine the reading "cursor" is at the end of the file after having read it. So there is nothing left to read. We can reset the "cursor" like this:

*# Seek to the start of file (index 0)*

print(my\_file**.**seek(0))

Output:0

*# Now read again*

print(my\_file**.**read())

Output:

'Hello, this is a quick test file.'

You can read a file line by line using the readlines method. Use caution with large files, since everything will be held in memory

*# Readlines returns a list of the lines in the file*

my\_file**.**seek(0)

my\_file**.**readlines()

Out:['Hello, this is a quick test file.']

When you have finished using a file, it is always good practice to close it.

print(my\_file**.**close())

**Writing to a File**

By default, the open() function will only allow us to read the file. We need to pass the argument 'w' to write over the file. For example:

*# Add a second argument to the function, 'w' which stands for write.*

*# Passing 'w+' lets us read and write to the file*

my\_file **=** open('test.txt','w+')

*# Write to the file*

my\_file**.**write('This is a new line')

Out:18

*# Read the file*

print(my\_file**.**seek(0))

print(my\_file**.**read())

Out:'This is a new line'

my\_file**.**close() *# always do this when you're done with a file*

**Appending to a File**

Passing the argument 'a' opens the file and puts the pointer at the end, so anything written is appended. Like 'w+', 'a+' lets us read and write to a file. If the file does not exist, one will be created.

my\_file **=** open('test.txt','a+')

my\_file**.**write('\nThis is text being appended to test.txt')

my\_file**.**write('\nAnd another line here.')

Out:23

my\_file**.**seek(0)

print(my\_file**.**read())

Out:This is a new line

This is text being appended to test.txt

And another line here.

my\_file**.**close()

**Iterating through a File**

**for** line **in** open('test.txt'):

print(line)

Output :First Line

Second Line

It's important to note a few things here:

1 We could have called the "line" object anything (see example below).

2 By not calling .read() on the file, the whole text file was not stored in memory.

3 Notice the indent on the second line for print. This whitespace is required in Python.

*# Pertaining to the first point above*

**for** asdf **in** open('test.txt'):

print(asdf)

Output:

First Line

Second Line

**Comparison Operators**

These operators will allow us to compare variables and output a Boolean value (True or False).

Table of Comparison Operators

In the table below, a=3 and b=4.

**Operator**

**Description**

**Example**

==

If the values of two operands are equal, then the condition becomes true.

(a == b) is not true.

!=

If values of two operands are not equal, then condition becomes true.

(a != b) is true

>

If the value of left operand is greater than the value of right operand, then condition becomes true.

(a > b) is not true.

<

If the value of left operand is less than the value of right operand, then condition becomes true.

(a < b) is true.

>=

If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.

(a >= b) is not true.

<=

If the value of left operand is less than or equal to the value of right operand, then condition becomes true.

(a <= b) is true.

Eg:

Equal

Inp:2 **==** 2

Out: True

Not Equal

Inp: 2 **!=** 1

Out: True

Greater than

Inp:2 **>** 1

Out: True

Less Than

Inp:2 **<** 4

Out: True

Greater Than or Equal to

Inp:2 **>=** 2

Out: True

Less than or Equal to

Inp:2 **<=** 2

Out: True

**Chained Comparison Operators**

An interesting feature of Python is the ability to *chain* multiple comparisons to perform a more complex test. You can use these chained comparisons as shorthand for larger Boolean Expressions.

Eg:

Inp:1 **<** 2 **<** 3

Out: True

Inp:1**<**2 **and** 2**<**3

Out: True

Inp:1**==**2 **or** 2**<**3

Out: True

**Introduction to Python Statements**

**if, elif, else Statements**

if Statements in Python allows us to tell the computer to perform alternative actions based on a certain set of results.

if this case happens, perform some action. Else, if another case happens, perform some other action. Else, if *none* of the above cases happened, perform this action.

Syntax:

if case1:

perform action1

elif case2:

perform action2

else:

perform action3

**Eg:**

**Inp:**

x **=** **False**

**if** x:

print('x was True!')

**else**:

print('I will be printed in any case where x is not true')

Out:

I will be printed in any case where x is not true

Multiple Branches

Eg:

Inp:

loc **=** 'Bank'

**if** loc **==** 'Auto Shop':

print('Welcome to the Auto Shop!')

**elif** loc **==** 'Bank':

print('Welcome to the bank!')

**else**:

print('Where are you?')

Out:

Welcome to the bank!

**Indentation**

It is important to keep a good understanding of how indentation works in Python to maintain the structure and order of your code.

**for Loops**

A for loop acts as an iterator in Python; it goes through items that are in a *sequence* or any other iterable item. Objects that we've learned about that we can iterate over include strings, lists, tuples, and even built-in iterables for dictionaries, such as keys or values.

Here's the general format for a for loop in Python:

for item in object:

statements to do stuff

**Example 1**

Iterating through a list

Inp:

*# We'll learn how to automate this sort of list in the next lecture*

list1 **=** [1,2,3,4,5,6,7,8,9,10]

**for** num **in** list1:

print(num)

Out:

1

2

3

4

5

6

7

8

9

10

**Modulo**

The modulo allows us to get the remainder in a division and uses the % symbol. For example:

Inp:17 **%** 5

Out:2

This makes sense since 17 divided by 5 is 3 remainder 2.

Eg:

print only the even numbers from that list!

Inp:

**for** num **in** list1:

**if** num **%** 2 **==** 0:

print(num)

Out:

2

4

6

8

10

Eg:

create a for loop that sums up the list:

list1 = [1,2,3,4,5,6,7,8,9,10]

*# Start sum at zero*

list\_sum = 0

for num in list1:

list\_sum = list\_sum + num

print(list\_sum)

Out:55

**Or**

*# Start sum at zero*

list\_sum **=** 0

**for** num **in** list1:

list\_sum **+=** num

print(list\_sum)

**Out:55**

**Eg:**

Remember strings are a sequence so when we iterate through them we will be accessing each item in that string.

**for** letter **in** 'This is a string.':

print(letter)

Out:

T

h

i

s

i

s

a

s

t

r

i

n

g

.

Eg:

how a for loop can be used with a tuple

tup **=** (1,2,3,4,5)

**for** t **in** tup:

print(t)

Out:

1

2

3

4

5

Eg:

Tuples have a special quality when it comes to for loops. If you are iterating through a sequence that contains tuples, the item can actually be the tuple itself, this is an example of *tuple unpacking*. During the for loop we will be unpacking the tuple inside of a sequence and we can access the individual items inside that tuple!

list2 **=** [(2,4),(6,8),(10,12)]

**for** tup **in** list2:

print(tup)

Out:

(2, 4)

(6, 8)

(10, 12)

*# Now with unpacking!*

**for** (t1,t2) **in** list2:

print(t1)

Out:

2

6

10

Eg:

In dictionary

d **=** {'k1':1,'k2':2,'k3':3}

**for** item **in** d:

print(item)

Out:

k1

k2

k3

We're going to introduce three new Dictionary methods: **.keys()**, **.values()** and **.items()**

*# Create a dictionary view object*

print(d**.**items())

Out:

dict\_items([('k1', 1), ('k2', 2), ('k3', 3)])

*# Dictionary unpacking*

**for** k,v **in** d**.**items():

print(k)

print(v)

Out:

k1

1

k2

2

k3

3

Inp: print(list(d**.**keys()))

Out: ['k1', 'k2', 'k3']

Inp: sorted(d**.**values())

Out:[1, 2, 3]

**while** **Loops**

The while statement in Python is one of most general ways to perform iteration. A while statement will repeatedly execute a single statement or group of statements as long as the condition is true. The reason it is called a 'loop' is because the code statements are looped through over and over again until the condition is no longer met.

The general format of a while loop is:

while test:

code statements

else:

final code statements

Eg:

x **=** 0

**while** x **<** 10:

print('x is currently: ',x)

print(' x is still less than 10, adding 1 to x')

x**+=**1

Out:

x is currently: 0

x is still less than 10, adding 1 to x

x is currently: 1

x is still less than 10, adding 1 to x

x is currently: 2

x is still less than 10, adding 1 to x

x is currently: 3

x is still less than 10, adding 1 to x

x is currently: 4

x is still less than 10, adding 1 to x

x is currently: 5

x is still less than 10, adding 1 to x

x is currently: 6

x is still less than 10, adding 1 to x

x is currently: 7

x is still less than 10, adding 1 to x

x is currently: 8

x is still less than 10, adding 1 to x

x is currently: 9

x is still less than 10, adding 1 to x

**break, continue, pass**

We can use break, continue, and pass statements in our loops to add additional functionality for various cases. The three statements are defined by:

break: Breaks out of the current closest enclosing loop.

continue: Goes to the top of the closest enclosing loop.

pass: Does nothing at all.

Thinking about break and continue statements, the general format of the while loop looks like this:

while test:

code statement

if test:

break

if test:

continue

else:

break and continue statements can appear anywhere inside the loop’s body, but we will usually put them further nested in conjunction with an if statement to perform an action based on some condition.

Eg:

x **=** 0

**while** x **<** 10:

print('x is currently: ',x)

print(' x is still less than 10, adding 1 to x')

x**+=**1

**if** x**==**3:

print('x==3')

**else**:

print('continuing...')

**continue**

Out:

x is currently: 0

x is still less than 10, adding 1 to x

continuing...

x is currently: 1

x is still less than 10, adding 1 to x

continuing...

x is currently: 2

x is still less than 10, adding 1 to x

x==3

x is currently: 3

x is still less than 10, adding 1 to x

continuing...

x is currently: 4

x is still less than 10, adding 1 to x

continuing...

x is currently: 5

x is still less than 10, adding 1 to x

continuing...

x is currently: 6

x is still less than 10, adding 1 to x

continuing...

x is currently: 7

x is still less than 10, adding 1 to x

continuing...

x is currently: 8

x is still less than 10, adding 1 to x

continuing...

x is currently: 9

x is still less than 10, adding 1 to x

continuing...

Eg1:

x **=** 0

**while** x **<** 10:

print('x is currently: ',x)

print(' x is still less than 10, adding 1 to x')

x**+=**1

**if** x**==**3:

print('Breaking because x==3')

**break**

**else**:

print('continuing...')

**continue**

Out:

x is currently: 0

x is still less than 10, adding 1 to x

continuing...

x is currently: 1

x is still less than 10, adding 1 to x

continuing...

x is currently: 2

x is still less than 10, adding 1 to x

Breaking because x==3

**range**

The range function allows you to quickly *generate* a list of integers, this comes in handy a lot, so take note of how to use it! There are 3 parameters you can pass, a start, a stop, and a step size. Let's see some examples:

Inp:range(0,11)

Out:range(0, 11)

Note that this is a **generator** function, so to actually get a list out of it, we need to cast it to a list with **list()**.

What is a generator? Its a special type of function that will generate information and not need to save it to memory.

Eg:

*# Notice how 11 is not included, up to but not including 11, just like slice notation!*

list(range(0,11))

Out:[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Eg:

*# Third parameter is step size!*

*# step size just means how big of a jump/leap/step you*

*# take from the starting number to get to the next number.*

list(range(0,11,2))

Out:[0, 2, 4, 6, 8, 10]

**enumerate**

enumerate is a very useful function to use with for loops. Let's imagine the following situation:

index\_count **=** 0

**for** letter **in** 'abcde':

print("At index {} the letter is {}"**.**format(index\_count,letter))

index\_count **+=** 1

Out:

At index 0 the letter is a

At index 1 the letter is b

At index 2 the letter is c

At index 3 the letter is d

At index 4 the letter is e

Keeping track of how many loops you've gone through is so common, that enumerate was created so you don't need to worry about creating and updating this index\_count or loop\_count variable

Eg:

*# Notice the tuple unpacking!*

**for** i,letter **in** enumerate('abcde'):

print("At index {} the letter is {}"**.**format(i,letter))

Out:

At index 0 the letter is a

At index 1 the letter is b

At index 2 the letter is c

At index 3 the letter is d

At index 4 the letter is e

**zip**

list(enumerate('abcde'))

Out:

[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e')]

**zip()** function to quickly create a list of tuples by "zipping" up together two lists.

Inp:

mylist1 **=** [1,2,3,4,5]

mylist2 **=** ['a','b','c','d','e']

*# This one is also a generator! We will explain this later, but for now let's transform it to a list*

zip(mylist1,mylist2)

Out:

<zip at 0x1d205086f08>

Inp:

list(zip(mylist1,mylist2))

Out:[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]

Inp:

**for** item1, item2 **in** zip(mylist1,mylist2):

print('For this tuple, first item was {} and second item was {}'**.**format(item1,item2))

Out:

For this tuple, first item was 1 and second item was a

For this tuple, first item was 2 and second item was b

For this tuple, first item was 3 and second item was c

For this tuple, first item was 4 and second item was d

For this tuple, first item was 5 and second item was e

**in operator**

 we can also use it to quickly check if an object is in a list.

Eg:

inp:’x’ **in** ['x','y','z']

Out:True

**not in**

To check if some object or variable is not present in a list.

Eg:

'x' **not** **in** ['x','y','z']

Out:false

**min and max**

Quickly check the minimum or maximum of a list with these functions.

Inp:mylist **=** [10,20,30,40,100]

print(min(mylist))

print(max(mylist))

Out:10

100

**random**

Python comes with a built in random library. There are a lot of functions included in this random library, so we will only show you two useful functions for now.

Eg:

**from** random **import** shuffle

*# This shuffles the list "in-place" meaning it won't return*

*# anything, instead it will effect the list passed*

shuffle(mylist)

print(mylist)

Out:[40, 10, 100, 30, 20]

**from** random **import** randint

*# Return random integer in range [a, b], including both end points.*

Print(randint(0,100))

Out:25

*# Return random integer in range [a, b], including both end points.*

randint(0,100)

Out:91

**Input**

input('Enter Something into this box: ')

Out:Enter Something into this box: great job!

**List Comprehensions**

List comprehensions allow us to build out lists using a different notation

Eg:

*# Grab every letter in string*

lst **=** [x **for** x **in** 'word']

Print (lst)

**Out:**

['w', 'o', 'r', 'd']

**Eg:**

*# Square numbers in range and turn into list*

lst **=** [x**\*\***2 **for** x **in** range(0,11)]

Print(lst)

**Out:**

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

**Eg:**

*# Check for even numbers in a range*

lst **=** [x **for** x **in** range(11) **if** x **%** 2 **==** 0]

Print (lst)

**Out:**[0, 2, 4, 6, 8, 10]

**Eg:**

*# Convert Celsius to Fahrenheit*

celsius **=** [0,10,20.1,34.5]

fahrenheit **=** [((9**/**5)**\***temp **+** 32) **for** temp **in** celsius ]

Print (fahrenheit)

Out**:**[32.0, 50.0, 68.18, 94.1]

Eg**:**

lst **=** [ x**\*\***2 **for** x **in** [x**\*\***2 **for** x **in** range(11)]]

Print **(**lst**)**

**Out:**[0, 1, 16, 81, 256, 625, 1296, 2401, 4096, 6561, 10000]

Statements Assessment Test

**1.Use for, .split(), and if to create a Statement that will print out words that start with 's':**

**Inp:**st **=** 'Print only the words that start with s in this sentence'

**2.Use range() to print all the even numbers from 0 to 10.**

**3.Use a List Comprehension to create a list of all numbers between 1 and 50 that are divisible by 3.**

**3.Go through the string below and if the length of a word is even print "even!"**

st **=** 'Print every word in this sentence that has an even number of letters'

**4.Write a program that prints the integers from 1 to 100. But for multiples of three print "Fizz" instead of the number, and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".**

**5.Use List Comprehension to create a list of the first letters of every word in the string below:**

**Inp:**st **=** 'Create a list of the first letters of every word in this string'

**Answers**

**————**

1a**.**

inp**:for** word **in** st**.**split():

**if** word[0] **==** 's':

print(word)

Out:

start

s

sentence

2a.

inp:list(range(0,11,2))

Out:[0, 2, 4, 6, 8, 10]

3a.

inp:[x **for** x **in** range(1,51) **if** x**%3** == 0]

Out:[3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48]

4a.

Inp:**for** word **in** st**.**split():

**if** len(word)**%2** == 0:

print(word**+**" <-- has an even length!")

Out:

word <-- has an even length!

in <-- has an even length!

this <-- has an even length!

sentence <-- has an even length!

that <-- has an even length!

an <-- has an even length!

even <-- has an even length!

number <-- has an even length!

of <-- has an even length!

5a.

Inp:

**if** num **%** 3 **==** 0 **and** num **%** 5 **==**0:

print("FizzBuzz")

**elif** num **%** 3 **==** 0:

print("Fizz")

**elif** num **%** 5 **==** 0:

print("Buzz")

**else**:

print(num)

6a.

Inp:[word[0] **for** word **in** st**.**split()]

Out:['C', 'a', 'l', 'o', 't', 'f', 'l', 'o', 'e', 'w', 'i', 't', 's']

**Methods**

**——————**

Methods are essentially functions built into objects.

Methods perform specific actions on an object and can also take arguments, just like a function.

Methods are in the form:

object.method(arg1,arg2,etc...)

Eg:

*# Create a simple list*

lst **=** [1,2,3,4,5]

The methods for a list are:

• append

• count

• extend

• insert

• pop

• remove

• reverse

• sort

**append**() allows us to add elements to the end of a list:

Eg:

lst**.**append(6)

print(lst)

Out:[1, 2, 3, 4, 5, 6]

The **count**() method will count the number of occurrences of an element in a list

Inp:*# Check how many times 2 shows up in the list*

lst**.**count(2)

Out:1

**Functions**

**————**

Functions will be one of our main building blocks when we construct larger and larger amounts of code to solve problems.

function is a useful device that groups together a set of statements so they can be run more than once. They can also let us specify parameters that can serve as inputs to the functions.

functions allow us to not have to repeatedly write the same code again and again.

▪ def keyword

▪ simple example of a function

▪ calling a function with ()

▪ accepting parameters

▪ print versus return

▪ adding in logic inside a function

▪ multiple returns inside a function

▪ adding in loops inside a function

▪ tuple unpacking

▪ interactions between functions

**def keyword**

Inp:

**def** name\_of\_function(arg1,arg2):

'''

This is where the function's Document String (docstring) goes.

When you call help() on your function it will be printed out.

'''

*# Do stuff here*

*# Return desired result*

def then a space followed by the name of the function.

Next come a pair of parentheses with a number of arguments separated by a comma. These arguments are the inputs for your function. You'll be able to use these inputs in your function and reference them. After this you put a colon.

Python makes use of *whitespace* to organize code

Eg:**def** say\_hello():

print('hello')

**Calling a function with ()**

Inp:say\_hello()

Out:hello

**Accepting parameters (arguments)**

Eg:

**def** greeting(name):

print(f'Hello {name}')

Inp:print (greeting('Jose')

)

Out:Hello Jose

**Using return**

return allows a function to *return* a result that can then be stored as a variable, or used in whatever manner a user wants.

Eg:

**def** add\_num(num1,num2):

**return** num1**+**num2

Print (add\_num(4,5)

)

Out:9

*# Can also save as variable due to return*

result **=** add\_num(4,5)

print(result)

Out:

9

**Whatisthedifferencebetween** ***return*** **and** **print**?

The return keyword allows you to actually save the result of the output of a function as a variable. The print() function simply displays the output to you, but doesn't save it for future use**.**

**Eg:**

**def** print\_result(a,b):

print(a**+**b)

**def** return\_result(a,b):

**return** a**+**b

print\_result(10,5)

Out:15

*# You won't see any output if you run this in a .py script*

return\_result(10,5)

Out:15

my\_result **=** print\_result(20,20)

Out:40

Print (my\_result)

Print (type(my\_result))

Out: NoneType

**Check if any number in a list is even**

**Eg:**

**def** check\_even\_list(num\_list):

*# Go through each number*

**for** number **in** num\_list:

*# Once we get a "hit" on an even number, we return True*

**if** number **%** 2 **==** 0:

**return** **True**

*# Otherwise we don't do anything*

**else**:

**pass**

check\_even\_list([1,2,3])

Out:true

**Return all even numbers in a list**

**def** check\_even\_list(num\_list):

even\_numbers **=** []

*# Go through each number*

**for** number **in** num\_list:

*# Once we get a "hit" on an even number, we append the even number*

**if** number **%** 2 **==** 0:

even\_numbers**.**append(number)

*# Don't do anything if its not even*

**else**:

**pass**

*# Notice the indentation! This ensures we run through the entire for loop*

**return** even\_numbers

check\_even\_list([1,2,3,4,5,6])

Out:[2, 4, 6]

**Returning Tuples for Unpacking**

stock\_prices **=** [('AAPL',200),('GOOG',300),('MSFT',400)]

**for** item **in** stock\_prices:

print(item)

Out:

('AAPL', 200)

('GOOG', 300)

('MSFT', 400)

**for** stock,price **in** stock\_prices:

print(stock)

Out:

AAPL

GOOG

MSFT

**for** stock,price **in** stock\_prices:

print(price)

Out:

200

300

400

**Interactions between functions**

How to shuffle a list in Python

example **=** [1,2,3,4,5]

**from** random **import** shuffle

*# Note shuffle is in-place*

shuffle(example)

print(example)

Out:[3, 1, 4, 5, 2]

**map function**

The **map** function allows you to "map" a function to an iterable object.

Eg:**def** square(num):

**return** num**\*\***2

my\_nums **=** [1,2,3,4,5]

map(square,my\_nums)

Out:<map at 0x205baec21d0>

*# To get the results, either iterate through map()*

*# or just cast to a list*

list(map(square,my\_nums))

Out:[1, 4, 9, 16, 25]

Eg:

**def** splicer(mystring):

**if** len(mystring) **%** 2 **==** 0:

**return** 'even'

**else**:

**return** mystring[0]

mynames **=** ['John','Cindy','Sarah','Kelly','Mike']

list(map(splicer,mynames))

Out:['even', 'C', 'S', 'K', 'even']

**filter function**

The filter function returns an iterator yielding those items of iterable for which function(item) is true. Meaning you need to filter by a function that returns either True or False. Then passing that into filter (along with your iterable) and you will get back only the results that would return True when passed to the function.

Eg:**def** check\_even(num):

**return** num **%** 2 **==** 0

nums **=** [0,1,2,3,4,5,6,7,8,9,10]

print(filter(check\_even,nums))

Out:<filter at 0x205baed4710>

Print (list(filter(check\_even,nums))

)

Out:[0, 2, 4, 6, 8, 10]

**lambda expression**

lambda expressions allow us to create "anonymous" functions. This basically means we can quickly make ad-hoc functions without needing to properly define a function using def.

**lambda's body is a single expression, not a block of statements.**

• The lambda's body is similar to what we would put in a def body's return statement. We simply type the result as an expression instead of explicitly returning it. Because it is limited to an expression, a lambda is less general that a def. We can only squeeze design, to limit program nesting. lambda is designed for coding simple functions, and def handles the larger tasks.

**def** square(num):

result **=** num**\*\***2

**return** result

square(2)

Out:4

**def** square(num): **return** num**\*\***2

square(2)

Out:4

square **=** **lambda** num: num **\*\***2

square(2)

Out:4

**Nested Statements and Scope**

When you create a variable name in Python the name is stored in a *name-space*. Variable names also have a *scope*, the scope determines the visibility of that variable name to other parts of your code.

x **=** 25

**def** printer():

x **=** 50

**return** x

*print(x)*

*Out:25*

*print(printer())*

*Out:50*

In simple terms, the idea of scope can be described by 3 general rules:

1 Name assignments will create or change local names by default.

2 Name references search (at most) four scopes, these are:

• local

• enclosing functions

• global

• built-in

3 Names declared in global and nonlocal statements map assigned names to enclosing module and function scopes.

**LEGB Rule:**

L: Local — Names assigned in any way within a function (def or lambda), and not declared global in that function.

E: Enclosing function locals — Names in the local scope of any and all enclosing functions (def or lambda), from inner to outer.

G: Global (module) — Names assigned at the top-level of a module file, or declared global in a def within the file.

B: Built-in (Python) — Names preassigned in the built-in names module : open, range, SyntaxError,...

**Enclosing function locals**

This occurs when we have a function inside a function (nested functions)

name **=** 'This is a global name'

**def** greet():

*# Enclosing function*

name **=** 'Sammy'

**def** hello():

print('Hello '**+**name)

hello()

greet()

Out:Hello Sammy

**Local Variables**

When you declare variables inside a function definition, they are not related in any way to other variables with the same names used outside the function - i.e. variable names are local to the function. This is called the scope of the variable. All variables have the scope of the block they are declared in starting from the point of definition of the name.

x **=** 50-global

**def** func(x):

print('x is', x)

x **=** 2-local

print('Changed local x to', x)

func(x)

print('x is still', x)

Out:

x is 50

Changed local x to 2

x is still 50

**The global statement**

If you want to assign a value to a name defined at the top level of the program (i.e. not inside any kind of scope such as functions or classes), then you have to tell Python that the name is not local, but it is global. We do this using the global statement. It is impossible to assign a value to a variable defined outside a function without the global statement

Eg:x **=** 50

**def** func():

**global** x

print('This function is now using the global x!')

print('Because of global x is: ', x)

x **=** 2

print('Ran func(), changed global x to', x)

print('Before calling func(), x is: ', x)

func()

print('Value of x (outside of func()) is: ', x)

Out:

Before calling func(), x is: 50

This function is now using the global x!

Because of global x is: 50

Ran func(), changed global x to 2

Value of x (outside of func()) is: 2

**\*args and \*\*kwargs**

**\*args**

When a function parameter starts with an asterisk, it allows for an *arbitrary number* of arguments, and the function takes them in as a tuple of values.

Eg:

**def** myfunc(**\***args):

**return** sum(args)**\***.05

myfunc(40,60,20)

Out:6.0

**\*\*kwargs**

Python offers a way to handle arbitrary numbers of *keyworded* arguments. Instead of creating a tuple of values, \*\*kwargs builds a dictionary of key/value pairs.

**def** myfunc(**\*\***kwargs):

**if** 'fruit' **in** kwargs:

print(f"My favorite fruit is {kwargs['fruit']}") *# review String Formatting and f-strings if this syntax is unfamiliar*

**else**:

print("I don't like fruit")

myfunc(fruit**=**'pineapple')

Out:My favorite fruit is pineapple

myfunc()

Out:I don't like fruit

**\*args and \*\*kwargs combined**

You can pass \*args and \*\*kwargs into the same function, but \*args have to appear before \*\*kwargs

**def** myfunc(**\***args, **\*\***kwargs):

**if** 'fruit' **and** 'juice' **in** kwargs:

print(f"I like {' and '**.**join(args)} and my favorite fruit is {kwargs['fruit']}")

print(f"May I have some {kwargs['juice']} juice?")

**else**:

**pass**

myfunc('eggs','spam',fruit**=**'cherries',juice**=**'orange')

Out:

I like eggs and spam and my favorite fruit is cherriesz

May I have some orange juice?

**Object Oriented Programming**

• Objects

• Using the *class* keyword

• Creating class attributes

• Creating methods in a class

• Learning about Inheritance

• Learning about Polymorphism

• Learning about Special Methods for classes

Lets start the lesson by remembering about the Basic Python Objects. For example:

lst **=** [1,2,3]

lst**.**count(2)

Out:1

**Objects**

In Python, *everything is an object*.

check the type of object something is:

Eg:

print(type(1))

print(type([]))

print(type(()))

print(type({}))

Out:

<class 'int'>

<class 'list'>

<class 'tuple'>

<class 'dict'>

**class**

User defined objects are created using the class keyword. The class is a blueprint that defines the nature of a future object. From classes we can construct instances. An instance is a specific object created from a particular class. For example, above we created the object lst which was an instance of a list object.

Eg:

*# Create a new object type called Sample*

**class** Sample:

**pass**

*# Instance of Sample*

x **=** Sample()

print(type(x))

Out:<class '\_\_main\_\_.Sample'>

Note how x is now the reference to our new instance of a Sample class.

Inside of the class we currently just have pass. But we can define class attributes and methods.

An **attribute** is a characteristic of an object. A **method** is an operation we can perform with the object.

For example, we can create a class called Dog. An attribute of a dog may be its breed or its name, while a method of a dog may be defined by a .bark() method which returns a sound.

**Attributes**

The syntax for creating an attribute is:

self.attribute = something

There is a special method called:

\_\_init\_\_()

This method is used to initialize the attributes of an object. For example:

**class** Dog:

**def** \_\_init\_\_(self,breed):

self**.**breed **=** breed

sam **=** Dog(breed**=**'Lab')

frank **=** Dog(breed**=**'Huskie')

Lets break down what we have above.The special method

**\_\_init\_\_()**

is called automatically right after the object has been created:

**def \_\_init\_\_(self, breed):**

Each attribute in a class definition begins with a reference to the instance object. It is by convention named self. The breed is the argument. The value is passed during the class instantiation.

**self.breed = breed**

Now we have created two instances of the Dog class. With two breed types, we can then access these attributes like this:

Print (sam**.**breed);

Out:Lab

Print (frank**.**breed)

Out: Huskie

Note how we don't have any parentheses after breed; this is because it is an attribute and doesn't take any arguments.

In Python there are also *class object attributes*. These Class Object Attributes are the same for any instance of the class. For example, we could create the attribute *species* for the Dog class. Dogs, regardless of their breed, name, or other attributes, will always be mammals. We apply this logic in the following manner:

Eg:

**class** Dog:

*# Class Object Attribute*

species **=** 'mammal'

**def** \_\_init\_\_(self,breed,name):

self**.**breed **=** breed

self**.**name **=** name

sam **=** Dog('Lab','Sam')

print(sam.name)

Out:sam

Note that the Class Object Attribute is defined outside of any methods in the class. Also by convention, we place them first before the init.

Print (sam**.**species);

Out:mammal

**Methods**

Methods are functions defined inside the body of a class. They are used to perform operations with the attributes of our objects. Methods are a key concept of the OOP paradigm. They are essential to dividing responsibilities in programming, especially in large applications.

You can basically think of methods as functions acting on an Object that take the Object itself into account through its *self* argument.

Let's go through an example of creating a Circle class:

**class** Circle:

pi **=** 3.14

*# Circle gets instantiated with a radius (default is 1)*

**def** \_\_init\_\_(self, radius**=**1):

self**.**radius **=** radius

self**.**area **=** radius **\*** radius **\*** Circle**.**pi

*# Method for resetting Radius*

**def** setRadius(self, new\_radius):

self**.**radius **=** new\_radius

self**.**area **=** new\_radius **\*** new\_radius **\*** self**.**pi

*# Method for getting Circumference*

**def** getCircumference(self):

**return** self**.**radius **\*** self**.**pi **\*** 2

c **=** Circle()

print('Radius is: ',c**.**radius)

print('Area is: ',c**.**area)

print('Circumference is: ',c**.**getCircumference())

Out:

Radius is: 1

Area is: 3.14

Circumference is: 6.28

In the \_\_init\_\_ method above, in order to calculate the area attribute, we had to call Circle.pi. This is because the object does not yet have its own .pi attribute, so we call the Class Object Attribute pi instead.

In the setRadius method, however, we'll be working with an existing Circle object that does have its own pi attribute. Here we can use either Circle.pi or self.pi.

Now let's change the radius and see how that affects our Circle object:

c**.**setRadius(2)

print('Radius is: ',c**.**radius)

print('Area is: ',c**.**area)

print('Circumference is: ',c**.**getCircumference())

Out:

Radius is: 2

Area is: 12.56

Circumference is: 12.56

**Inheritance**

Inheritance is a way to form new classes using classes that have already been defined. The newly formed classes are called derived classes, the classes that we derive from are called base classes. Important benefits of inheritance are code reuse and reduction of complexity of a program. The derived classes (descendants) override or extend the functionality of base classes (ancestors).

Let's see an example by incorporating our previous work on the Dog class:

**class** Animal:

**def** \_\_init\_\_(self):

print("Animal created")

**def** whoAmI(self):

print("Animal")

**def** eat(self):

print("Eating")

**class** Dog(Animal):

**def** \_\_init\_\_(self):

Animal**.**\_\_init\_\_(self)

print("Dog created")

**def** whoAmI(self):

print("Dog")

**def** bark(self):

print("Woof!")

d **=** Dog()

Out:

Animal created

Dog created

print(d**.**whoAmI())

Out: Dog

print(d**.**eat())

Out: Eating

Print (d**.**bark())

Out:Woof!

In this example, we have two classes: Animal and Dog. The Animal is the base class, the Dog is the derived class.

The derived class inherits the functionality of the base class.

• It is shown by the eat() method.

The derived class modifies existing behavior of the base class.

• shown by the whoAmI() method.

Finally, the derived class extends the functionality of the base class, by defining a new bark() method.

**Polymorphism**

In Python, *polymorphism* refers to the way in which different object classes can share the same method name, and those methods can be called from the same place even though a variety of different objects might be passed in.

Eg:

**class** Dog:

**def** \_\_init\_\_(self, name):

self**.**name **=** name

**def** speak(self):

**return** self**.**name**+**' says Woof!'

**class** Cat:

**def** \_\_init\_\_(self, name):

self**.**name **=** name

**def** speak(self):

**return** self**.**name**+**' says Meow!'

niko **=** Dog('Niko')

felix **=** Cat('Felix')

print(niko**.**speak())

print(felix**.**speak())

Out:Niko says Woof!

Felix says Meow!

Here we have a Dog class and a Cat class, and each has a .speak() method. When called, each object's .speak() method returns a result unique to the object.

**There a few different ways to demonstrate polymorphism. First, with a for loop:**

Eg:

**for** pet **in** [niko,felix]:

print(pet**.**speak())

Out:

Niko says Woof!

Felix says Meow!

**Another is with functions:**

**Eg:**

**def** pet\_speak(pet):

print(pet**.**speak())

pet\_speak(niko)

pet\_speak(felix)

Out:

Niko says Woof!

Felix says Meow!

A more common practice is to use abstract classes and inheritance. An abstract class is one that never expects to be instantiated. For example, we will never have an Animal object, only Dog and Cat objects, although Dogs and Cats are derived from Animals:

Eg:

**class** Animal:

**def** \_\_init\_\_(self, name): *# Constructor of the class*

self**.**name **=** name

**def** speak(self): *# Abstract method, defined by convention only*

**raise** NotImplementedError("Subclass must implement abstract method")

**class** Dog(Animal):

**def** speak(self):

**return** self**.**name**+**' says Woof!'

**class** Cat(Animal):

**def** speak(self):

**return** self**.**name**+**' says Meow!'

fido **=** Dog('Fido')

isis **=** Cat('Isis')

print(fido**.**speak())

print(isis**.**speak())

Out:

Fido says Woof!

Isis says Meow!

Real life examples of polymorphism include:

• opening different file types - different tools are needed to display Word, pdf and Excel files

• adding different objects - the + operator performs arithmetic and concatenation

**Special Methods**

Finally let's go over special methods. Classes in Python can implement certain operations with special method names. These methods are not actually called directly but by Python specific language syntax. For example let's create a Book class:

Eg:

**class** Book:

**def** \_\_init\_\_(self, title, author, pages):

print("A book is created")

self**.**title **=** title

self**.**author **=** author

self**.**pages **=** pages

**def** \_\_str\_\_(self):

**return** "Title: %s, author: %s, pages: %s" **%**(self**.**title, self**.**author, self**.**pages)

**def** \_\_len\_\_(self):

**return** self**.**pages

**def** \_\_del\_\_(self):

print("A book is destroyed")

book **=** Book("Python Rocks!", "Jose Portilla", 159)

*#Special Methods*

print(book)

print(len(book))

**del** book

Out:

A book is created

Title: Python Rocks!, author: Jose Portilla, pages: 159

159

A book is destroyed

The \_\_init\_\_(), \_\_str\_\_(), \_\_len\_\_() and \_\_del\_\_() methods

These special methods are defined by their use of underscores. They allow us to use Python specific functions on objects created through our class.

**Problem 1**

Fill in the Line class methods to accept coordinates as a pair of tuples and return the slope and distance of the line.

**class** Line:

**def** \_\_init\_\_(self,coor1,coor2):

**pass**

**def** distance(self):

**pass**

**def** slope(self):

**pass**

*# EXAMPLE OUTPUT*

coordinate1 **=** (3,2)

coordinate2 **=** (8,10)

li **=** Line(coordinate1,coordinate2)

print(li**.**distance())

Out: 9.433981132056603

print(li**.**slope())

Out:1.6

**Problem 2**

Fill in the class

**class** Cylinder:

**def** \_\_init\_\_(self,height**=**1,radius**=**1):

**pass**

**def** volume(self):

**pass**

**def** surface\_area(self):

**pass**

*# EXAMPLE OUTPUT*

c **=** Cylinder(2,3)

print(c**.**volume())

Out:56.52

Print (c**.**surface\_area())

Out:94.2

**Answer:**

**Problem 1:**

**class** Line(object):

**def** \_\_init\_\_(self,coor1,coor2):

self**.**coor1 **=** coor1

self**.**coor2 **=** coor2

**def** distance(self):

x1,y1 **=** self**.**coor1

x2,y2 **=** self**.**coor2

**return** ((x2**-**x1)**\*\***2 **+** (y2**-**y1)**\*\***2)**\*\***0.5

**def** slope(self):

x1,y1 **=** self**.**coor1

x2,y2 **=** self**.**coor2

**return** (y2**-**y1)**/**(x2**-**x1)

coordinate1 **=** (3,2)

coordinate2 **=** (8,10)

li **=** Line(coordinate1,coordinate2)

print(li**.**distance());

Out: 9.433981132056603

Print (li**.**slope())

Out:1.6

**Problem 2**

**class** Cylinder:

**def** \_\_init\_\_(self,height**=**1,radius**=**1):

self**.**height **=** height

self**.**radius **=** radius

**def** volume(self):

**return** self**.**height**\***3.14**\***(self**.**radius)**\*\***2

**def** surface\_area(self):

top **=** 3.14 **\*** (self**.**radius)**\*\***2

**return** (2**\***top) **+** (2**\***3.14**\***self**.**radius**\***self**.**height)

c **=** Cylinder(2,3)

Print(c**.**volume())

Out:56.52

Print (c**.**surface\_area())

Out:

94.2

**Modules and Packages**

Modules in Python are simply Python files with the .py extension, which implement a set of functions. Modules are imported from other modules using the import command.

To import a module, we use the import command.

The first time a module is loaded into a running Python script, it is initialized by executing the code in the module once. If another module in your code imports the same module again, it will not be loaded twice but once only - so local variables inside the module act as a "singleton" - they are initialized only once.

If we want to import the math module, we simply import the name of the module:

Eg:

*# import the library*

**import** math

*# use it (ceiling rounding)*

math**.**ceil(2.4)

Out:3

**Exploring built-in modules**

Two very important functions come in handy when exploring modules in Python - the dir and help functions.

We can look for which functions are implemented in each module by using the dir function:

print(dir(math))

out;

['\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']

When we find the function in the module we want to use, we can read about it more using the help function, inside the Python interpreter:

help(math**.**ceil)

Out:

Help on built-in function ceil in module math:

ceil(...)

ceil(x)

Return the ceiling of x as an Integral.

This is the smallest integer >= x.

**Writing modules**

Writing Python modules is very simple. To create a module of your own, simply create a new .py file with the module name, and then import it using the Python file name (without the .py extension) using the import command.

**Writing packages**

Packages are name-spaces which contain multiple packages and modules themselves. They are simply directories, but with a twist.

Each package in Python is a directory which MUST contain a special file called **\_\_init\_\_.py**. This file can be empty, and it indicates that the directory it contains is a Python package, so it can be imported the same way a module can be imported.

If we create a directory called foo, which marks the package name, we can then create a module inside that package called bar. We also must not forget to add the **\_\_init\_\_.py** file inside the foo directory.

To use the module bar, we can import it in two ways:

*# Just an example, this won't work*

**import** foo.bar

*# OR could do it this way*

**from** foo **import** bar

In the first method, we must use the foo prefix whenever we access the module bar. In the second method, we don't, because we import the module to our module's name-space.

The **\_\_init\_\_.py** file can also decide which modules the package exports as the API, while keeping other modules internal, by overriding the **\_\_all\_\_** variable, like so:

\_\_init\_\_**.**py:

\_\_all\_\_ **=** ["bar"]

**Errors and Exception Handling**

Eg:

print('Hello)

Error:

**File "<ipython-input-1-db8c9988558c>", line 1**

**print('Hello)**

**^**

**SyntaxError:** EOL while scanning string literal

Note how we get a SyntaxError, with the further description that it was an EOL (End of Line Error) while scanning the string literal. This is specific enough for us to see that we forgot a single quote at the end of the line. Understanding these various error types will help you debug your code much faster.

This type of error and description is known as an **Exception**. Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal.

**try and except**

The basic terminology and syntax used to handle errors in Python are the try and except statements. The code which can cause an exception to occur is put in the try block and the handling of the exception is then implemented in the except block of code.

The syntax follows:

try:

You do your operations here...

...

except ExceptionI:

If there is ExceptionI, then execute this block.

except ExceptionII:

If there is ExceptionII, then execute this block.

...

else:

If there is no exception then execute this block.

We can also just check for any exception with just using except: To get a better understanding of all this let's check out an example: We will look at some code that opens and writes a file:

**try**:

f **=** open('testfile','w')

f**.**write('Test write this')

**except** IOError:

*# This will only check for an IOError exception and then execute this print statement*

print("Error: Could not find file or read data")

**else**:

print("Content written successfully")

f**.**close()

Out:Content written successfully

Now let's see what would happen if we did not have write permission (opening only with 'r'):

**try**:

f **=** open('testfile','r')

f**.**write('Test write this')

**except** IOError:

*# This will only check for an IOError exception and then execute this print statement*

print("Error: Could not find file or read data")

**else**:

print("Content written successfully")

f**.**close()

Out:

Error: Could not find file or read data

We could have also just said except: if we weren't sure what exception would occur. For example:

**try**:

f **=** open('testfile','r')

f**.**write('Test write this')

**except**:

*# This will check for any exception and then execute this print statement*

print("Error: Could not find file or read data")

**else**:

print("Content written successfully")

f**.**close()

Out:

Error: Could not find file or read data

Now what if we kept wanting to run code after the exception occurred? This is where finally comes in.

**finally**

The finally: block of code will always be run regardless if there was an exception in the try code block.

The syntax is:

try:

Code block here

...

Due to any exception, this code may be skipped!

finally:

This code block would always be executed.

For example:

**try**:

f **=** open("testfile", "w")

f**.**write("Test write statement")

f**.**close()

**finally**:

print("Always execute finally code blocks")

Out:Always execute finally code blocks

We can use this in conjunction with except. Let's see a new example that will take into account a user providing the wrong input:

**def** askint():

**try**:

val **=** int(input("Please enter an integer: "))

**except**:

print("Looks like you did not enter an integer!")

**finally**:

print("Finally, I executed!")

print(val)

askint()

Out:

Please enter an integer: 5

Finally, I executed!

5

Error:

**---------------------------------------------------------------------------**

**UnboundLocalError** Traceback (most recent call last)

**<ipython-input-8-cc291aa76c10>** in <module>**()**

**----> 1** askint**()**

**<ipython-input-6-c97dd1c75d24>** in askint**()**

7 **finally:**

8 print**("Finally, I executed!")**

**----> 9** print**(**val**)**

**UnboundLocalError**: local variable 'val' referenced before assignment

Notice how we got an error when trying to print val (because it was never properly assigned). Let's remedy this by asking the user and checking to make sure the input type is an integer:

**def** askint():

**try**:

val **=** int(input("Please enter an integer: "))

**except**:

print("Looks like you did not enter an integer!")

val **=** int(input("Try again-Please enter an integer: "))

**finally**:

print("Finally, I executed!")

print(val)

askint()

Out:

Please enter an integer: five

Looks like you did not enter an integer!

Try again-Please enter an integer: four

Finally, I executed!

Errors:

**---------------------------------------------------------------------------**

**ValueError** Traceback (most recent call last)

**<ipython-input-9-92b5f751eb01>** in askint**()**

2 **try:**

**----> 3** val **=** int**(**input**("Please enter an integer: "))**

4 **except:**

**ValueError**: invalid literal for int() with base 10: 'five'

During handling of the above exception, another exception occurred:

**ValueError** Traceback (most recent call last)

**<ipython-input-10-cc291aa76c10>** in <module>**()**

**----> 1** askint**()**

**<ipython-input-9-92b5f751eb01>** in askint**()**

4 **except:**

5 print**("Looks like you did not enter an integer!")**

**----> 6** val **=** int**(**input**("Try again-Please enter an integer: "))**

7 **finally:**

8 print**("Finally, I executed!")**

**ValueError**: invalid literal for int() with base 10: 'four'

How can we continually keep checking? We can use a while loop!

**def** askint():

**while** **True**:

**try**:

val **=** int(input("Please enter an integer: "))

**except**:

print("Looks like you did not enter an integer!")

**continue**

**else**:

print("Yep that's an integer!")

**break**

**finally**:

print("Finally, I executed!")

print(val)

askint()

Out:

Please enter an integer: five

Looks like you did not enter an integer!

Finally, I executed!

Please enter an integer: four

Looks like you did not enter an integer!

Finally, I executed!

Please enter an integer: 3

Yep that's an integer!

Finally, I executed!

So why did our function print "Finally, I executed!" after each trial, yet it never printed val itself? This is because with a try/except/finally clause, any continue or break statements are reserved until *after* the try clause is completed. This means that even though a successful input of **3** brought us to the else: block, and a break statement was thrown, the try clause continued through to finally: before breaking out of the while loop. And since print(val) was outside the try clause, the break statement prevented it from running.

Let's make one final adjustment:

**def** askint():

**while** **True**:

**try**:

val **=** int(input("Please enter an integer: "))

**except**:

print("Looks like you did not enter an integer!")

**continue**

**else**:

print("Yep that's an integer!")

print(val)

**break**

**finally**:

print("Finally, I executed!")

askint()

Out:

Please enter an integer: six

Looks like you did not enter an integer!

Finally, I executed!

Please enter an integer: 6

Yep that's an integer!

6

Finally, I executed!

**Problem 1**

Handle the exception thrown by the code below by using try and except blocks.

**for** i **in** ['a','b','c']:

print(i**\*\***2)

Error:

**---------------------------------------------------------------------------**

**TypeError** Traceback (most recent call last)

**<ipython-input-1-c35f41ad7311>** in <module>**()**

1 **for** i **in** **['a','b','c']:**

**----> 2** print**(**i**\*\*2)**

**TypeError**: unsupported operand type(s) for \*\* or pow(): 'str' and 'int'

**Problem 2**

Handle the exception thrown by the code below by using try and except blocks. Then use a finally block to print 'All Done.'

x **=** 5

y **=** 0

z **=** x**/**y

Error:

**---------------------------------------------------------------------------**

**ZeroDivisionError** Traceback (most recent call last)

**<ipython-input-2-6f985c4c80dd>** in <module>**()**

2 y **=** **0**

3

**----> 4** z **=** x**/**y

**ZeroDivisionError**: division by zero

**Problem 3**

Write a function that asks for an integer and prints the square of it. Use a while loop with a try, except, else block to account for incorrect inputs.

**def** ask():

**pass**

ask()

Out:

Input an integer: null

An error occurred! Please try again!

Input an integer: 2

Thank you, your number squared is: 4

**Answers**

**Problem 1**

**try**:

**for** i **in** ['a','b','c']:

print(i**\*\***2)

**except**:

print("An error occurred!")

Out:

An error occurred!

**Problem 2**

x **=** 5

y **=** 0

**try**:

z **=** x**/**y

**except** ZeroDivisionError:

print("Can't divide by Zero!")

**finally**:

print('All Done!')

Out:

Can't divide by Zero!

All Done!

**Problem 3**

**def** ask():

**while** **True**:

**try**:

n **=** int(input('Input an integer: '))

**except**:

print('An error occurred! Please try again!')

**continue**

**else**:

**break**

print('Thank you, your number squared is: ',n**\*\***2)

ask()

**Out:**

Input an integer: null

An error occurred! Please try again!

Input an integer: 2

Thank you, your number squared is: 4

**Unit Testing**

Equally important as writing good code is writing good tests. Better to find bugs yourself than have them reported to you by end users!

**Testing tools**

There are dozens of good testing libraries out there. Most are third-party packages that require an install, such as:

• pylint

• pyflakes

• pep8

These are simple tools that merely look at your code, and they'll tell you if there are style issues or simple problems like variable names being called before assignment.

A far better way to test your code is to write tests that send sample data to your program, and compare what's returned to a desired outcome.  
Two such tools are available from the standard library:

• unittest

• doctest

Let's look at pylint first, then we'll do some heavier lifting with unittest.

**pylint**

pylint tests for style as well as some very basic program logic.

you should install pylint:

**!** pip install pylint

Let's save a very simple script:

**%%writefile** simple1.py

a **=** 1

b **=** 2

print(a)

print(B)

**Out:**

Overwriting simple1.py

Now let's check it using pylint

**!** pylint simple1.py

**Out:**

\*\*\*\*\*\*\*\*\*\*\*\*\* Module simple1

C: 4, 0: Final newline missing (missing-final-newline)

C: 1, 0: Missing module docstring (missing-docstring)

C: 1, 0: Invalid constant name "a" (invalid-name)

C: 2, 0: Invalid constant name "b" (invalid-name)

E: 4, 6: Undefined variable 'B' (undefined-variable)

Your code has been rated at -12.50/10 (previous run: 8.33/10, -20.83)

No config file found, using default configuration

Pylint first lists some styling issues - it would like to see an extra newline at the end, modules and function definitions should have descriptive docstrings, and single characters are a poor choice for variable names.

More importantly, however, pylint identified an error in the program - a variable called before assignment.

More importantly, however, pylint identified an error in the program - a variable called before assignment. This needs fixing.

**unittest**

unittest lets you write your own test programs. The goal is to send a specific set of data to your program, and analyze the returned results against an expected result.

Let's generate a simple script that capitalizes words in a given string. We'll call it **cap.py**.

**%%writefile** cap.py

**def** cap\_text(text):

**return** text**.**capitalize()

Overwriting cap.py

**map()**

map() is a built-in Python function that takes in two or more arguments: a function and one or more iterables, in the form:

map(function, iterable, ...)

map() returns an *iterator* - that is, map() returns a special object that yields one result at a time as needed.

When we went over list comprehensions we created a small expression to convert Celsius to Fahrenheit. Let's do the same here but use map:

**def** fahrenheit(celsius):

**return** (9**/**5)**\***celsius **+** 32

temps **=** [0, 22.5, 40, 100]

Now let's see map() in action:

F\_temps **=** map(fahrenheit, temps)

*#Show*

list(F\_temps)

**Out:**

[32.0, 72.5, 104.0, 212.0]

In the example above, map() applies the fahrenheit function to every item in temps. However, we don't have to define our functions beforehand; we can use a lambda expression instead:

list(map(**lambda** x: (9**/**5)**\***x **+** 32, temps))

**Out:**

[32.0, 72.5, 104.0, 212.0]

**map() with multiple iterables**

map() can accept more than one iterable. The iterables should be the same length - in the event that they are not, map() will stop as soon as the shortest iterable is exhausted.

For instance, if our function is trying to add two values **x** and **y**, we can pass a list of **x** values and another list of **y** values to map(). The function (or lambda) will be fed the 0th index from each list, and then the 1st index, and so on until the n-th index is reached.

Let's see this in action with two and then three lists:

a **=** [1,2,3,4]

b **=** [5,6,7,8]

c **=** [9,10,11,12]

list(map(**lambda** x,y:x**+**y,a,b))

Out:

[6, 8, 10, 12]

*# Now all three lists*

list(map(**lambda** x,y,z:x**+**y**+**z,a,b,c))

Out:[15, 18, 21, 24]

**reduce()**

The function reduce(function, sequence) continually applies the function to the sequence. It then returns a single value.

If seq = [ s1, s2, s3, ... , sn ], calling reduce(function, sequence) works like this:

• •At first the first two elements of seq will be applied to function, i.e. func(s1,s2)

• The list on which reduce() works looks now like this: [ function(s1, s2), s3, ... , sn ]

• In the next step the function will be applied on the previous result and the third element of the list, i.e. function(function(s1, s2),s3)

• The list looks like this now: [ function(function(s1, s2),s3), ... , sn ]

• It continues like this until just one element is left and return this element as the result of reduce()

Let's see an example:

**from** functools **import** reduce

lst **=**[47,11,42,13]

reduce(**lambda** x,y: x**+**y,lst)

Out:

113

**all() and any()**

all() and any() are built-in functions in Python that allow us to conveniently check for boolean matching in an iterable. all() will return True if all elements in an iterable are True. It is the same as this function code:

def all(iterable):

for element in iterable:

if not element:

return False

return True

any() will return True if any of the elements in the iterable are True. It is equivalent to the following function code:

def any(iterable):

for element in iterable:

if element:

return True

return False

Eg:

lst **=** [**True**,**True**,**False**,**True**]

all(lst)

Out:False

#Returns False because not all elements are True.

any(lst)

Out:True

#Returns True because at least one of the elements in the list is True

**complex()**

complex() returns a complex number with the value real + imag\*1j or converts a string or number to a complex number.

If the first parameter is a string, it will be interpreted as a complex number and the function must be called without a second parameter. The second parameter can never be a string. Each argument may be any numeric type (including complex). If imag is omitted, it defaults to zero and the constructor serves as a numeric conversion like int and float. If both arguments are omitted, returns 0j.

Eg:

*# Create 2+3j*

complex(2,3)

Out:(2+3j)

Eg:

complex(10,1)

Out:(10+1j)

We can also pass strings:

complex('12+2j')

Out:(12+2j)

**Problem 1**

Use map() to create a function which finds the length of each word in the phrase (broken by spaces) and returns the values in a list.

The function will have an input of a string, and output a list of integers.

**def** word\_lengths(phrase):

**pass**

word\_lengths('How long are the words in this phrase')

Out:[3, 4, 3, 3, 5, 2, 4, 6]

**Problem 2**

Use reduce() to take a list of digits and return the number that they correspond to. For example, [1, 2, 3] corresponds to one-hundred-twenty-three.

*Do not convert the integers to strings!*

**from** functools **import** reduce

**def** digits\_to\_num(digits):

**pass**

digits\_to\_num([3,4,3,2,1])

Out:34321

**Problem 3**

Use filter to return the words from a list of words which start with a target letter.

**def** filter\_words(word\_list, letter):

**pass**

l **=** ['hello','are','cat','dog','ham','hi','go','to','heart']

filter\_words(l,'h')

Out:['hello', 'ham', 'hi', 'heart']

**Problem 4**

Use zip() and a list comprehension to return a list of the same length where each value is the two strings from L1 and L2 concatenated together with connector between them. Look at the example output below:

**def** concatenate(L1, L2, connector):

**pass**

concatenate(['A','B'],['a','b'],'-')

Out:['A-a', 'B-b']

**Problem 5**

Use enumerate() and other skills to return a dictionary which has the values of the list as keys and the index as the value. You may assume that a value will only appear once in the given list.

**def** d\_list(L):

**pass**

d\_list(['a','b','c'])

Out:{'a': 0, 'b': 1, 'c': 2}

**Problem 6**

Use enumerate() and other skills from above to return the count of the number of items in the list whose value equals its index.

**def** count\_match\_index(L):

**pass**

count\_match\_index([0,2,2,1,5,5,6,10])

Out:4

**Answers:**

**Problem 1**

**def** word\_lengths(phrase):

**return** list(map(len, phrase**.**split()))

word\_lengths('How long are the words in this phrase')

out:[3, 4, 3, 3, 5, 2, 4, 6]

**Problem 2**

**from** functools **import** reduce

**def** digits\_to\_num(digits):

**return** reduce(**lambda** x,y:x**\***10 **+** y,digits)

digits\_to\_num([3,4,3,2,1])

Out:34321

**Problem 3**

**def** filter\_words(word\_list, letter):

**return** list(filter(**lambda** word:word[0]**==**letter,word\_list))

words **=** ['hello','are','cat','dog','ham','hi','go','to','heart']

filter\_words(words,'h')

Out:['hello', 'ham', 'hi', 'heart']

**Problem 4**

**def** concatenate(L1, L2, connector):

**return** [word1**+**connector**+**word2 **for** (word1,word2) **in** zip(L1,L2)]

concatenate(['A','B'],['a','b'],'-')

Out:['A-a', 'B-b']

**Problem 5**

**def** d\_list(L):

**return** {key:value **for** value,key **in** enumerate(L)}

d\_list(['a','b','c'])

**Out:**{'a': 0, 'b': 1, 'c': 2}

**Problem 6**

**def** count\_match\_index(L):

**return** len([num **for** count,num **in** enumerate(L) **if** num**==**count])

count\_match\_index([0,2,2,1,5,5,6,10])

**Out:4**

**Decorators**

Decorators can be thought of as functions which modify the *functionality* of another function. They help to make your code shorter and more "Pythonic".

To properly explain decorators we will slowly build up from functions.

**Functions Review**

**def** func():

**return** 1

func()

Out:1

**Scope Review**

s **=** 'Global Variable'

**def** check\_for\_locals():

print(locals())

Remember that Python functions create a new scope, meaning the function has its own namespace to find variable names when they are mentioned within the function. We can check for local variables and global variables with the locals() and globals() functions. For example:

Inp: print(globals())

Here we get back a dictionary of all the global variables, many of them are predefined in Python. So let's go ahead and look at the keys:

print(globals()**.**keys())

Note how **s** is there, the Global Variable we defined as a string:

globals()['s']

Now let's run our function to check for local variables that might exist inside our function (there shouldn't be any)

check\_for\_locals()

Remember that in Python **everything is an object**. That means functions are objects which can be assigned labels and passed into other functions. Lets start with some simple examples:

**def** hello(name**=**'Jose'):

**return** 'Hello '**+**name

hello()

Out:'Hello Jose'

Assign another label to the function. Note that we are not using parentheses here because we are not calling the function **hello**, instead we are just passing a function object to the **greet** variable.

greet **=** hello

greet

Out:<function \_\_main\_\_.hello>

greet()

Out:'Hello Jose'

So what happens when we delete the name **hello**?

**del** hello

hello()

**---------------------------------------------------------------------------**

**NameError** Traceback (most recent call last)

**<ipython-input-9-a75d7781aaeb>** in <module>**()**

**----> 1** hello**()**

**NameError**: name 'hello' is not defined

greet()

out:Hello Jose'

Even though we deleted the name **hello**, the name **greet** *still points to* our original function object. It is important to know that functions are objects that can be passed to other objects!

**Functions within functions**

 now let's see how we can define functions inside of other functions:

**def** hello(name**=**'Jose'):

print('The hello() function has been executed')

**def** greet():

**return** '\t This is inside the greet() function'

**def** welcome():

**return** "\t This is inside the welcome() function"

print(greet())

print(welcome())

print("Now we are back inside the hello() function")

hello()

Out:The hello() function has been executed

This is inside the greet() function

This is inside the welcome() function

Now we are back inside the hello() function

welcome()

Error:

**---------------------------------------------------------------------------**

**NameError** Traceback (most recent call last)

**<ipython-input-13-a401d7101853>** in <module>**()**

**----> 1** welcome**()**

**NameError**: name 'welcome' is not defined

Note how due to scope, the welcome() function is not defined outside of the hello() function.

**Returning Functions**

**def** hello(name**=**'Jose'):

**def** greet():

**return** '\t This is inside the greet() function'

**def** welcome():

**return** "\t This is inside the welcome() function"

**if** name **==** 'Jose':

**return** greet

**else**:

**return** welcome

Now let's see what function is returned if we set x = hello(), note how the empty parentheses means that name has been defined as Jose.

x **=** hello()

x

Out:

<function \_\_main\_\_.hello.<locals>.greet>

print(x())

Out: This is inside the greet() function

When we write x = hello(), hello() gets executed and because the name is Jose by default, the function greet is returned. If we change the statement to x = hello(name = "Sam") then the welcome function will be returned. We can also do print(hello()()) which outputs *This is inside the greet() function*.

**Functions as Arguments**

Now let's see how we can pass functions as arguments into other functions:

**def** hello():

**return** 'Hi Jose!'

**def** other(func):

print('Other code would go here')

print(func())

other(hello)

**Out:**

Other code would go here

Hi Jose!

**Creating a Decorator**

**def** new\_decorator(func):

**def** wrap\_func():

print("Code would be here, before executing the func")

func()

print("Code here will execute after the func()")

**return** wrap\_func

**def** func\_needs\_decorator():

print("This function is in need of a Decorator")

func\_needs\_decorator()

**Out:**This function is in need of a Decorator

*# Reassign func\_needs\_decorator*

func\_needs\_decorator **=** new\_decorator(func\_needs\_decorator)

func\_needs\_decorator()

**Out:**

Code would be here, before executing the func

This function is in need of a Decorator

Code here will execute after the func()

A decorator simply wrapped the function and modified its behavior. Now let's understand how we can rewrite this code using the @ symbol, which is what Python uses for Decorators:

@new\_decorator

**def** func\_needs\_decorator():

print("This function is in need of a Decorator")

func\_needs\_decorator()

Out:Code would be here, before executing the func

This function is in need of a Decorator

Code here will execute after the func()

**Iterators and Generators**

Generators allow us to generate as we go along, instead of holding everything in memory.

Generator functions allow us to write a function that can send back a value and then later resume to pick up where it left off. This type of function is a generator in Python, allowing us to generate a sequence of values over time. The main difference in syntax will be the use of a yield statement.

a generator function will appear very similar to a normal function. The main difference is when a generator function is compiled they become an object that supports an iteration protocol. That means when they are called in your code they don't actually return a value and then exit. Instead, generator functions will automatically suspend and resume their execution and state around the last point of value generation. The main advantage here is that instead of having to compute an entire series of values up front, the generator computes one value and then suspends its activity awaiting the next instruction. This feature is known as *state suspension*.

*# Generator function for the cube of numbers (power of 3)*

**def** gencubes(n):

**for** num **in** range(n):

**yield** num**\*\***3

**for** x **in** gencubes(10):

print(x)

0

1

8

27

64

125

216

343

512

729

Generators are best for calculating large sets of results (particularly in calculations that involve loops themselves) in cases where we don’t want to allocate the memory for all of the results at the same time.

Let's create another example generator which calculates fibonacci numbers:

**def** genfibon(n):

"""

Generate a fibonnaci sequence up to n

"""

a **=** 1

b **=** 1

**for** i **in** range(n):

**yield** a

a,b **=** b,a**+**b

**for** num **in** genfibon(10):

print(num)

Out:

1

1

2

3

5

8

13

21

34

55

**def** fibon(n):

a **=** 1

b **=** 1

output **=** []

**for** i **in** range(n):

output**.**append(a)

a,b **=** b,a**+**b

**return** output

fibon(10)

out:[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

**next() and iter() built-in functions**

A key to fully understanding generators is the next() function and the iter() function.

The next() function allows us to access the next element in a sequence. Lets check it out:

**def** simple\_gen():

**for** x **in** range(3):

**yield** x

*# Assign simple\_gen*

g **=** simple\_gen()

print(next(g))

Out:0

print(next(g))

Out:1

print(next(g))

Out:2

print(next(g))

Error occurs

**---------------------------------------------------------------------------**

**StopIteration** Traceback (most recent call last)

**<ipython-input-12-1dfb29d6357e>** in <module>**()**

**----> 1** print**(**next**(**g**))**

**StopIteration**:

After yielding all the values next() caused a StopIteration error. What this error informs us of is that all the values have been yielded.

Let's go ahead and check out how to use iter(). You remember that strings are iterables:

s **=** 'hello'

*#Iterate over string*

**for** let **in** s:

print(let)

h

e

l

l

o

But that doesn't mean the string itself is an *iterator*! We can check this with the next() function:

next(s)

**---------------------------------------------------------------------------**

**TypeError** Traceback (most recent call last)

**<ipython-input-14-61c30b5fe1d5>** in <module>**()**

**----> 1** next**(**s**)**

**TypeError**: 'str' object is not an iterator

Interesting, this means that a string object supports iteration, but we can not directly iterate over it as we could with a generator function. The iter() function allows us to do just that!

s\_iter **=** iter(s)

next(s\_iter)

Out:h

next(s\_iter)

Out:e

**Collections Module**

The collections module is a built-in module that implements specialized container data types providing alternatives to Python’s general purpose built-in containers.

**Counter**

*Counter* is a *dict* subclass which helps count hashable objects. Inside of it elements are stored as dictionary keys and the counts of the objects are stored as the value.

**from** collections **import** Counter

#**Counter() with lists**

lst **=** [1,2,2,2,2,3,3,3,1,2,1,12,3,2,32,1,21,1,223,1]

Counter(lst)

Out:

Counter({1: 6, 2: 6, 3: 4, 12: 1, 21: 1, 32: 1, 223: 1})

#**Counter with strings**

Counter('aabsbsbsbhshhbbsbs')

Out:Counter({'a': 2, 'b': 7, 'h': 3, 's': 6})

#**Counter with words in a sentence**

s **=** 'How many times does each word show up in this sentence word times each each word'

words **=** s**.**split()

Counter(words)

Out:

Counter({'How': 1,

'does': 1,

'each': 3,

'in': 1,

'many': 1,

'sentence': 1,

'show': 1,

'this': 1,

'times': 2,

'up': 1,

'word': 3})

*# Methods with Counter()*

c **=** Counter(words)

c**.**most\_common(2)

Out:[('each', 3), ('word', 3)]

**Common patterns when using the Counter() object**

sum(c.values()) # total of all counts

c.clear() # reset all counts

list(c) # list unique elements

set(c) # convert to a set

dict(c) # convert to a regular dictionary

c.items() # convert to a list of (elem, cnt) pairs

Counter(dict(list\_of\_pairs)) # convert from a list of (elem, cnt) pairs

c.most\_common()[:-n-1:-1] # n least common elements

c += Counter() # remove zero and negative counts

**defaultdict**

defaultdict is a dictionary-like object which provides all methods provided by a dictionary but takes a first argument (default\_factory) as a default data type for the dictionary. Using defaultdict is faster than doing the same using dict.set\_default method.

**A defaultdict will never raise a KeyError. Any key that does not exist gets the value returned by the default factory.**

**from** collections **import** defaultdict

d **=** {}

d['one']

Error Occur:

**---------------------------------------------------------------------------**

**KeyError** Traceback (most recent call last)

**<ipython-input-8-07706fc5dc20>** in <module>**()**

**----> 1** d**['one']**

**KeyError**: 'one'

d **=** defaultdict(object)

d['one']

<object at 0x216de27bcf0>

**for** item **in** d:

print(item)

Out: one

Can also initialize with default values:

d **=** defaultdict(**lambda**: 0)

d['one']

Out:0

**namedtuple**

The standard tuple uses numerical indexes to access its members, for example:

t **=** (12,13,14)

t[0]

Out:12

Each kind of namedtuple is represented by its own class, created by using the namedtuple() factory function. The arguments are the name of the new class and a string containing the names of the elements.

Eg:

**from** collections **import** namedtuple

Dog **=** namedtuple('Dog',['age','breed','name'])

sam **=** Dog(age**=**2,breed**=**'Lab',name**=**'Sammy')

frank **=** Dog(age**=**2,breed**=**'Shepard',name**=**"Frankie")

We construct the namedtuple by first passing the object type name (Dog) and then passing a string with the variety of fields as a string with spaces between the field names. We can then call on the various attributes:

Print(sam);

Out:Dog(age=2, breed='Lab', name='Sammy')

print(sam**.**age);

Out:2

print(sam**.**breed);

Out:Lab

print(sam[0]);

Out:2

**datetime module**

Python has the datetime module to help deal with timestamps in your code. Time values are represented with the time class. Times have attributes for hour, minute, second, and microsecond. They can also include time zone information. The arguments to initialize a time instance are optional, but the default of 0 is unlikely to be what you want.

**time**

create a timestamp by specifying datetime.time(hour,minute,second,microsecond)

Eg:

**import** datetime

t **=** datetime**.**time(4, 20, 1)

*# Let's show the different components*

print(t)

print('hour :', t**.**hour)

print('minute:', t**.**minute)

print('second:', t**.**second)

print('microsecond:', t**.**microsecond)

print('tzinfo:', t**.**tzinfo)

Out:

04:20:01

hour : 4

minute: 20

second: 1

microsecond: 0

tzinfo: None

Note: A time instance only holds values of time, and not a date associated with the time.

We can also check the min and max values a time of day can have in the module:

print('Earliest :', datetime**.**time**.**min)

print('Latest :', datetime**.**time**.**max)

print('Resolution:', datetime**.**time**.**resolution)

Out:Earliest : 00:00:00

Latest : 23:59:59.999999

Resolution: 0:00:00.000001

The min and max class attributes reflect the valid range of times in a single day.

**Dates**

datetime (as you might suspect) also allows us to work with date timestamps. Calendar date values are represented with the date class. Instances have attributes for year, month, and day. It is easy to create a date representing today’s date using the today() class method.

Let's see some examples:

today **=** datetime**.**date**.**today()

print(today)

print('ctime:', today**.**ctime())

print('tuple:', today**.**timetuple())

print('ordinal:', today**.**toordinal())

print('Year :', today**.**year)

print('Month:', today**.**month)

print('Day :', today**.**day)

Out:2020-06-10

ctime: Wed Jun 10 00:00:00 2020

tuple: time.struct\_time(tm\_year=2020, tm\_mon=6, tm\_mday=10, tm\_hour=0, tm\_min=0, tm\_sec=0, tm\_wday=2, tm\_yday=162, tm\_isdst=-1)

ordinal: 737586

Year : 2020

Month: 6

Day : 10

As with time, the range of date values supported can be determined using the min and max attributes.

print('Earliest :', datetime**.**date**.**min)

print('Latest :', datetime**.**date**.**max)

print('Resolution:', datetime**.**date**.**resolution)

Out:

Earliest : 0001-01-01

Latest : 9999-12-31

Resolution: 1 day, 0:00:00

Another way to create new date instances uses the replace() method of an existing date. For example, you can change the year, leaving the day and month alone.

d1 **=** datetime**.**date(2015, 3, 11)

print('d1:', d1)

d2 **=** d1**.**replace(year**=**1990)

print('d2:', d2)

Out:

d1: 2015-03-11

d2: 1990-03-11

**Arithmetic**

We can perform arithmetic on date objects to check for time differences. For example:

print(d1);

Out:datetime.date(2015, 3, 11)

Print (d2);

Out:datetime.date(1990, 3, 11)

Print (d1**-**d2);

Out:datetime.timedelta(9131)

This gives us the difference in days between the two dates. You can use the timedelta method to specify various units of times (days, minutes, hours, etc.)

**Math and Random Modules**

**import** math

help(math)

Out:

Help on built-in module math:

NAME

math

DESCRIPTION

This module is always available. It provides access to the

mathematical functions defined by the C standard.

FUNCTIONS

acos(...)

acos(x)

Return the arc cosine (measured in radians) of x.

acosh(...)

acosh(x)

Return the inverse hyperbolic cosine of x.

asin(...)

asin(x)

Return the arc sine (measured in radians) of x.

asinh(...)

asinh(x)

Return the inverse hyperbolic sine of x.

atan(...)

atan(x)

Return the arc tangent (measured in radians) of x.

atan2(...)

atan2(y, x)

Return the arc tangent (measured in radians) of y/x.

Unlike atan(y/x), the signs of both x and y are considered.

atanh(...)

atanh(x)

Return the inverse hyperbolic tangent of x.

ceil(...)

ceil(x)

Return the ceiling of x as an Integral.

This is the smallest integer >= x.

copysign(...)

copysign(x, y)

Return a float with the magnitude (absolute value) of x but the sign

of y. On platforms that support signed zeros, copysign(1.0, -0.0)

returns -1.0.

cos(...)

cos(x)

Return the cosine of x (measured in radians).

cosh(...)

cosh(x)

Return the hyperbolic cosine of x.

degrees(...)

degrees(x)

Convert angle x from radians to degrees.

erf(...)

erf(x)

Error function at x.

erfc(...)

erfc(x)

Complementary error function at x.

exp(...)

exp(x)

Return e raised to the power of x.

expm1(...)

expm1(x)

Return exp(x)-1.

This function avoids the loss of precision involved in the direct evaluation of exp(x)-1 for small x.

fabs(...)

fabs(x)

Return the absolute value of the float x.

factorial(...)

factorial(x) -> Integral

Find x!. Raise a ValueError if x is negative or non-integral.

floor(...)

floor(x)

Return the floor of x as an Integral.

This is the largest integer <= x.

fmod(...)

fmod(x, y)

Return fmod(x, y), according to platform C. x % y may differ.

frexp(...)

frexp(x)

Return the mantissa and exponent of x, as pair (m, e).

m is a float and e is an int, such that x = m \* 2.\*\*e.

If x is 0, m and e are both 0. Else 0.5 <= abs(m) < 1.0.

fsum(...)

fsum(iterable)

Return an accurate floating point sum of values in the iterable.

Assumes IEEE-754 floating point arithmetic.

gamma(...)

gamma(x)

Gamma function at x.

gcd(...)

gcd(x, y) -> int

greatest common divisor of x and y

hypot(...)

hypot(x, y)

Return the Euclidean distance, sqrt(x\*x + y\*y).

isclose(...)

isclose(a, b, \*, rel\_tol=1e-09, abs\_tol=0.0) -> bool

Determine whether two floating point numbers are close in value.

rel\_tol

maximum difference for being considered "close", relative to the

magnitude of the input values

abs\_tol

maximum difference for being considered "close", regardless of the

magnitude of the input values

Return True if a is close in value to b, and False otherwise.

For the values to be considered close, the difference between them

must be smaller than at least one of the tolerances.

-inf, inf and NaN behave similarly to the IEEE 754 Standard. That

is, NaN is not close to anything, even itself. inf and -inf are

only close to themselves.

isfinite(...)

isfinite(x) -> bool

Return True if x is neither an infinity nor a NaN, and False otherwise.

isinf(...)

isinf(x) -> bool

Return True if x is a positive or negative infinity, and False otherwise.

isnan(...)

isnan(x) -> bool

Return True if x is a NaN (not a number), and False otherwise.

ldexp(...)

ldexp(x, i)

Return x \* (2\*\*i).

lgamma(...)

lgamma(x)

Natural logarithm of absolute value of Gamma function at x.

log(...)

log(x[, base])

Return the logarithm of x to the given base.

If the base not specified, returns the natural logarithm (base e) of x.

log10(...)

log10(x)

Return the base 10 logarithm of x.

log1p(...)

log1p(x)

Return the natural logarithm of 1+x (base e).

The result is computed in a way which is accurate for x near zero.

log2(...)

log2(x)

Return the base 2 logarithm of x.

modf(...)

modf(x)

Return the fractional and integer parts of x. Both results carry the sign

of x and are floats.

pow(...)

pow(x, y)

Return x\*\*y (x to the power of y).

radians(...)

radians(x)

Convert angle x from degrees to radians.

sin(...)

sin(x)

Return the sine of x (measured in radians).

sinh(...)

sinh(x)

Return the hyperbolic sine of x.

sqrt(...)

sqrt(x)

Return the square root of x.

tan(...)

tan(x)

Return the tangent of x (measured in radians).

tanh(...)

tanh(x)

Return the hyperbolic tangent of x.

trunc(...)

trunc(x:Real) -> Integral

Truncates x to the nearest Integral toward 0. Uses the \_\_trunc\_\_ magic method.

DATA

e = 2.718281828459045

inf = inf

nan = nan

pi = 3.141592653589793

tau = 6.283185307179586

FILE

(built-in)

**Rounding Numbers**

value **=** 4.35

Print (math**.**floor(value))

Out: 4

print(math**.**ceil(value))

Out:5

Print (round(value))

Out:4

**Mathematical Constants**

Print (math**.**pi);

Out:3.141592653589793

**from** math **import** pi

Print (pi);

Out:3.141592653589793

print(math**.**e);

Out:2.718281828459045

Print (math**.**tau);

Out:6.283185307179586

Print (math**.**inf);

Out:inf

Print (math**.**nan);

Out:nan

**Logarithmic Values**

**Eg:**

**Print (**math**.**e**);**

Out:2.718281828459045

*# Log Base e*

Print (math**.**log(math**.**e))

Out:1.0

*# Will produce an error if value does not exist mathmatically*

Print (math**.**log(0))

Error:

**---------------------------------------------------------------------------**

**ValueError** Traceback (most recent call last)

**<ipython-input-12-7563e0a48092>** in <module>**()**

**----> 1** math**.**log**(0)**

**ValueError**: math domain error

Print (math**.**log(10));

Out:2.302585092994046

Print (math**.**e **\*\*** 2.302585092994046);

Out: 10.000000000000002

**Custom Base**

*# math.log(x,base)*

Print (math**.**log(100,10))

out:2.0

print(10**\*\***2)

Out:100

**Trigonometrics Functions**

*# Radians*

print(math**.**sin(10));

Out:-0.5440211108893698

print(math**.**degrees(pi**/**2));

Out:90.0

print(math**.**radians(180));

Out:3.141592653589793

**Python Debugger**

You've probably used a variety of print statements to try to find errors in your code. A better way of doing this is by using Python's built-in debugger module (pdb). The pdb module implements an interactive debugging environment for Python programs. It includes features to let you pause your program, look at the values of variables, and watch program execution step-by-step, so you can understand what your program actually does and find bugs in the logic.

Eg:

Here we will create an error on purpose, trying to add a list to an integer

x **=** [1,3,4]

y **=** 2

z **=** 3

result **=** y **+** z

print(result)

result2 **=** y**+**x

print(result2)

Out:5

**---------------------------------------------------------------------------**

**TypeError** Traceback (most recent call last)

**<ipython-input-1-905e8cfe6928>** in <module>**()**

5 result **=** y **+** z

6 print**(**result**)**

**----> 7** result2 **=** y**+**x

8 print**(**result2**)**

**TypeError**: unsupported operand type(s) for +: 'int' and 'list'

looks like we get an error! Let's implement a set\_trace() using the pdb module. This will allow us to basically pause the code at the point of the trace and check if anything is wrong.

**import** pdb

x **=** [1,3,4]

y **=** 2

z **=** 3

result **=** y **+** z

print(result)

*# Set a trace using Python Debugger*

pdb**.**set\_trace()

result2 **=** y**+**x

print(result2)

Out:

5

--Return--

> <ipython-input-2-1084246755fa>(11)<module>()->None

-> pdb.set\_trace()

(Pdb) x

[1, 3, 4]

(Pdb) y

2

(Pdb) result2

\*\*\* NameError: name 'result2' is not defined

(Pdb) q

**---------------------------------------------------------------------------**

**BdbQuit** Traceback (most recent call last)

**<ipython-input-2-1084246755fa>** in <module>**()**

9

10 **# Set a trace using Python Debugger**

**---> 11** pdb**.**set\_trace**()**

12

13 result2 **=** y**+**x

**C:\Users\Marcial\Anaconda3\lib\bdb.py** in trace\_dispatch**(self, frame, event, arg)**

53 **return** self**.**dispatch\_call**(**frame**,** arg**)**

54 **if** event **==** **'return':**

**---> 55 return** self**.**dispatch\_return**(**frame**,** arg**)**

56 **if** event **==** **'exception':**

57 **return** self**.**dispatch\_exception**(**frame**,** arg**)**

**C:\Users\Marcial\Anaconda3\lib\bdb.py** in dispatch\_return**(self, frame, arg)**

97 **finally:**

98 self**.**frame\_returning **=** **None**

**---> 99 if** self**.**quitting**:** **raise** BdbQuit

100 **# The user issued a 'next' or 'until' command.**

101 **if** self**.**stopframe **is** frame **and** self**.**stoplineno **!=** **-1:**

**BdbQuit**:

**Overview of Regular Expressions**

Regular Expressions (sometimes called regex for short) allows a user to search for strings using almost any sort of rule they can come up. For example, finding all capital letters in a string, or finding a phone number in a document.

Regular expressions are notorious for their seemingly strange syntax. This strange syntax is a byproduct of their flexibility. Regular expressions have to be able to filter out any string pattern you can imagine, which is why they have a complex string pattern format

**Searching for Basic Patterns**

text **=** "The person's phone number is 408-555-1234. Call soon!"

We'll start off by trying to find out if the string "phone" is inside the text string. Now we could quickly do this with:

Print ('phone' **in** text);

Out:true

But let's show the format for regular expressions, because later on we will be searching for patterns that won't have such a simple solution.

**import** re

pattern **=** 'phone'

Print (re**.**search(pattern,text))

Out:<\_sre.SRE\_Match object; span=(13, 18), match='phone'>

pattern **=** "NOT IN TEXT"

print(re**.**search(pattern,text))

Let's take a closer look at this Match object.

pattern **=** 'phone'

match **=** re**.**search(pattern,text)

print(**match**)

Out:

<\_sre.SRE\_Match object; span=(13, 18), match='phone'>

Notice the span, there is also a start and end index information.

**Print (match.**span())

Out:(13, 18)

**Print (match.**start());

Out:13

Print (**match.**end());

Out:18

But what if the pattern occurs more than once?

text **=** "my phone is a new phone"

match **=** re**.**search("phone",text)

print(**match.**span());

Out:(3, 8)

Notice it only matches the first instance. If we wanted a list of all matches, we can use .findall() method:

matches **=** re**.**findall("phone",text)

Print (matches);

Out:['phone', 'phone']

print(len(matches));

Out:2

To get actual match objects, use the iterator:

**for** match **in** re**.**finditer("phone",text):

print(match**.**span())

Out:

(3, 8)

(18, 23)

If you wanted the actual text that matched, you can use the .group() method.

**Print (match.**group());

Out: phone

**Patterns**

**Identifiers for Characters in Patterns**

Characters such as a digit or a single string have different codes that represent them. You can use these to build up a pattern string. Notice how these make heavy use of the backwards slash \ . Because of this when defining a pattern string for regular expression we use the format:

**r'mypattern**'

placing the r in front of the string allows python to understand that the \ in the pattern string are not meant to be escape slashes.

Below you can find a table of all the possible identifiers:

**Character**

**Description**

**Example Pattern Code**

**Exammple Match**

\d

A digit

file\_\d\d

file\_25

\w

Alphanumeric

\w-\w\w\w

A-b\_1

\s

White space

a\sb\sc

a b c

\D

A non digit

\D\D\D

ABC

\W

Non-alphanumeric

\W\W\W\W\W

\*-+=)

\S

Non-whitespace

\S\S\S\S

Yoyo

For example:

text **=** "My telephone number is 408-555-1234"

phone **=** re**.**search(r'\d\d\d-\d\d\d-\d\d\d\d',text)

Print (phone**.**group());

Out:'408-555-1234'

Notice the repetition of \d. That is a bit of an annoyance, especially if we are looking for very long strings of numbers. Let's explore the possible quantifiers.

**Quantifiers**

Now that we know the special character designations, we can use them along with quantifiers to define how many we expect.

**Character**

**Description**

**Example Pattern Code**

**Exammple Match**

+

Occurs one or more times

Version \w-\w+

Version A-b1\_1

{3}

Occurs exactly 3 times

\D{3}

abc

{2,4}

Occurs 2 to 4 times

\d{2,4}

123

{3,}

Occurs 3 or more

\w{3,}

anycharacters

\\*

Occurs zero or more times

A\\*B\\*C\*

AAACC

?

Once or none

plurals?

plural

Let's rewrite our pattern using these quantifiers:

print(re**.**search(r'\d{3}-\d{3}-\d{4}',text));

Out:<\_sre.SRE\_Match object; span=(23, 35), match='408-555-1234'>

**Groups**

What if we wanted to do two tasks, find phone numbers, but also be able to quickly extract their area code (the first three digits). We can use groups for any general task that involves grouping together regular expressions (so that we can later break them down).

Using the phone number example, we can separate groups of regular expressions using parenthesis:

phone\_pattern **=** re**.**compile(r'(\d{3})-(\d{3})-(\d{4})')

results **=** re**.**search(phone\_pattern,text)

*# The entire result*

print(results**.**group());

Out:'408-555-1234'

*# Can then also call by group position.*

*# remember groups were separated by parenthesis ()*

*# Something to note is that group ordering starts at 1. Passing in 0 returns everything*

print(results**.**group(1));

Out: 408

Print (results**.**group(2));

Out: 555

Print (results**.**group(3));

Out: 1234

*# We only had three groups of parenthesis*

Print (results**.**group(4))

Error:

**---------------------------------------------------------------------------**

**IndexError** Traceback (most recent call last)

**<ipython-input-32-866de7a94a57>** in <module>**()**

1 **# We only had three groups of parenthesis**

**----> 2** results**.**group**(4)**

**IndexError**: no such group

**Additional Regex Syntax**

**Or operator |**

Use the pipe operator to have an **or** statment. For example

print(re**.**search(r"man|woman","This man was here.”));

Out:<\_sre.SRE\_Match object; span=(5, 8), match='man'>

print(re**.**search(r"man|woman","This woman was here.”));

Out:<\_sre.SRE\_Match object; span=(5, 10), match='woman'>

**The Wildcard Character**

Use a "wildcard" as a placement that will match any character placed there. You can use a simple period **.** for this. For example:

print(re**.**findall(r".at","The cat in the hat sat here."));

Out:

['cat', 'hat', 'sat']

print(re**.**findall(r".at","The bat went splat”));

Out:

['bat', 'lat']

Notice how we only matched the first 3 letters, that is because we need a **.** for each wildcard letter. Or use the quantifiers described above to set its own rules.

Print (re**.**findall(r"...at","The bat went splat”));

Out:['e bat', 'splat']

However this still leads the problem to grabbing more beforehand. Really we only want words that end with "at".

*# One or more non-whitespace that ends with 'at'*

print(re**.**findall(r'\S+at',"The bat went splat”));

Out:['bat', 'splat']

**Starts with and Ends With**

We can use the **^** to signal starts with, and the **$** to signal ends with:

*# Ends with a number*

Print(re**.**findall(r'\d$','This ends with a number 2’));

Out:['2']

*# Starts with a number*

print(re**.**findall(r'^\d','1 is the loneliest number.’));

Out:

['1']

**Exclusion**

To exclude characters, we can use the **^** symbol in conjunction with a set of brackets **[]**. Anything inside the brackets is excluded. For example:

phrase **=** "there are 3 numbers 34 inside 5 this sentence."

print(re**.**findall(r'[^\d]',phrase));

Out:

['t',

'h',

'e',

'r',

'e',

' ',

'a',

'r',

'e',

' ',

' ',

'n',

'u',

'm',

'b',

'e',

'r',

's',

' ',

' ',

'i',

'n',

's',

'i',

'd',

'e',

' ',

' ',

't',

'h',

'i',

's',

' ',

's',

'e',

'n',

't',

'e',

'n',

'c',

'e',

'.']

To get the words back together, use a + sign

print(re**.**findall(r'[^\d]+',phrase));

Out:['there are ', ' numbers ', ' inside ', ' this sentence.']

We can use this to remove punctuation from a sentence.

test\_phrase **=** 'This is a string! But it has punctuation. How can we remove it?'

Print (re**.**findall('[^!.? ]+',test\_phrase));

Out:

['This',

'is',

'a',

'string',

'But',

'it',

'has',

'punctuation',

'How',

'can',

'we',

'remove',

'it']

clean **=** ' '**.**join(re**.**findall('[^!.? ]+',test\_phrase))

print(clean);

Out:'This is a string But it has punctuation How can we remove it'

**Brackets for Grouping**

As we showed above we can use brackets to group together options, for example if we wanted to find hyphenated words:

text **=** 'Only find the hypen-words in this sentence. But you do not know how long-ish they are'

print(re**.**findall(r'[\w]+-[\w]+',text));

**Out:**['hypen-words', 'long-ish']

**Parenthesis for Multiple Options**

If we have multiple options for matching, we can use parenthesis to list out these options. For Example:

*# Find words that start with cat and end with one of these options: 'fish','nap', or 'claw'*

text **=** 'Hello, would you like some catfish?'

texttwo **=** "Hello, would you like to take a catnap?"

textthree **=** "Hello, have you seen this caterpillar?"

**print(**re**.**search(r'cat(fish|nap|claw)',text)**);**

Out:<\_sre.SRE\_Match object; span=(27, 34), match='catfish'>

print(re**.**search(r'cat(fish|nap|claw)',texttwo));

Out:<\_sre.SRE\_Match object; span=(32, 38), match='catnap'>

*# None returned*

print(re**.**search(r'cat(fish|nap|claw)',textthree));

**Timing your code**

Sometimes it's important to know how long your code is taking to run, or at least know if a particular line of code is slowing down your entire project. Python has a built-in timing module to do this.

Example Function or Script

Here we have two functions that do the same thing, but in different ways. How can we tell which one is more efficient? Let's time it!

**def** func\_one(n):

'''

Given a number n, returns a list of string integers

['0','1','2',...'n]

'''

**return** [str(num) **for** num **in** range(n)]

Print (func\_one(10))

Out:['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

**def** func\_two(n):

'''

Given a number n, returns a list of string integers

['0','1','2',...'n]

'''

**return** list(map(str,range(n)))

Print (func\_two(10));

Out:['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

**Timing Start and Stop**

We can try using the time module to simply calculate the elapsed time for the code. Keep in mind, due to the time module's precision, the code needs to take **at least** 0.1 seconds to complete.

**import** time

*# STEP 1: Get start time*

start\_time **=** time**.**time()

*# Step 2: Run your code you want to time*

result **=** func\_one(1000000)

*# Step 3: Calculate total time elapsed*

end\_time **=** time**.**time() **-** start\_time

Print (end\_time)

Out: 0.18550348281860352

*# STEP 1: Get start time*

start\_time **=** time**.**time()

*# Step 2: Run your code you want to time*

result **=** func\_two(1000000)

*# Step 3: Calculate total time elapsed*

end\_time **=** time**.**time() **-** start\_time

print(end\_time);

Out:0.1496279239654541

**Timeit Module**

What if we have two blocks of code that are quite fast, the difference from the time.time() method may not be enough to tell which is fater. In this case, we can use the timeit module.

The timeit module takes in two strings, a statement (stmt) and a setup. It then runs the setup code and runs the stmt code some n number of times and reports back average length of time it took.

**import** timeit

The setup (anything that needs to be defined beforehand, such as def functions.)

setup **=** '''

def func\_one(n):

return [str(num) for num in range(n)]

'''

stmt **=** 'func\_one(100)'

print(timeit**.**timeit(stmt,setup,number**=**100000))

Out:1.3161248000000114

**Web Scraping**

**Web scraping** is a general term for techniques involving automating the gathering of data from a website.

Inorder to web scrape with python we need to understand the basic concepts of ow a website works.

When a browser loads a website, the user gets to see what is known as the “front-end”of the website.

Main things we need to understand

• Rules of web scraping

• limitations of web scraping

• basic html and css

**Rules of web scraping**

1. Always be respectful and try to get premission to scrape, do not bombard a website with scraping requests, otherwise your IP address may be blocked!

2. Be aware that websites change often, meaning your code could go from working to totally broken from one day to the next.

3. Pretty much every web scraping project of interest is a unique and custom job, so try your best to generalize the skills learned here.

**limitations of web scraping**

1.in general every website is unique, which means every web scraping script is unique.

2.a slight change or update to a website may completely break your web scraping script.

When viewing a website, the browser doesn’t show you all the source code behind the website,instead it shows you the html and some css and js that the website sends to your browser

Html is used to create the basic structure and content of a webpage .

css is used for the design and style of a webpage ,where elements are placed and how it looks.

Javascript is used to define the interactive elements of a webpage

For effective basic web scraping we only need to have a basic understanding of html and css.

Python can view these html and css elements programmatically, and then extract information from the website.

Html is hypertext mark up languages and is present on every website on the Internet.

You can right click on a website and select “view page source ”to get an example.

Eg:

<!Doctype html>

<html>

<head>

<title>

Title of the browser </title >

</head>

<body>

<h1>

Website header</h1>

<p>paragraphs </p>

</body>

</html>

Css stands for cascading style sheets.

Css gives “style” to a website, such as changing colours and fonts.

Css uses tags to define what html elements will be styled.

<!Doctype html>

<html>

<head>

<link rel =“stylesheet” href=“styles.css”>

<title>

Title of the browser </title >

</head>

<body>

<h1>

Website header</h1>

<p id=‘para2’ >paragraphs </p>

</body>

</html>

Eg of the style.css file:

#para2{

Color:red;

}

Note:

Html contains the information

Css contains the styling

We can use html and css tags to locate specific information on a page.

To webscrape with python we can use the beautifulsoup and requests libraries.

These are external libraries outside of python so you need to install them with either conda or pip at your command line.

Directly at your command line use :

Pip install requests

Pip install lxml

Pip install bs4

Install the necessary libraries

Explore how to inspect elements and view source of a webpage.

Note:we will suggest you use chrome so you can follow along exactly as we do,but these tools are available in all major browsers.

Eg:

Import requests

Result =requests.get(“htttp:///www.example.com”)

type(Result )

Out:requests.models.Response

result.text

Out:

Import bs4

Soup = bs4.BeautifulSoup(result.text,”lxml ”)

Soup

Out:

Soup.select(‘p’)

Out:

Soup.select(‘p’)

Out:

Soup.select(’title )[0].getText()

Out:

Site\_paragraphs=soup.select(“p’)

site\_paragraph[0].getText()

Out:

Syntax match results

Soup.select(‘div’) all elements will ‘div’ tag

Soup.select(‘#some\_id) elements containing id=’some\_id’

Soup.select(‘.some\_class’) elements containing class =“some\_class”

Soup.select(‘div span’) any elements named span within a div element

Soup.(‘div>span’) any elements named span directly within a div element ,with nothing in

Between

res=requests.get(“htpps://en.wikkipedia.org/wiki/Grace\_Hopper”)

soup=bs4.beautifulSoup(res.text,”lxml”)

Soup

Out:

Soup.select(‘.toctext’)

Out:

type(Soup.select(‘.toctext’)[0])

Out:bs4.element.Tag

first\_item=Soup.select(‘.toctext’)[0]

first\_item.text

Out:

**Grabbing an image**

Images on a website typically have their own url link(ending in .jpg or .png)

Beautiful soup can scan a page ,locate the <img> tags and grab these urls.

Then we can download the urls as images and write them to the computer.

Note: you should always check copy right permissions before downloading and using an image from a website.

res=requests.get(“https://en.wikipedia.org/wiki/deep\_blue\_(chess\_computer)”)

soup=bs4.BeautifulSop(res.text,’lxml)

Soup

Soup.select(‘img’)[0]

Out:

Soup.select(‘.thumbimage’)

Out:

Computer =soup.select(‘. thumbimage’)[0]

Computer[’src’]

Out:

<img src= ”out”>

image\_link=requests.get(“out”)

image\_link.content

f=open(‘mycomputer .jpg’,’wb’)

F.write(image\_link.content)

Out:16806

f.close()

Pwd

#GOAL:get title of every book with a 2 star rating

Import requests

Import bs4

‘http://toscrape.com/catalouge/page-2.html’

Out:

‘http://toscrape.com/catalouge/page-3.html’

Out:

base\_url=“http://toscrape.com/catalouge/page-{}.html”

res=requests.get(base\_url.format(1))

Soup=bs4.Beautifulsoup(res.text,’lxml’)

Products =soup.select(“.product\_pod ”)

Example =products[0]

‘star-rating Three’ in str (example)

Out: True

example.select(“.star-rating.two”)

Out:[]

example.select(‘a’)[1][‘title’]

Out:

#we can check if something is 2 stars (string call in, example.select(rating))

#example.select(‘a’)[1][‘title ’] to grab the book title

two\_star\_titles=[]

for n in range(1,51):

scrape\_url=base\_url.format(n)

res=requests.get(scrape\_url)

Soup =bs4.BeautifulSoup(res.text,’lxml’)

books=soup.select(“.product\_pod”)

For book in books:

If ‘.star-rating.two’ in str(book)

If len(book.select(‘.star-rating.two’)) ! = 0:

book\_title=book.select(‘a’)[1][‘title’]

two\_star\_titles.append(book\_title)

**Working with images with python**

**We ‘ll install it with pip install pillow and then show you how to open and save image files and interact with images.**

**We will use the pillow library for this, which is a fork of the PIL (python imaging library)with easy to use function calls.**

**Install :**

**Pip install pillow**

**Eg:**

**Opening Images**

from PIL import Image

ac=Image.open(‘example.jpg’)

type(ac)

Out:PIL.JpegImagePluginImageFile

Print (ac)

Out:

**Image Information**

Print (ac.size)

Out:(1993,1257)

print(ac.filename)

Out:example.jpg

Print (ac.format\_description)

Out:JPEG (ISO 10918)

**Cropping Images**

To crop images (that is grab a sub section) you can use the crop() method on the image object. The crop() method returns a rectangular region from this image. The box is a 4-tuple defining the left, upper, right, and lower pixel coordinate.

Note! If you take a look at the documentation string, it says the tuple you pass in is defined as (x,y,w,h). These variables can be a bit decieving. Its not really a height or width that is being passed, but instead the end coordinates of your width and height.

All the coordinates of box (x, y, w, h) are measured from the top left corner of the image. Again, all 4 of these values are coordinates!

Print (ac.crop(0,0,100,100) )

Out:

Eg:

pencils=Image.open(‘pencils.jpg’)

print(pencils)

Out:

Print (pencils.size)

Out:(1950,1300)

x=0

Y=0

w=1950/3

h=1300/10

**Copying and Pasting Images**

We can create copies with the copy() method and paste images on top of others with the paste() method.

print(Pencils.crop((x,y,w,h) ))

Out:

print(pencils.paste(im=pencils,box=(796,0)))

Pencils

Out:

Pencils.rotate(90)

**Resizing**

You can use the resize() method to resize an image

mac**.**size

**Out:**(1993, 1257)

h,w **=** mac**.**size

**Out:**

new\_h **=** int(h**/**3)

new\_w **=** int(w**/**3)

*# Note this is not permanent change*

*# for permanent change, do a reassignment*

*# e.g. mac = mac.resize((100,100))*

mac**.**resize((new\_h,new\_w))

Can also stretch and squeeze

mac**.**resize((3000,500))

**Rotating Images**

You can rotate images by specifying the amount of degrees to rotate on the rotate() method. The original dimensions will be kept and "filled" in with black. You can optionally pass in the expand parameter to fill the new rotated image to the old dimensions.

pencils**.**rotate(90)

Out:

pencils**.**rotate(90,expand**=True**)

Out:

Notice what happens when we rotate by 120.

*# The image is cut off*

pencils**.**rotate(120)

Out:

*#pencils.rotate(120,expand=True)*

**Color transparency**

We can add an alpha value (RGBA stands for RED,Green,Blue, Alpha) where values can go from 0 to 255. If Alpha is 0 the image is completely transparent, if it is 255 then its completely opaque.

**RGBA-red,green,blue,alpha**

We can adjust image alpha values with the putalpha() method:

red=Image.open(‘red\_color.jpg’)

print(red)

Out:

blue=Image.open(‘blue\_color.png’)

print(blue)

Out:

print(blue.putalpha(0))

Out:

print(blue.putalpha(12))

Out:

print(red.putalpha(128))

Out:

Blue.paste(im=red,box=(0,0),mask=red)

*# Get back an image that is more purple.*

print(blue)

Out:

**Saving Images**

Let's save this updated "blue" image as 'purple.png' in this folder.

Blue.save(“purple.png”) or Blue.save(“/c/users/purple.png”)

purple=Image.open(“purple.png”)

Print (purple )

Out:

**PDFs and spreadsheets with python**

Python has the ability to work with PDF files and spreadsheet files.

In this we will explore libraries that allow us to interact with these files.

CSV stands for comma separated variables and is a very common output for spreadsheet programs.all CSV files are plain text, contain alphanumeric characters, and structure the data contained within them in a tabular form. Don't confuse Excel Files with csv files, while csv files are formatted very similarly to excel files, they don't have data types for their values, they are all strings with no font or color. They also don't have worksheets the way an excel file does.

Files in the CSV format are generally used to exchange data, usually when there's a large amount, between different applications. Database programs, analytical software, and other applications that store massive amounts of information (like contacts and customer data), will usually support the CSV format.

Eg:

Name,hours,rate

David,20,15

Claire,40,20

Note that while it’s possible to export Excel files and google spreadsheet to .csv files, it only exports the information.

Things like formulas, images ,and macros can be within a .csv file.

Simply put, a .csv file only contains the raw data from the spread sheet.

We will work with the built-in csv module for python ,which will allow us to grab columns, rows and values from a .csv file as well as write to a .csv file.

Keep in mind ,this is a very popular space for outside libraries, which you may want to explore.

Other libraries to consider:

1. Pandas

• Full data analysis library, can work with almost any tabular data type.

• Runs visualisations and analysis.

2.Openpyxl

• Designed specifically for excel files.

• Retains a lot of excel specific functionality

• Supports excel formulas.

• Python-excel.org tracks various other excel based python libraries.

3.Google sheets python api

• Direct python interface for working with google spread sheets

• Allows you to directly make changes to the spread sheets hosted online.

• More complex syntax ,but available in many programming languages.

The common factor between all of these spreadsheet programs is that they can always exports to .csv.

**Location**

inp:Pwd

**Reading CSV Files**

Import csv

#open the file

data=open(‘example.csv’,encoding=‘utf-8’)

print(data)

Out:<\_io.TextIOWrapper name='example.csv' mode='r' encoding='cp1252'>

**Encoding**

Often csv files may contain characters that you can't interpret with standard python, this could be something like an **@** symbol, or even foreign characters. Let's view an example of this sort of error (its pretty common, so its important to go over).

#csv.reader

csv\_data **=** csv**.**reader(data)

#reformat it into a python object list of lists

data\_lines=list(csv\_data)

Print (data\_lines)

Out:

Print (len(data\_lines))

Out:

For line in data\_lines[:5]:

Print (line )

Out:

Print (data\_lines[10])

Out:

Print (data\_lines[10][3])

Out:

all\_emails =[ ]

For line in data\_lines[1:]:

all\_emails.append(line[3])

Out:

print(all\_emails)

#all\_emails

data\_lines[10]

Out:

full\_names **=** []

**for** line **in** data\_lines[1:15]:

full\_names**.**append(line[1]**+**' '**+**line[2])

Print (full\_names)

Out:['Joseph Zaniolini',

'Freida Drillingcourt',

'Nanni Herity',

'Orazio Frayling',

'Julianne Murrison',

'Lucy Gamet',

'Dyana Howatt',

'Kassey Herion',

'Chrissy Hedworth',

'Hyatt Gasquoine',

'Felicdad Tarr',

'Andrew Bath',

'Lucais Chastang',

'Car Cerie']

**Writing to CSV Files**

We can also write csv files, either new ones or add on to existing ones.

*# newline controls how universal newlines works (it only applies to text*

*# mode). It can be None, '', '\n', '\r', and '\r\n'.*

file\_to\_output **=** open('to\_save\_file.csv','w',newline**=**'')

csv\_writer **=** csv**.**writer(file\_to\_output,delimiter**=**',')

csv\_writer**.**writerow(['a','b','c'])

Out:7

csv\_writer**.**writerows([['1','2','3'],['4','5','6']])

file\_to\_output**.**close()

**Existing File**

f **=** open('to\_save\_file.csv','a',newline**=**'')

csv\_writer **=** csv**.**writer(f)

csv\_writer**.**writerow(['new','new','new'])

Out:13

f**.**close()

**Working with PyPDF2**

Let's being showing the basics of the PyPDF2 library.

**!**pip install PyPDF2

*# note the capitalization*

**import** PyPDF2

**Reading PDFs**

Similar to the csv library, we open a pdf, then create a reader object for it. Notice how we use the binary method of reading , 'rb', instead of just 'r'.

*# Notice we read it as a binary with 'rb'*

f **=** open('Working\_Business\_Proposal.pdf','rb')

pdf\_reader **=** PyPDF2**.**PdfReader(f)

len(pdf\_reader**.**pages)

Out:5

page\_number **=** 0

page\_one **=** pdf\_reader**.**pages[0]

We can then extract the text:

page\_one\_text **=** page\_one**.**extract\_text()

print(page\_one\_text)

f**.**close()

**Adding to PDFs**

We can not write to PDFs using Python because of the differences between the single string type of Python, and the variety of fonts, placements, and other parameters that a PDF could have.

What we can do is copy pages and append pages to the end.

f **=** open('Working\_Business\_Proposal.pdf','rb')

pdf\_reader **=** PyPDF2**.**PdfReader(f)

page\_number **=** 0

page\_one **=** pdf\_reader**.**pages[0]

pdf\_writer **=** PyPDF2**.**PdfWriter()

pdf\_writer**.**add\_page(page\_one);

pdf\_output **=** open("Some\_New\_Doc.pdf","wb")

pdf\_writer**.**write(pdf\_output)

Out:(False, <\_io.BufferedWriter name='Some\_New\_Doc.pdf'>)

f**.**close()

**Simple Example**

Let's try to grab all the text from this PDF file:

f **=** open('Working\_Business\_Proposal.pdf','rb')

*# List of every page's text.*

*# The index will correspond to the page number.*

pdf\_text **=** []

pdf\_reader **=** PyPDF2**.**PdfReader(f)

**for** p **in** range(len(pdf\_reader**.**pages)):

page **=** pdf\_reader**.**pages[0]

pdf\_text**.**append(page**.**extract\_text())

Print (pdf\_text)

Out:

**Overview of Sending Emails**

The smtplib library allows you to manually go through the steps of creating and sending an email in Python:

**import** smtplib

Create an SMTP object for a server. Here are the main Server Domain Name for the top email services. If you don't see your email server here, you may need to do a quick Google Search to see if there SMTP server domain name is available:

**Provider**

**SMTP server domain name**

Gmail (will need App Password)

smtp.gmail.com

Yahoo Mail

smtp.mail.yahoo.com

Outlook.com/Hotmail.com

smtp-mail.outlook.com

AT&T

smpt.mail.att.net (Use port 465)

Verizon

smtp.verizon.net (Use port 465)

Comcast

smtp.comcast.net

Next is to create an STMP object that can make the method calls to log you in to your email in order to send messages. Notice how also specify a port number. If the number 587 does not work on your computer, try using 465 instead. Keep in mind, a firewall or antivirus may prevent Python from opening up this port, so you may need to disable it on your computer.

smtp\_object **=** smtplib**.**SMTP('smtp.gmail.com',587)

Next we run the ehlo() command which "greets" the server and establishes the connection. This method call should be done directly after creating the object. Calling it after other methods may result in errors in connecting later on. The first item in the tuple that is returned should be 250, indicating a successful connection.

smtp\_object**.**ehlo()

Out:

(250,

b'smtp.gmail.com at your service, [47.143.81.4]\nSIZE 35882577\n8BITMIME\nSTARTTLS\nENHANCEDSTATUSCODES\nPIPELINING\nCHUNKING\nSMTPUTF8')

When using the 587 port, this means you are using TLS encryption, which you need to initiate by running the starttls() command. If you are using port 465, this means you are using SSL and you can skip this step.

smtp\_object**.**starttls()

Out:(220, b'2.0.0 Ready to start TLS')

Now its time to set up the email and the passwords. You should never save the raw string of your password or email in a script, because anyone that sees this script will then be able to see you email and password! Instead you should use input() to get that information. If you also don't want your password to be visible when typing it in, you can use the built-in **getpass** library that will hide your password as you type it in, either with asterisks or by just keeping it invisible.

*# For hidden passwords*

**import** getpass

result **=** getpass**.**getpass("Type something here and it will be hidden: ")

Out:Type something here and it will be hidden: ········

*# Just keep in mind that its still visible as an object internally:*

result

Out: a

*# Or just use input()*

input("Enter your password")

out:Enter your passwords

's'

**Note for Gmail Users, you need to generate an app password instead of your normal email password. This also requires enabling 2-step authentication. Follow the instructions here to set-up 2-Step Factor Authentication as well as App Password Generation:https://support.google.com/accounts/answer/185833?hl=en/. Set-up 2 Factor Authentication, then create the App Password, choose Mail as the App and give it any name you want. This will output a 16 letter password for you. Pass in this password as your login password for the smtp.**

email **=** getpass**.**getpass("Enter your email: ")

password **=** getpass**.**getpass("Enter your password: ")

smtp\_object**.**login(email,password)

Now we can send an email using the .sendmail() method.

from\_address **=** getpass**.**getpass("Enter your email: ")

to\_address **=** getpass**.**getpass("Enter the email of the recipient: ")

subject **=** input("Enter the subject line: ")

message **=** input("Type out the message you want to send: ")

msg **=** "Subject: " **+** subject **+** '\n' **+** message

smtp\_object**.**sendmail(from\_address,to\_address,msg)

Out:Enter your email: ········

Enter the email of the recipient: ········

Enter the subject line: This is a test

Type out the message you want to send: Here is the message.

Out:{}

If you get back an empty dictionary, then the sending was successful.

You can then close your session with the .quit() method.

smtp\_object**.**quit()

Out:(221, b'2.0.0 closing connection j1sm22376227pgq.33 - gsmtp')

**Overview of Received Emails**

let's explore how we can read and search recieved emails. To do we will use the built-in imaplib library. We will also use the built in email library for parsing through the recieved emails.

**import** imaplib

M **=** imaplib**.**IMAP4\_SSL('imap.gmail.com')

**import** getpass

user **=** input("Enter your email: ")

*# Remember , you may need an app password if you are a gmail user*

*#*

password **=** getpass**.**getpass("Enter your password: ")

Out:Enter your password: ········

M**.**login(user,password)

M**.**list()

Out:

('OK',

[b'(\\HasNoChildren) "/" "INBOX"',

b'(\\HasNoChildren) "/" "Personal"',

b'(\\HasNoChildren) "/" "Receipts"',

b'(\\HasNoChildren) "/" "Sent"',

b'(\\HasNoChildren) "/" "Trash"',

b'(\\HasNoChildren) "/" "Travel"',

b'(\\HasNoChildren) "/" "Work"',

b'(\\HasChildren \\Noselect) "/" "[Gmail]"',

b'(\\All \\HasNoChildren) "/" "[Gmail]/All Mail"',

b'(\\Drafts \\HasNoChildren) "/" "[Gmail]/Drafts"',

b'(\\HasNoChildren \\Important) "/" "[Gmail]/Important"',

b'(\\HasNoChildren \\Sent) "/" "[Gmail]/Sent Mail"',

b'(\\HasNoChildren \\Junk) "/" "[Gmail]/Spam"',

b'(\\Flagged \\HasNoChildren) "/" "[Gmail]/Starred"',

b'(\\HasNoChildren \\Trash) "/" "[Gmail]/Trash"'])

*# Connect to your inbox*

M**.**select("inbox")

Out:('OK', [b'28297'])

**Searching Mail**

Now that we have connected to our mail, we should be able to search for it using the specialized syntax of IMAP. Here are the different search keys you can use:

<tr>

<td>'BEFORE date'</td>

<td>

Returns all messages before the date. Date must be formatted as 01-Nov-2000.

</td>

</tr>

<tr>

<td>'ON date'</td>

<td>

Returns all messages on the date. Date must be formatted as 01-Nov-2000.

</td>

</tr>

<tr>

<td>'SINCE date'</td>

<td>

Returns all messages after the date. Date must be formatted as 01-Nov-2000.

</td>

</tr>

<tr>

<td>'FROM some\_string '</td>

<td>

Returns all from the sender in the string. String can be an email, for example 'FROM user@example.com' or just a string that may appear in the email, "FROM example"

</td>

</tr>

<tr>

<td>'TO some\_string'</td>

<td>

Returns all outgoing email to the email in the string. String can be an email, for example 'FROM user@example.com' or just a string that may appear in the email, "FROM example"

</td>

</tr>

<tr>

<td>'CC some\_string' and/or 'BCC some\_string'</td>

<td>

Returns all messages in your email folder. Often there are size limits from imaplib.

To change these use imaplib.\_MAXLINE = 100 , where 100 is whatever you want the limit to be.

</td>

</tr>

<tr>

<td>'SUBJECT string','BODY string','TEXT "string with spaces"'</td>

<td>

Returns all messages with the subject string or the string in the body of the email. If the string you are searching for has spaces in it, wrap it in double quotes.

</td>

</tr>

<tr>

<td>'SEEN', 'UNSEEN'</td>

<td>

Returns all messages that have been seen or unseen. (Also known as read or unread)

</td>

</tr>

<tr>

<td>'ANSWERED', 'UNANSWERED'</td>

<td>

Returns all messages that have been replied to or unreplied to.

</td>

</tr>

<tr>

<td>'DELETED', 'UNDELETED'</td>

<td>

Returns all messages that have been deleted or that have not been deleted.

</td>

</tr>

**Keywrod**

**Definition**

'ALL'

Returns all messages in your email folder. Often there are size limits from imaplib. To change these use imaplib.\_MAXLINE = 100 , where 100 is whatever you want the limit to be.

You can also use the logical operators AND and OR to combine the above statements. Check out the full list of search keys here: http://www.4d.com/docs/CMU/CMU88864.HTM.

Please note that some IMAP server providers for different email services will have slightly different syntax. You may need to experiment to get the results you want.

Now we can search our mail for any term we want.

*# Use if you get an error saying limit was reached*

imaplib**.**\_MAXLINE **=** 10000000

Send yourself a test email with the subject line:

this is a test email for python

Or some other uniquely identifying string.

We will now need to reconnect to our imap server.

*# Restart your kernel and run the following:*

**import** imaplib

**import** getpass

M **=** imaplib**.**IMAP4\_SSL('imap.gmail.com')

user **=** input("Enter your email: ")

password **=** getpass**.**getpass("Enter your password: ")

M**.**login(user,password)

*# Connect to your inbox*

M**.**select("inbox")

Out:('OK', [b'28299'])

Let's now search and confirm if it is there:

typ ,data **=** M**.**search(**None**,'SUBJECT "this is a test email for python"')

We can now save what it has returned:

typ

Out: OK

data

Out:[b'28298']

The data will be a list of unique ids.

*# typ, data = M.fetch(data[0],"(RFC822)")*

result, email\_data **=** M**.**fetch(data[0],"(RFC822)")

raw\_email **=** email\_data[0][1]

raw\_email\_string **=** raw\_email**.**decode('utf-8')

We can use the built in email library to help parse this raw string.

**import** email

email\_message **=** email**.**message\_from\_string(raw\_email\_string)

**for** part **in** email\_message**.**walk():

**if** part**.**get\_content\_type() **==** "text/plain":

body **=** part**.**get\_payload(decode**=True**)

print(body)

Out:b'This is a test to see if the python search worked.\r\n'

**Advanced Numbers**

In this lecture we will learn about a few more representations of numbers in Python.

**Hexadecimal**

Using the function hex() you can convert numbers into a hexadecimal format:

hex(246)

Out:'0xf6'

hex(512)

Out:'0x200'

**Binary**

Using the function bin() you can convert numbers into their binary format.

bin(1234)

Out:0b10011010010

bin(128)

Out: 0b10000000

bin(512)

Out:0b1000000000

**Exponentials**

The function pow() takes two arguments, equivalent to x^y. With three arguments it is equivalent to (x^y)%z, but may be more efficient for long integers.

pow(3,4)

Out:81

pow(3,4,5)

Out:1

**Absolute Value**

The function abs() returns the absolute value of a number. The argument may be an integer or a floating point number. If the argument is a complex number, its magnitude is returned.

abs(**-**3.14)

Out: 3.14

abs(3)

Out:3

**Round**

The function round() will round a number to a given precision in decimal digits (default 0 digits). It does not convert integers to floats.

round(3,2)

Out:3

round(395,**-**2)

Out:400

round(3.1415926535,2)

Out:3.14

**Advanced Strings**

String objects have a variety of methods we can use to save time and add functionality. Let's explore some of them in this lecture:

s **=** 'hello world'

**Changing case**

We can use methods to capitalize the first word of a string, or change the case of the entire string.

*# Capitalize first word in string*

s**.**capitalize()

Out:'Hello world'

s**.**upper()

Out:'HELLO WORLD'

s**.**lower()

Out:'hello world'

Remember, strings are immutable. None of the above methods change the string in place, they only return modified copies of the original string.

Print (s)

Out:'hello world'

To change a string requires reassignment:

s **=** s**.**upper()

Print (s)

Out:'HELLO WORLD'

s **=** s**.**lower()

Print (s)

Out:'hello world'

**Location and Counting**

s**.**count('o') *# returns the number of occurrences, without overlap*

Out:2

s**.**find('o') *# returns the starting index position of the first occurence*

Out:4

**Formatting**

The center() method allows you to place your string 'centered' between a provided string with a certain length. Personally, I've never actually used this in code as it seems pretty esoteric...

s**.**center(20,'z')

Out:'zzzzhello worldzzzzz'

The expandtabs() method will expand tab notations \t into spaces:

'hello\thi'**.**expandtabs()

Out:'hello hi'

**is check methods**

These various methods below check if the string is some case. Let's explore them:

s **=** 'hello'

isalnum() will return True if all characters in **s** are alphanumeric

s**.**isalnum()

out:True

isalpha() will return True if all characters in **s** are alphabetic

s**.**isalpha()

Out:True

islower() will return True if all cased characters in **s** are lowercase and there is at least one cased character in **s**, False otherwise.

s**.**islower()

Out:True

isspace() will return True if all characters in **s** are whitespace.

s**.**isspace()

Out:False

istitle() will return True if **s** is a title cased string and there is at least one character in **s**, i.e. uppercase characters may only follow uncased characters and lowercase characters only cased ones. It returns False otherwise.

s**.**istitle()

Out:False

isupper() will return True if all cased characters in **s** are uppercase and there is at least one cased character in **s**, False otherwise.

s**.**isupper()

Out: False

Another method is endswith() which is essentially the same as a boolean check on s[-1]

s**.**endswith('o')

Out:True

**Built-in Reg. Expressions**

Strings have some built-in methods that can resemble regular expression operations. We can use split() to split the string at a certain element and return a list of the results. We can use partition() to return a tuple that includes the first occurrence of the separator sandwiched between the first half and the end half.

s**.**split('e')

Out:['h', 'llo']

s**.**partition('l')

Out:('he', 'l', 'lo')

**Advanced Sets**

s **=** set()

**add**

add elements to a set. Remember, a set won't duplicate elements; it will only present them once (that's why it's called a set!)

s**.**add(1)

s**.**add(2)

Print (s)

Out:{1, 2}

**clear**

removes all elements from the set

s**.**clear()

print(s)

Out:set()

**copy**

returns a copy of the set. Note it is a copy, so changes to the original don't effect the copy.

s **=** {1,2,3}

sc **=** s**.**copy()

Print (sc)

Out:{1, 2, 3}

Print (s)

Out:{1, 2, 3}

s**.**add(4)

Print (s)

Out:{1, 2, 3, 4}

Print (sc)

Out:{1, 2, 3}

**difference**

difference returns the difference of two or more sets. The syntax is:

set1.difference(set2)

For example:

s**.**difference(sc)

Out:{4}

**difference\_update**

difference\_update syntax is:

set1.difference\_update(set2)

the method returns set1 after removing elements found in set2

s1 **=** {1,2,3}

s2 **=** {1,4,5}

s1**.**difference\_update(s2)

Print (s1)

Out:{2, 3}

**discard**

Removes an element from a set if it is a member. If the element is not a member, do nothing.

Print (s)

Out:{1, 2, 3, 4}

s**.**discard(2)

Print (s)

Out:{1, 3, 4}

**intersection and intersection\_update**

Returns the intersection of two or more sets as a new set.(i.e. elements that are common to all of the sets.)

s1 **=** {1,2,3}

s2 **=** {1,2,4}

s1**.**intersection(s2)

Out:{1, 2}

Print (s1)

Out:{1, 2, 3}

intersection\_update will update a set with the intersection of itself and another.

s1**.**intersection\_update(s2)

Print (s1)

Out:{1, 2}

**isdisjoint**

This method will return True if two sets have a null intersection.

Eg:

s1 **=** {1,2}

s2 **=** {1,2,4}

s3 **=** {5}

s1**.**isdisjoint(s2)

out:False

s1**.**isdisjoint(s3)

Out: True

**issubset**

This method reports whether another set contains this set.

eg:

Print (s1)

Out:{1, 2}

Print (s2)

Out:{1, 2, 4}

s1**.**issubset(s2)

Out: True

**issuperset**

This method will report whether this set contains another set.

s2**.**issuperset(s1)

Out: True

s1**.**issuperset(s2)

Out:False

**symmetric\_difference and symmetric\_update**

Return the symmetric difference of two sets as a new set.(i.e. all elements that are in exactly one of the sets.)

print(s1)

Out:{1, 2}

Print (s2)

Out:{1, 2, 4}

s1**.**symmetric\_difference(s2)

Out:{4}

**union**

Returns the union of two sets (i.e. all elements that are in either set.)

s1**.**union(s2)

Out:{1, 2, 4}

**update**

Update a set with the union of itself and others.

s1**.**update(s2)

Print (s1)

Out:{1, 2, 4}

**Advanced Dictionaries**

**Dictionary Comprehensions**

Just like List Comprehensions, Dictionary Data Types also support their own version of comprehension for quick creation. It is not as commonly used as List Comprehensions, but the syntax is:

{x:x**\*\***2 **for** x **in** range(10)}

Out:{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}

One of the reasons it is not as common is the difficulty in structuring key names that are not based off the values.

**Iteration over keys, values, and items**

Dictionaries can be iterated over using the keys(), values() and items() methods. For example:

d **=** {'k1':1,'k2':2}

**for** k **in** d**.**keys():

print(k)

Out:k1

k2

**for** v **in** d**.**values():

print(v)

Out:('k1', 1)

('k2', 2)

**Viewing keys, values and items**

By themselves the keys(), values() and items() methods return a dictionary *view object*. This is not a separate list of items. Instead, the view is always tied to the original dictionary.

key\_view **=** d**.**keys()

key\_view

Out:dict\_keys(['k1', 'k2'])

d['k3'] **=** 3

Print (d)

Out:{'k1': 1, 'k2': 2, 'k3': 3}

Print (key\_view)

Out:dict\_keys(['k1', 'k2', 'k3'])

**Advanced Lists**

list1 **=** [1,2,3]

**append**

You will definitely have used this method by now, which merely appends an element to the end of a list:

list1**.**append(4)

list1

Out:[1, 2, 3, 4]

**count**

We discussed this during the methods lectures, but here it is again. count() takes in an element and returns the number of times it occurs in your list:

list1**.**count(10)

Out:0

list1**.**count(2)

Out:1

**extend**

Many times people find the difference between extend and append to be unclear. So note:

**append: appends whole object at end:**

x **=** [1, 2, 3]

x**.**append([4, 5])

print(x)

Out:[1, 2, 3, [4, 5]]

**extend: extends list by appending elements from the iterable:**

x **=** [1, 2, 3]

x**.**extend([4, 5])

print(x)

Out:[1, 2, 3, 4, 5]

Note how extend() appends each element from the passed-in list. That is the key difference.

**index**

index() will return the index of whatever element is placed as an argument. Note: If the the element is not in the list an error is raised.

list1**.**index(2)

Out:1

list1**.**index(12)

**---------------------------------------------------------------------------**

**ValueError** Traceback (most recent call last)

**<ipython-input-8-56b94ada72bf>** in <module>**()**

**----> 1** list1**.**index**(12)**

**ValueError**: 12 is not in list

**insert**

insert() takes in two arguments: insert(index,object) This method places the object at the index supplied. For example:

Print (list1)

Out:[1, 2, 3, 4]

*# Place a letter at the index 2*

list1**.**insert(2,'inserted')

print(list1)

Out:[1, 2, 'inserted', 3, 4]

**pop**

You most likely have already seen pop(), which allows us to "pop" off the last element of a list. However, by passing an index position you can remove and return a specific element.

ele **=** list1**.**pop(1) *# pop the second element*

Print(list1)

Out:[1, 'inserted', 3, 4]

Print (ele)

Out:2

**remove**

The remove() method removes the first occurrence of a value. For example:

list1

Out:[1, 'inserted', 3, 4]

list1**.**remove('inserted')

Print (list1)

Out:[1, 3, 4]

list2 **=** [1,2,3,4,3]

list2**.**remove(3)

Print(list2)

Out:[1, 2, 4, 3]

**reverse**

As you might have guessed, reverse() reverses a list. Note this occurs in place! Meaning it affects your list permanently.

list2**.**reverse()

Print (list2)

Out:[3, 4, 2, 1]

**sort**

The sort() method will sort your list in place:

list2

Out:[3, 4, 2, 1]

list2**.**sort()

Print (list2)

Out:[1, 2, 3, 4]

The sort() method takes an optional argument for reverse sorting. Note this is different than simply reversing the order of items.

list2**.**sort(reverse**=True**)

list2

Out:[4, 3, 2, 1]

**Be Careful With Assignment!**

A common programming mistake is to assume you can assign a modified list to a new variable. While this typically works with immutable objects like strings and tuples:

x **=** 'hello world'

y **=** x**.**upper()

print(y)

Out:HELLO WORLD

This will NOT work the same way with lists:

x **=** [1,2,3]

y **=** x**.**append(4)

print(y)

Out:none

What happened? In this case, since list methods like append() affect the list *in-place*, the operation returns a None value. This is what was passed to **y**. In order to retain **x** you would have to assign a *copy* of **x** to **y**, and then modify **y**:

x **=** [1,2,3]

y **=** x**.**copy()

y**.**append(4)

print(x)

Out:[1, 2, 3]

print(y)

Out:[1, 2, 3, 4]

**Advanced Numbers**

Problem 1: Convert 1024 to binary and hexadecimal representation

Problem 2: Round 5.23222 to two decimal places

**Advanced Strings**

**Problem 3: Check if every letter in the string s is lower case**

s **=** 'hello how are you Mary, are you feeling okay?'

**Problem 4: How many times does the letter 'w' show up in the string below?**

s **=** 'twywywtwywbwhsjhwuwshshwuwwwjdjdid'

**Advanced Sets**

**Problem 5: Find the elements in set1 that are not in set2:**

set1 **=** {2,3,1,5,6,8}

set2 **=** {3,1,7,5,6,8}

**Problem 6: Find all elements that are in either set:**

**Advanced Dictionaries**

**Problem 7: Create this dictionary: {0: 0, 1: 1, 2: 8, 3: 27, 4: 64} using a dictionary comprehension.**

**Advanced Lists**

**Problem 8: Reverse the list below:**

list1 **=** [1,2,3,4]

**Problem 9: Sort the list below:**

list2 **=** [3,4,2,5,1]

**Solutions:**

**1.**

print(bin(1024))

print(hex(1024))

Out:

0b10000000000

0x400

**2**.

round(5.23222,2)

Out:

5.23

**3**.

s **=** 'hello how are you Mary, are you feeling okay?'

s**.**islower()

Out:False

**4**.

s **=** 'twywywtwywbwhsjhwuwshshwuwwwjdjdid'

s**.**count('w')

Out:12

**5**.

set1 **=** {2,3,1,5,6,8}

set2 **=** {3,1,7,5,6,8}

set1**.**difference(set2)

Out:{2}

**6**.

set1**.**union(set2)

Out:

{1, 2, 3, 5, 6, 7, 8}

**7**.

{x:x**\*\***3 **for** x **in** range(5)}

Out:{0: 0, 1: 1, 2: 8, 3: 27, 4: 64}

**8**.

list1 **=** [1,2,3,4]

list1**.**reverse()

list1

Out:[4, 3, 2, 1]

**9**.

list2 **=** [3,4,2,5,1]

list2**.**sort()

list2

Out:[1, 2, 3, 4, 5]

**With Statement Context Managers**

When you open a file using f = open('test.txt'), the file stays open until you specifically call f.close(). Should an exception be raised while working with the file, it remains open. This can lead to vulnerabilities in your code, and inefficient use of resources.

A context manager handles the opening and closing of resources, and provides a built-in try/finally block should any exceptions occur.

**Standard open() procedure, with a raised exception:**

p **=** open('oops.txt','a')

p**.**readlines()

p**.**close()

Out:

Error occurred

**---------------------------------------------------------------------------**

**UnsupportedOperation** Traceback (most recent call last)

**<ipython-input-1-ad7a2000735b>** in <module>**()**

1 p **=** open**('oops.txt','a')**

**----> 2** p**.**readlines**()**

3 p**.**close**()**

**UnsupportedOperation**: not readable

Let's see if we can modify our file:

p**.**write('add more text')

Out:13

Unfortunately, the exception prevented the last line, p.close() from running. Let's close the file manually:

p**.**close()

**Protect the file with try/except/finally**

A common workaround is to insert a try/except/finally clause to close the file whenever an exception is raised:

p **=** open('oops.txt','a')

**try**:

p**.**readlines()

**except**:

print('An exception was raised!')

**finally**:

p**.**close()

Out:An exception was raised!

Let's see if we can modify our file this time:

p**.**write('add more text')

**---------------------------------------------------------------------------**

**ValueError** Traceback (most recent call last)

**<ipython-input-5-1209a18e617d>** in <module>**()**

**----> 1** p**.**write**('add more text')**

**ValueError**: I/O operation on closed file.

**Save steps with with**

Now we'll employ our context manager. The syntax follows with [resource] as [target]: do something

**with** open('oops.txt','a') **as** p:

p**.**readlines()

**---------------------------------------------------------------------------**

**UnsupportedOperation** Traceback (most recent call last)

**<ipython-input-6-7ccc44e332f9>** in <module>**()**

1 **with** open**('oops.txt','a')** **as** p**:**

**----> 2** p**.**readlines**()**

**UnsupportedOperation**: not readable

Can we modify the file?

p**.**write('add more text')

**---------------------------------------------------------------------------**

**ValueError** Traceback (most recent call last)

**<ipython-input-7-1209a18e617d>** in <module>**()**

**----> 1** p**.**write**('add more text')**

**ValueError**: I/O operation on closed file.

With just one line of code we've handled opening the file, enclosing our code in a try/finally block, and closing our file all at the same time.

**Projects**

**Find PI to the Nth Digit** - Enter a number and have the program generate PI up to that many decimal places. Keep a limit to how far the program will go.

**Find e to the Nth Digit** - Just like the previous problem, but with e instead of PI. Enter a number and have the program generate e up to that many decimal places. Keep a limit to how far the program will go.

**Fibonacci Sequence** - Enter a number and have the program generate the Fibonacci sequence to that number or to the Nth number.

Numbers

**Find PI to the Nth Digit** - Enter a number and have the program generate PI up to that many decimal places. Keep a limit to how far the program will go.

**Find e to the Nth Digit** - Just like the previous problem, but with e instead of PI. Enter a number and have the program generate e up to that many decimal places. Keep a limit to how far the program will go.

**Fibonacci Sequence** - Enter a number and have the program generate the Fibonacci sequence to that number or to the Nth number.

**Prime Factorization** - Have the user enter a number and find all Prime Factors (if there are any) and display them.

**Next Prime Number** - Have the program find prime numbers until the user chooses to stop asking for the next one.

**Find Cost of Tile to Cover W x H Floor** - Calculate the total cost of tile it would take to cover a floor plan of width and height, using a cost entered by the user.

**Mortgage Calculator** - Calculate the monthly payments of a fixed term mortgage over given Nth terms at a given interest rate. Also figure out how long it will take the user to pay back the loan. For added complexity, add an option for users to select the compounding interval (Monthly, Weekly, Daily, Continually).

**Binary to Decimal and Back Converter** - Develop a converter to convert a decimal number to binary or a binary number to its decimal equivalent.

**Calculator** - A simple calculator to do basic operators. Make it a scientific calculator for added complexity.

**Unit Converter (temp, currency, volume, mass and more)** - Converts various units between one another. The user enters the type of unit being entered, the type of unit they want to convert to and then the value. The program will then make the conversion.

**Distance Between Two Cities** - Calculates the distance between two cities and allows the user to specify a unit of distance. This program may require finding coordinates for the cities like latitude and longitude.

**Tax Calculator** - Asks the user to enter a cost and either a country or state tax. It then returns the tax plus the total cost with tax.

**Factorial Finder** - The Factorial of a positive integer, n, is defined as the product of the sequence n, n-1, n-2, ...1 and the factorial of zero, 0, is defined as being 1. Solve this using both loops and recursion.

**Complex Number Algebra** - Show addition, multiplication, negation, and inversion of complex numbers in separate functions. (Subtraction and division operations can be made with pairs of these operations.) Print the results for each operation tested.

**Single card class**

Creating a Card Class with outside variables

Here we will use some outside variables that we know don't change regardless of the situation, such as a deck of cards. Regardless of what round,match, or game we're playing, we'll still need the same deck of cards.

*# We'll use this later*

**import** random

suits **=** ('Hearts', 'Diamonds', 'Spades', 'Clubs')

ranks **=** ('Two', 'Three', 'Four', 'Five', 'Six', 'Seven', 'Eight', 'Nine', 'Ten', 'Jack', 'Queen', 'King', 'Ace')

values **=** {'Two':2, 'Three':3, 'Four':4, 'Five':5, 'Six':6, 'Seven':7, 'Eight':8,

'Nine':9, 'Ten':10, 'Jack':11, 'Queen':12, 'King':13, 'Ace':14}

**class** Card:

**def** \_\_init\_\_(self,suit,rank):

self**.**suit **=** suit

self**.**rank **=** rank

self**.**value **=** values[rank]

**def** \_\_str\_\_(self):

**return** self**.**rank **+** ' of ' **+** self**.**suit

Create an example card

suits[0]

Out:'Hearts'

ranks[0]

Out:'Two'

two\_hearts **=** Card(suits[0],ranks[0])

two\_hearts

Out:<\_\_main\_\_.Card at 0x1dfaff6b898>

print(two\_hearts)

Out:Two of Hearts

two\_hearts**.**rank

Out: Two

two\_hearts**.**value

Out:2

values[two\_hearts**.**rank]

Out:2

**Deck Class**

**Using a class within another class**

We just created a single card, but how can we create an entire Deck of cards? Let's explore doing this with a class that utilizes the Card class.

A Deck will be made up of multiple Cards. Which mean's we will actually use the Card class within the \_\_init\_\_ of the Deck class.

**class** Deck:

**def** \_\_init\_\_(self):

*# Note this only happens once upon creation of a new Deck*

self**.**all\_cards **=** []

**for** suit **in** suits:

**for** rank **in** ranks:

*# This assumes the Card class has already been defined!*

self**.**all\_cards**.**append(Card(suit,rank))

**def** shuffle(self):

*# Note this doesn't return anything*

random**.**shuffle(self**.**all\_cards)

**def** deal\_one(self):

*# Note we remove one card from the list of all\_cards*

**return** self**.**all\_cards**.**pop()

**Create a Deck**

mydeck **=** Deck()

len(mydeck**.**all\_cards)

Out: 52

mydeck**.**all\_cards[0]

Out:<\_\_main\_\_.Card at 0x1dfaff269e8>

print(mydeck**.**all\_cards[0])

Out:Two of Hearts

mydeck**.**shuffle()

print(mydeck**.**all\_cards[0])

Out:Five of Spades

my\_card **=** mydeck**.**deal\_one()

print(my\_card)

Out:King of Clubs

**Player Class**

Let's create a Player Class, a player should be able to hold instances of Cards, they should also be able to remove and add them from their hand. We want the Player class to be flexible enough to add one card, or many cards so we'll use a simple if check to keep it all in the same method.

We'll keep this all in mind as we create the methods for the Player class.

**Player Class**

**class** Player:

**def** \_\_init\_\_(self,name):

self**.**name **=** name

*# A new player has no cards*

self**.**all\_cards **=** []

**def** remove\_one(self):

*# Note we remove one card from the list of all\_cards*

*# We state 0 to remove from the "top" of the deck*

*# We'll imagine index -1 as the bottom of the deck*

**return** self**.**all\_cards**.**pop(0)

**def** add\_cards(self,new\_cards):

**if** type(new\_cards) **==** type([]):

self**.**all\_cards**.**extend(new\_cards)

**else**:

self**.**all\_cards**.**append(new\_cards)

**def** \_\_str\_\_(self):

**return** f'Player {self**.**name} has {len(self**.**all\_cards)} cards.'

jose **=** Player("Jose")

jose

Out:<\_\_main\_\_.Player at 0x1dfaff8b940>

print(jose)

Out:Player Jose has 0 cards.

print(two\_hearts)

Out:<\_\_main\_\_.Card at 0x1dfaff6b898>

jose**.**add\_cards(two\_hearts)

print(jose)

Out:Player Jose has 1 cards.

jose**.**add\_cards([two\_hearts,two\_hearts,two\_hearts])

print(jose)

Out:Player Jose has 4 cards.

**War Game Logic**

player\_one **=** Player("One")

player\_two **=** Player("Two")

**Setup New Game**

new\_deck **=** Deck()

new\_deck**.**shuffle()

**Split the Deck between players**

len(new\_deck**.**all\_cards)**/**2

Out: 26.0

**for** x **in** range(26):

player\_one**.**add\_cards(new\_deck**.**deal\_one())

player\_two**.**add\_cards(new\_deck**.**deal\_one())

len(new\_deck**.**all\_cards)

Out:0

len(player\_one**.**all\_cards)

Out:26

len(player\_two**.**all\_cards)

Out:26

**Play the Game**

**import** pdb

game\_on **=** **True**

round\_num **=** 0

**while** game\_on:

round\_num **+=** 1

print(f"Round {round\_num}")

*# Check to see if a player is out of cards:*

**if** len(player\_one**.**all\_cards) **==** 0:

print("Player One out of cards! Game Over")

print("Player Two Wins!")

game\_on **=** **False**

**break**

**if** len(player\_two**.**all\_cards) **==** 0:

print("Player Two out of cards! Game Over")

print("Player One Wins!")

game\_on **=** **False**

**break**

*# Otherwise, the game is still on!*

*# Start a new round and reset current cards "on the table"*

player\_one\_cards **=** []

player\_one\_cards**.**append(player\_one**.**remove\_one())

player\_two\_cards **=** []

player\_two\_cards**.**append(player\_two**.**remove\_one())

at\_war **=** **True**

**while** at\_war:

**if** player\_one\_cards[**-**1]**.**value **>** player\_two\_cards[**-**1]**.**value:

*# Player One gets the cards*

player\_one**.**add\_cards(player\_one\_cards)

player\_one**.**add\_cards(player\_two\_cards)

*# No Longer at "war" , time for next round*

at\_war **=** **False**

*# Player Two Has higher Card*

**elif** player\_one\_cards[**-**1]**.**value **<** player\_two\_cards[**-**1]**.**value:

*# Player Two gets the cards*

player\_two**.**add\_cards(player\_one\_cards)

player\_two**.**add\_cards(player\_two\_cards)

*# No Longer at "war" , time for next round*

at\_war **=** **False**

**else**:

print('WAR!')

*# This occurs when the cards are equal.*

*# We'll grab another card each and continue the current war.*

*# First check to see if player has enough cards*

*# Check to see if a player is out of cards:*

**if** len(player\_one**.**all\_cards) **<** 5:

print("Player One unable to play war! Game Over at War")

print("Player Two Wins! Player One Loses!")

game\_on **=** **False**

**break**

**elif** len(player\_two**.**all\_cards) **<** 5:

print("Player Two unable to play war! Game Over at War")

print("Player One Wins! Player One Loses!")

game\_on **=** **False**

**break**

*# Otherwise, we're still at war, so we'll add the next cards*

**else**:

**for** num **in** range(5):

player\_one\_cards**.**append(player\_one**.**remove\_one())

player\_two\_cards**.**append(player\_two**.**remove\_one())

Out:

Round 1

Round 2

Round 3

Round 4

Round 5

Round 6

Round 7

Round 8

Round 9

Round 10

Round 11

Round 12

Round 13

Round 14

Round 15

Round 16

Round 17

Round 18

Round 19

Round 20

Round 21

Round 22

Round 23

Round 24

Round 25

Round 26

Round 27

Player One out of cards! Game Over

**Game Setup in One Cell**

player\_one **=** Player("One")

player\_two **=** Player("Two")

new\_deck **=** Deck()

new\_deck**.**shuffle()

**for** x **in** range(26):

player\_one**.**add\_cards(new\_deck**.**deal\_one())

player\_two**.**add\_cards(new\_deck**.**deal\_one())

game\_on **=** **True**

round\_num **=** 0

**while** game\_on:

round\_num **+=** 1

print(f"Round {round\_num}")

*# Check to see if a player is out of cards:*

**if** len(player\_one**.**all\_cards) **==** 0:

print("Player One out of cards! Game Over")

print("Player Two Wins!")

game\_on **=** **False**

**break**

**if** len(player\_two**.**all\_cards) **==** 0:

print("Player Two out of cards! Game Over")

print("Player One Wins!")

game\_on **=** **False**

**break**

*# Otherwise, the game is still on!*

*# Start a new round and reset current cards "on the table"*

player\_one\_cards **=** []

player\_one\_cards**.**append(player\_one**.**remove\_one())

player\_two\_cards **=** []

player\_two\_cards**.**append(player\_two**.**remove\_one())

at\_war **=** **True**

**while** at\_war:

**if** player\_one\_cards[**-**1]**.**value **>** player\_two\_cards[**-**1]**.**value:

*# Player One gets the cards*

player\_one**.**add\_cards(player\_one\_cards)

player\_one**.**add\_cards(player\_two\_cards)

*# No Longer at "war" , time for next round*

at\_war **=** **False**

*# Player Two Has higher Card*

**elif** player\_one\_cards[**-**1]**.**value **<** player\_two\_cards[**-**1]**.**value:

*# Player Two gets the cards*

player\_two**.**add\_cards(player\_one\_cards)

player\_two**.**add\_cards(player\_two\_cards)

*# No Longer at "war" , time for next round*

at\_war **=** **False**

**else**:

print('WAR!')

*# This occurs when the cards are equal.*

*# We'll grab another card each and continue the current war.*

*# First check to see if player has enough cards*

*# Check to see if a player is out of cards:*

**if** len(player\_one**.**all\_cards) **<** 5:

print("Player One unable to play war! Game Over at War")

print("Player Two Wins! Player One Loses!")

game\_on **=** **False**

**break**

**elif** len(player\_two**.**all\_cards) **<** 5:

print("Player Two unable to play war! Game Over at War")

print("Player One Wins! Player One Loses!")

game\_on **=** **False**

**break**

*# Otherwise, we're still at war, so we'll add the next cards*

**else**:

**for** num **in** range(5):

player\_one\_cards**.**append(player\_one**.**remove\_one())

player\_two\_cards**.**append(player\_two**.**remove\_one())

Output:

len(player\_one**.**all\_cards)

Out:27

len(player\_two**.**all\_cards)

Out:25

print(player\_one\_cards[**-**1])

Out:Ace of Diamonds

print(player\_two\_cards[**-**1])

Out:Four of Hearts

**Milestone Project 2 - Blackjack Game**

In this milestone project you will be creating a Complete BlackJack Card Game in Python.

Here are the requirements:

• You need to create a simple text-based BlackJack game

• The game needs to have one player versus an automated dealer.

• The player can stand or hit.

• The player must be able to pick their betting amount.

• You need to keep track of the player's total money.

• You need to alert the player of wins, losses, or busts, etc...

And most importantly:

**• You must use OOP and classes in some portion of your game. You can not just use functions in your game. Use classes to help you define the Deck and the Player's hand. There are many right ways to do this, so explore it well!**

**Milestone Project 2 - Walkthrough Steps Workbook**

Below is a set of steps for you to follow to try to create the Blackjack Milestone Project game!

**Game Play**

To play a hand of Blackjack the following steps must be followed:

1 Create a deck of 52 cards

2 Shuffle the deck

3 Ask the Player for their bet

4 Make sure that the Player's bet does not exceed their available chips

5 Deal two cards to the Dealer and two cards to the Player

6 Show only one of the Dealer's cards, the other remains hidden

7 Show both of the Player's cards

8 Ask the Player if they wish to Hit, and take another card

9 If the Player's hand doesn't Bust (go over 21), ask if they'd like to Hit again.

10 If a Player Stands, play the Dealer's hand. The dealer will always Hit until the Dealer's value meets or exceeds 17

11 Determine the winner and adjust the Player's chips accordingly

12 Ask the Player if they'd like to play again

**Playing Cards**

A standard deck of playing cards has four suits (Hearts, Diamonds, Spades and Clubs) and thirteen ranks (2 through 10, then the face cards Jack, Queen, King and Ace) for a total of 52 cards per deck. Jacks, Queens and Kings all have a rank of 10. Aces have a rank of either 11 or 1 as needed to reach 21 without busting. As a starting point in your program, you may want to assign variables to store a list of suits, ranks, and then use a dictionary to map ranks to values.

**The Game**

\*\* Step 1: Import the random module. This will be used to shuffle the deck prior to dealing. Then, declare variables to store suits, ranks and values. You can develop your own system, or copy ours below. Finally, declare a Boolean value to be used to control while loops. This is a common practice used to control the flow of the game.\*\*

suits = ('Hearts', 'Diamonds', 'Spades', 'Clubs')

ranks = ('Two', 'Three', 'Four', 'Five', 'Six', 'Seven', 'Eight', 'Nine', 'Ten', 'Jack', 'Queen', 'King', 'Ace')

values = {'Two':2, 'Three':3, 'Four':4, 'Five':5, 'Six':6, 'Seven':7, 'Eight':8, 'Nine':9, 'Ten':10, 'Jack':10,

'Queen':10, 'King':10, 'Ace':11}

**import** random

suits **=** **pass**

ranks **=** **pass**

values **=** **pass**

playing **=** **True**

**Class Definitions**

Consider making a Card class where each Card object has a suit and a rank, then a Deck class to hold all 52 Card objects, and can be shuffled, and finally a Hand class that holds those Cards that have been dealt to each player from the Deck.

**Step 2: Create a Card Class**

A Card object really only needs two attributes: suit and rank. You might add an attribute for "value" - we chose to handle value later when developing our Hand class.  
In addition to the Card's \_\_init\_\_ method, consider adding a \_\_str\_\_ method that, when asked to print a Card, returns a string in the form "Two of Hearts"

**class** Card:

**def** \_\_init\_\_(self):

**pass**

**def** \_\_str\_\_(self):

**pass**

**Step 3: Create a Deck Class**

Here we might store 52 card objects in a list that can later be shuffled. First, though, we need to *instantiate* all 52 unique card objects and add them to our list. So long as the Card class definition appears in our code, we can build Card objects inside our Deck \_\_init\_\_ method. Consider iterating over sequences of suits and ranks to build out each card. This might appear inside a Deck class \_\_init\_\_ method:

for suit in suits:

for rank in ranks:

In addition to an \_\_init\_\_ method we'll want to add methods to shuffle our deck, and to deal out cards during gameplay.

OPTIONAL: We may never need to print the contents of the deck during gameplay, but having the ability to see the cards inside it may help troubleshoot any problems that occur during development. With this in mind, consider adding a \_\_str\_\_ method to the class definition.

**class** Deck:

**def** \_\_init\_\_(self):

self**.**deck **=** [] *# start with an empty list*

**for** suit **in** suits:

**for** rank **in** ranks:

**pass**

**def** \_\_str\_\_(self):

**pass**

**def** shuffle(self):

random**.**shuffle(self**.**deck)

**def** deal(self):

**pass**

TESTING: Just to see that everything works so far, let's see what our Deck looks like!

test\_deck **=** Deck()

print(test\_deck)

**Step 4: Create a Hand Class**

In addition to holding Card objects dealt from the Deck, the Hand class may be used to calculate the value of those cards using the values dictionary defined above. It may also need to adjust for the value of Aces when appropriate.

**class** Hand:

**def** \_\_init\_\_(self):

self**.**cards **=** [] *# start with an empty list as we did in the Deck class*

self**.**value **=** 0 *# start with zero value*

self**.**aces **=** 0 *# add an attribute to keep track of aces*

**def** add\_card(self,card):

**pass**

**def** adjust\_for\_ace(self):

**pass**

**Step 5: Create a Chips Class**  
In addition to decks of cards and hands, we need to keep track of a Player's starting chips, bets, and ongoing winnings. This could be done using global variables, but in the spirit of object oriented programming, let's make a Chips class instead!

**class** Chips:

**def** \_\_init\_\_(self):

self**.**total **=** 100 *# This can be set to a default value or supplied by a user input*

self**.**bet **=** 0

**def** win\_bet(self):

**pass**

**def** lose\_bet(self):

**pass**

**Function Defintions**

A lot of steps are going to be repetitive. That's where functions come in! The following steps are guidelines - add or remove functions as needed in your own program.

**Step 6: Write a function for taking bets**

Since we're asking the user for an integer value, this would be a good place to use try/except. Remember to check that a Player's bet can be covered by their available chips.

**def** take\_bet():

**pass**

**Step 7: Write a function for taking hits**

Either player can take hits until they bust. This function will be called during gameplay anytime a Player requests a hit, or a Dealer's hand is less than 17. It should take in Deck and Hand objects as arguments, and deal one card off the deck and add it to the Hand. You may want it to check for aces in the event that a player's hand exceeds 21.

**def** hit(deck,hand):

**pass**

**Step 8: Write a function prompting the Player to Hit or Stand**

This function should accept the deck and the player's hand as arguments, and assign playing as a global variable.  
If the Player Hits, employ the hit() function above. If the Player Stands, set the playing variable to False - this will control the behavior of a while loop later on in our code.

**def** hit\_or\_stand(deck,hand):

**global** playing *# to control an upcoming while loop*

**pass**

**Step 9: Write functions to display cards**

When the game starts, and after each time Player takes a card, the dealer's first card is hidden and all of Player's cards are visible. At the end of the hand all cards are shown, and you may want to show each hand's total value. Write a function for each of these scenarios.

**def** show\_some(player,dealer):

**pass**

**def** show\_all(player,dealer):

**pass**

**Step 10: Write functions to handle end of game scenarios**

Remember to pass player's hand, dealer's hand and chips as needed.

**def** player\_busts():

**pass**

**def** player \_wins():

**pass**

**def** dealer\_busts():

**pass**

**def** dealer\_wins():

**pass**

**def** push():

**pass**

And now on to the game!!

**while** **True**:

*# Print an opening statement*

*# Create & shuffle the deck, deal two cards to each player*

*# Set up the Player's chips*

*# Prompt the Player for their bet*

*# Show cards (but keep one dealer card hidden)*

**while** playing: *# recall this variable from our hit\_or\_stand function*

*# Prompt for Player to Hit or Stand*

*# Show cards (but keep one dealer card hidden)*

*# If player's hand exceeds 21, run player\_busts() and break out of loop*

**break**

*# If Player hasn't busted, play Dealer's hand until Dealer reaches 17*

*# Show all cards*

*# Run different winning scenarios*

*# Inform Player of their chips total*

*# Ask to play again*

**break**

**Git and GitHub learning resources**

There are a lot of helpful Git and GitHub resources available.

**GitHub flow**

Follow GitHub flow to collaborate on projects.

**Introduction**

GitHub flow is a lightweight, branch-based workflow. The GitHub flow is useful for everyone, not just developers.

**Note:**

You can complete all steps of GitHub flow through the GitHub web interface, command line and GitHub CLI, or GitHub Desktop.

**Create a branch**

Create a branch in your repository. A short, descriptive branch name enables your collaborators to see ongoing work at a glance. For example, increase-test-timeout or add-code-of-conduct.

**Make changes**

On your branch, make any desired changes to the repository. For more information, see "Creating new files", "Editing files", "Renaming a file", "Moving a file to a new location", or "Deleting files in a repository."

Your branch is a safe place to make changes. If you make a mistake, you can revert your changes or push additional changes to fix the mistake. Your changes will not end up on the default branch until you merge your branch.

Commit and push your changes to your branch. Give each commit a descriptive message to help you and future contributors understand what changes the commit contains. For example, fix typo or increase rate limit.

Ideally, each commit contains an isolated, complete change. This makes it easy to revert your changes if you decide to take a different approach. For example, if you want to rename a variable and add some tests, put the variable rename in one commit and the tests in another commit. Later, if you want to keep the tests but revert the variable rename, you can revert the specific commit that contained the variable rename. If you put the variable rename and tests in the same commit or spread the variable rename across multiple commits, you would spend more effort reverting your changes.

By committing and pushing your changes, you back up your work to remote storage. This means that you can access your work from any device. It also means that your collaborators can see your work, answer questions, and make suggestions or contributions.

Make a separate branch for each set of unrelated changes. This makes it easier for reviewers to give feedback. It also makes it easier for you and future collaborators to understand the changes and to revert or build on them. Additionally, if there is a delay in one set of changes, your other changes aren't also delayed.

**Create a pull request**