Advanced Python with Joe Marini (LinkedIn Learning)

PEP8 - Style Guide for Python Code:

https://www.python.org/dev/peps/pep-0008/

Code and Structure Formatting: (See ***codingstyle_finished.py*** for examples of rules)

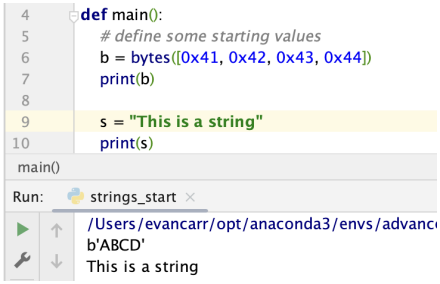
- ➔ Import statements always go at the top of the file and each have their own line
- ➔ Indent code using spaces instead of tabs
- ➔ Use four spaces for each indentation level
- ➔ Limit lines of code to 79 characters including indentation (72 characters for comments and docstrings) ⇨ makes it easy to have multiple files open side by side.
- ➔ Functions and classes should be separated by two blank lines and one blank line between methods within class definitions.
- ➔ No white spaces around function calls, indexes, or keyword arguments

  WHITE SPACE CONVENTIONS:
  - ➔ No white spaces immediately within brackets, braces, or parentheses
  - ➔ No white spaces between function call and list of arguments
  - ➔ No spaces between data structure names and their accessors
  - ➔ Do put spaces around math operators

Python Truth Values:

- ➔ Any object is considered to be equivalent to Boolean true unless its class defines a bool method that returns False or has a len() method that returns a zero length
- ➔ **Built-In Objects that return False**:
  - False and None evaluate to false
  - Numeric values: 0, 0.0, 0j
  - Decimal(0) (decimal object when given a value of zero), Fraction(0,x) (fractional object with zero in numerator.
  - Empty sequences and collections: “”, (), {}, []
  - Empty sets and ranges: set(), range(0)
- ➔ Custom objects automatically return true unless they override the bool function and return a False value or override the len() function and return 0
- ➔ **Boolean Operations**:
  - **and** ⇨ if the first operand evaluates to false, the second operand is not evaluated, since the and will automatically evaluate to False
  - **or** ⇨ only evaluates the second operand if the first evaluates to False, because anything or evaluated with True will come out to be True

Strings Vs. Bytes:



- ➔ A string is a sequence of unicode characters while bytes are a sequence of raw 8-bit values. You cannot just treat a string as if it were a sequence of 8-bit values
- ➔ b and s cannot be added together as they are and will throw a type error, but they can be if we use the built-in decode function

```
# print(b + s)  <= throws a type error
s2 = b.decode('utf-8')
print(s + s2)
```

```
strings_start
/Users/evancarr/opt/anaconda3/envs/ad
b'ABCD'
This is a string
This is a stringABCD
```

➔ Likewise, we can use the encode() function which takes a string and returns bytes

➔ But when encoded to utf-32, it returns a bunch of hex numbers when printed

➔ Remember:

encode() ⇨ characters to bytes

decode() ⇨ bytes to characters

```
b2 = s.encode('utf-8')
print(b + b2)
```

```
strings_start
/Users/evancarr/opt/anaconda3/envs/ad
b'ABCD'
This is a string
This is a stringABCD
b'ABCDThis is a string'
```

Template Strings (String formatting):

- ➔ Requires an import of the template class from the string module: ***from string import Template***
- ➔ Also requires special syntax of ***${ }***
- ➔ Values are given as keyword arguments and the ***substitute()*** method
- ➔ You can use a dictionary to hold the values you want to substitute into particular fields
- ➔ If all you need is simple variable substitution, the template string method is easier to use and the code is more readable. It is just about straight forward value substitution.
- ➔ Format method offers a lot of specific format options, however.
- ➔ If templates are supplied from a source you do not control or fully trust, the template method offers more security.

Utilities and Built In Functions:

- ➔ Use these functions wherever possible to write more smoothe code. They are effective at this because they are native code.
- ➔ ***.any()*** will return True if any of the sequence values are True
- ➔ ***.all()*** will return True if ALL of the sequence values are True
- ➔ ***.min()*** and ***.max()*** will return minimum and maximum values in a sequence
- ➔ ***.sum()*** will return the sum of all values in a sequence

Iterators:

- ➔ ***.iter()*** creates an iterator object out of the sequence passed to it
- ➔ ***.next()*** returns the next item in the iterator

- ➔ ***.iter()*** is useful when you give it a function to use to generate the sequence items
- ➔ In the code here, the ***.iter()*** function is being called to iterate through the lines of the text file that has been opened. For each line, it will call the function we have passed to it, in this case ***.readline(),*** to generate each value, and the ***sentinel argument (" ")***, which is the second one passed, is what tells the ***.iter()*** function to stop, as it iterates through values and returns them. So here, the iterator is looking for the empty string, which indicates the end of the file.

```
with open("testfile.txt", "r") as filepointer:
    for line in iter(filepointer.readline, ''):
        # " " is the sentinel, which means that the loop will stop when
        # it encounters a blank line
        print(line)
```

- ➔ ***.enumerate()*** returns a tuple containing a count (from start which defaults to 0) and the values obtained from iterating over a sequence, rather than needing cumbersome code

```
for i, m in enumerate(zip(days, daysFr)):
    print(i, m[0], "=", m[1], "in French")
```

```
0 Sun = Dim in French
1 Mon = Lun in French
2 Tue = Mar in French
3 Wed = Mer in French
4 Thu = Jeu in French
5 Fri = Ven in French
6 Sat = Sam in French
```

- ➔ ***.zip()*** returns an iterator of tuples, where the i-th tuple contains the i-th element from each of the argument sequences or iterables
  - If ***.zip()*** is used on sequences of differing lengths, it will terminate when the shorter sequence is exhausted.

Transforms:

```
nums = (1, 8, 4, 5, 13, 26, 381, 410, 58, 47)
chars = "abcDeFGHiJklmnoP"
grades = (81, 89, 94, 78, 61, 66, 99, 74)

# TODO: use filter to remove items from a list
odds = list(filter(filterFunc, nums))
print(odds)

# TODO: use filter on non-numeric sequence
lowers = list(filter(filterFunc2, chars))
print(lowers)

# TODO: use map to create a new sequence of values
squares = list(map(squareFunc, nums))
print(squares)

# TODO: use sorted and map to change numbers to grades
grades = sorted(grades)
letters = list(map(toGrade, grades))
num_and_letter = list(zip(grades, letters))
print(num_and_letter)
```

```
[1, 5, 13, 381, 47]
['a', 'b', 'c', 'e', 'i', 'k', 'l', 'm', 'n', 'o']
[1, 64, 16, 25, 169, 676, 145161, 168100, 3364, 2209]
[(61, 'F'), (66, 'D'), (74, 'C'), (78, 'C'), (81, 'B'), (89, 'B'), (94, 'A'), (99, 'A')]
```

```
def filterFunc(x):
    if x % 2 == 0:
        return False
    return True

def filterFunc2(x):
    if x.isupper():
        return False
    return True

def squareFunc(x):
    return x ** 2

def toGrade(x):
    if x >= 90:
        return 'A'
    elif 90 > x >= 80:
        return 'B'
    elif 80 > x >= 70:
        return 'C'
    elif 70 > x >= 65:
        return 'D'
    return 'F'
```

- ➔ ***.filter()*** is a function that takes two arguments: a function and a sequence. The function is called with all the items in the sequence and a new list is returned which contains items for which the function evaluates to True. It creates an iterator that filters out values from a given sequence. You pass it a function to perform a boolean (True or False) test. Any items that return False to the passed function are removed from the sequence.
- ➔ ***.map()*** is a function that takes two arguments, a function and a sequence. It calls the function with all the items in the sequence and returns a list of the return values. ***.map()*** creates an iterator that takes one or more sequences of values and creates a new sequence by applying a passed function to each value in the original sequences

Itertools:

- ➔ Infinite iterators will generate iterators for as long as you want them to and will not stop until told to terminate. (Remember to use list() to print out iterators, since they return objects.)
- ➔ ***.cycle()*** ⇨ cycles over a set of values and will start back at the beginning upon reaching end.

- ➔ ***.count()*** ⇨ creates a counter, takes a starting value (default = 0) and a step value (default = 1)
- ➔ ***.accumulate()*** ⇨ (Not an infinite iterator, so it can be given to a list variable, for example, and it will create a list.) This creates a list of accumulated totals of the numbers passed to it. The default operation is addition, but it will take others as well, such as ***max***, which will repeatedly give the highest value it has reached thus far.
- ➔ ***.chain()*** ⇨ takes multiple sequences and chains them together to act as one. (Also not an infinite iterator, so it can be used to produce a list.)
- ➔ ***.dropwhile()*** and ***.takewhile()*** ⇨ provide values until a trigger value is reached. They both take a function to perform the value test. ***.dropwhile()*** will drop values from the sequence while the test function returns True and will start returning values after that. Conversely, ***.takewhile()*** will return to you values while the test function returns true and then stop doing so once the target value is reached

```
def testFunction(x):
    return x < 40
```

```
print(list(itertools.dropwhile(testFunction, vals)))
print(list(itertools.takewhile(testFunction, vals)))
```

```
[40, 50, 40, 30]
[10, 20, 30]
```

Advanced Python Functions:

DocStrings:

- ➔ To get the documentation for a function, use: print(function_name.__doc__)
- ➔ The above also works for collections and modules, along with their classes
- ➔ This way, you can easily learn about various Python APIs as you work and as you need them rather than having to go to cumbersome documentation
- ➔ To create such documentation for your own functions, just include a docstring at the beginning of your function in triple quotes “”” DOC STRING ”””
- ➔ Start a docstring with the name of the function, the arguments it takes (as they will be passed to it, so basically the signature of the function), and a little description of the function.
- ➔ After describing the function, describe each parameter on their own lines beneath.
- ➔ Best practices:
  - Should always be enclosed in triple quotes, even if they’re short, so they can be expanded later if necessary.
  - First line should summarize the function, class, or module and what its main purpose is.
  - For modules, it should list important classes, sub-modules, functions, and exceptions
  - For classes, list important methods, along with any enums, static member functions, or properties that the class supports
  - For functions:
    - List and explain each of the parameters, one per line
    - If it returns a value, explain it is the docstring, if not, do not mention
    - If it raises an exception, list those as well
- ➔ PEP-257 for more

Variable Argument Lists (using * and **):

- ➔ These make it possible to build functions that have a high degree of flexibility and can take a number of parameters
- ➔ An example would be a function that adds up all the numbers passed to it. It would be silly to require a list of numbers, so instead, it can be defined: ***def addition(*numbers):***
- ➔ Put an asterisk character in front of the parameter that you want to make variable. This parameter MUST COME after all of the positional parameters that the function defines
  - Ex: ***def log_message(msg_type, msg, *params):***
  - These are conventionally called ****args***
  - You can also pass a list by putting * in front of the list name
- ➔ Potential drawback ⇨ if you decide later to change the function signature to add more positional parameters, all of the callers of your function must also change.
- ➔ Mostly useful when the number of parameters the function takes are very small, so that it does not cause issues later on.

```
def addition(*args):
    result = 0
    for arg in args:
        result += arg
    return result

def main():
    # pass different arguments
    print(addition(5, 10, 15, 20))
    print(addition(1, 2, 3))

    # pass an existing list
    myNums = [5, 10, 15, 20]
    print(addition(*myNums))
```

```
50
6
50
```

Lambda Functions:

- ➔ Small, anonymous functions that can be used on their own or passed to other functions where you need one. Often used where defining a whole other function would needlessly increase complexity and reduce readability.
- ➔ Defined as: ***lambda (parameters) : (expression)***
- ➔ They are also nice because you can see code right where it's being used vs a function name.

```
def CelsisusToFahrenheit(temp):
    return (temp * 9/5) + 32

def FahrenheitToCelsisus(temp):
    return (temp-32) * 5/9

def main():
    ctemps = [0, 12, 34, 100]
    ftemps = [32, 65, 100, 212]

    # Use regular functions to convert temps
    print(list(map(FahrenheitToCelsisus, ftemps)))
    print(list(map(CelsisusToFahrenheit, ctemps)))

    # Use lambdas to accomplish the same thing
    print(list(map(lambda t: (t-32) * 5/9, ftemps)))
    print(list(map(lambda t: (t * 9/5) + 32, ctemps)))
```

Keyword-Only Arguments:

- ➔ Ex: ***def my_function(arg1, arg2, arg3 = something):***
- ➔ Calling the function: ***my_function(1, 2, arg3 = "boots")***
- ➔ These can be important when there is a critical parameter that you want the caller to be very aware of when passing it, thus they must specify the parameter via keyword argument
- ➔ This is also helpful when others read the code and offer much clarity.
- ➔ In your functions, you can separate positional arguments from your keyword only arguments with an *. Ex: ***def my_critical_function(arg1, *, suppress_exceptions = False):***
- ➔ Main function is to improve the readability of code

Advanced Collections:

Basic Collections:

- ➔ Lists ⇨ mutable sequences of values, declared using [] and separated by commas
- ➔ Tuples ⇨ fixed, immutable sequence of values, defined with () and separated by commas
- ➔ Set ⇨ unordered, mutable collection of distinct values, defined using {} and separated by commas
- ➔ Dictionary ⇨ unordered, mutable collection of key: value pairs, defined with {}
- ➔ More collections are available by importing Python's collections module. These include:
  - namedtuple → tuple with named fields, giving names and meaning to each item's position in the tuple
  - OrderedDict and defaultdict → dictionaries with special properties
  - counter → can track the number of distinct values added to them
  - deque → a double ended list

namedtuple:

- ➔ Useful so that code does not get hard to read when working with other developers
- ➔ Useful, for example, when dealing with coordinate points
- ➔ Also have helpful functions for working with them
- ➔ The tuple itself gets a name, as well as the contents.
- ➔ Makes code much easier to read and maintain.
- ➔ Helper functions make it easy to work with these, such as ._replace() as seen here.
- ➔ Limitation: no default argument values, etc., for which you would need to use a class

```
Point = collections.namedtuple("Point", "x y")

p1 = Point(10, 20)
p2 = Point(30, 40)

print(p1, p2)
print(p1.x, p1.y)

# use _replace to create a new instance
p1 = p1._replace(x=100)
print(p1)
```

prints out ↓

```
Point(x=10, y=20) Point(x=30, y=40)
10 20
Point(x=100, y=20)
```

defaultdict:

- ➔ These come in handy in situations where a normal dict would require extra code, for example, when you want to create a dictionary that keeps count of how many of each item you have in a list. You would have to write an entire for loop with if-else conditions to check if the item is already in the dict, then to increment the count kept as the value, and then if not in the dict, add to the dict, etc.
- ➔ defaultdict on the other hand contains a default value for items, thus eliminating some of the code.
- ➔ Upon initializing, it requires you to give it a factory function, in this case, the int class, which acts as the creator of the default value
  - Ex: ***fruit_counter = defaultdict(int)*** ⇦ this says if I try to access a key that does not exist, create a default value using the object passed as the constructor (this will initialize the value to zero by default)
  - If you want the value to start off at a different number, you could use a lambda function
    - Ex: ***fruit_counter = defaultdict(lambda: 100)*** ⇦ this would initialize each key added to the defaultdict with a value of 100
    - The default value can be any existing or custom object you want
    - MUST REMEMBER: any key that is added will give the default value when accessed

- In a situation where a key missing from a dictionary is an important indicator, defaultdict is not the right collection type to use

Counter:

- ➔ A dictionary subclass for counting hashable objects
- ➔ Have features in addition to the ones in defaultdict
- ➔ See code for explanation
- ➔ ***.most_common()*** will return the item that is most common in the counter dict, takes an argument that will give you the top however many counts for the dict. By default, it will return all and their counts in descending order, returned as a list of tuples where the first item is the item and the second is the count of that item from the counter dictionary object.
- ➔ ***.update()*** will add the passed sequence to the counter that calls it
- ➔ ***.subtract()*** will remove the passed sequence from the counter that passed it.
- ➔ ***Logical &*** will return what items are present in both counters, for example, print(counter1, counter2) will print whatever items are common to both and their counts

```
# list of students in class 1
class1 = ["Bob", "James", "Chad", "Darcy", "Penny", "Hannah"
        "Kevin", "James", "Melanie", "Becky", "Steve", "Frank"]

# list of students in class 2
class2 = ["Bill", "Barry", "Cindy", "Debbie", "Frank",
        "Gabby", "Kelly", "James", "Joe", "Sam", "Tara", "Ziggy"]

# Create a Counter for class1 and class2
c1 = Counter(class1)
c2 = Counter(class2)

# How many students in class 1 named James?
print(c1["James"])

# How many students are in class 1?
print(sum(c1.values()), "students in class 1")

# Combine the two classes
c1.update(class2)
print(sum(c1.values()), "students in class 1 and 2")

# What's the most common name in the two classes?
print(c1.most_common(3))

# Separate the classes again
c1.subtract(class2)
print(c1.most_common(1))

# What's common between the two classes?
print(c1 & c2)
```

⇨

```
2
11 students in class 1
23 students in class 1 and 2
[('James', 3), ('Frank', 2), ('Bob', 1)]
[('James', 2)]
Counter({'James': 1, 'Frank': 1})
```

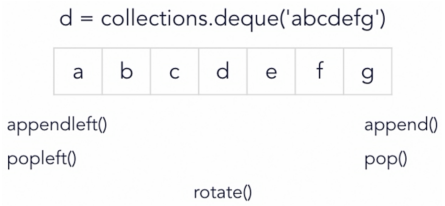
OrderedDict:

- ➔ Remembers the order in which items are inserted
- ➔ If you mess with the data though, you will have to resort. It will not keep it sorted.
- ➔ ***.popitem()*** will pop items from the beginning or end (depending on the argument passed, True for last item which is also default, and False for first item added.) of the OrderedDict as if it were a stack
- ➔ Equality testing depends on the order of the values. They have to be the same for two different OrderedDicts to be considered equal. But when comparing an OrderedDict

```
# list of sport teams with wins and losses
sportTeams = [("Royals", (18, 12)), ("Rockets", (24, 6)),
        ("Cardinals", (20, 10)), ("Dragons", (22, 8)),
        ("Kings", (15, 15)), ("Chargers", (20, 10)),
        ("Jets", (16, 14)), ("Warriors", (25, 5))]

# sort the teams by number of wins
sortedTeams = sorted(sportTeams, key=lambda t: t[1][0], reverse=True)

# create an ordered dictionary of the teams
teams = OrderedDict(sortedTeams)
print(teams)

# Use popitem to remove the top item
tm, wl = teams.popitem(False)
print("Top team: ", tm, wl)

# What are next the top 4 teams?
for i, team in enumerate(teams, start=1):
    print(i, team)
    if i == 4:
        break

# test for equality
a = OrderedDict({"a": 1, "b": 2, "c": 3})
b = OrderedDict({"a": 1, "c": 3, "b": 2})
print("Equality test: ", a == b)
```

to a regular dictionary, the order does not matter.

Deque:



- ➔ Hybrid, iterable object between a stack and a queue, a double-ended queue
- ➔ Memory-efficient, and can be used to add or pop data from either side
  - ◆ ***.appendleft(), .append(), .popleft(), .pop()***
- ➔ Can be initialized empty or getting their initial data from an iterable object.
  - ◆ Ex: ***d = collections.deque('abcdefg')***
- ➔ Can also be specified to have a specific length
- ➔ ***.rotate()*** ⇨ can operate in either direction, takes a parameter (default = 1) for how many items worth to rotate. Positive numbers rotate right, and negatives go left.
- ➔ Useful when you need to be able to operate on both ends of a list

---

Advanced Classes:

⇨ can be used to create enumerations, to customize string and byte representations of objects, to define computed and default attributes, to control how objects are logically compared to each other, and to give objects numeric-like behavior for operations like addition, subtraction, etc.

Defining Enumerators:

- ➔ Usually used to assign easy to read names to constant values in a program, increasing the readability of the code
- ➔ They can also be used as hash values, and you can iterate over them like other iterables
- ➔ Enumerations are defined using the class syntax
- ➔ To create an enum class, you use the Enum super-class: ***class Fruit(Enum):***
- ➔ Assigning values, your variables are in all caps, constants → You can use these constants in your code instead of having to use hard-coded numbers (each constant set to a different numeric value)
- ➔ To access a constant, use class.CONSTANT, ex: ***print(Fruit.APPLE)***
- ➔ When declared, enum constants get a name and a value, ex: ***Fruit.APPLE.name*** and ***Fruit.APPLE.value*** (so if APPLE = 1, then the name would return ***APPLE*** and the value ***1***.)
- ➔ Keys cannot be duplicated, but values can be.

```python
from enum import Enum, unique, auto

@unique
class Fruit(Enum):
    APPLE = 1
    BANANA = 2
    ORANGE = 3
    TOMATO = 4
    PEAR = auto()

def main():
    # enums have human-readable values and types
    print(Fruit.APPLE)
    print(type(Fruit.APPLE))
    print(repr(Fruit.APPLE))

    # enums have name and value properties
    print(Fruit.APPLE.name, Fruit.APPLE.value)

    # print the auto-generated value
    print(Fruit.PEAR.value)

    # enums are hashable - can be used as keys
    myFruits = {}
    myFruits[Fruit.BANANA] = "Come Mr. Tally-man"
    print(myFruits[Fruit.BANANA])
```

```
Fruit.APPLE
<enum 'Fruit'>
<Fruit.APPLE: 1>
APPLE 1
5
Come Mr. Tally-man
```

- ➔ If you want to not allow duplicate values, you can use the ***@unique*** decorator, which must also be imported from the ***enum*** module. When used, duplicate values will throw ***ValueError***.
- ➔ You can automatically assign values to enum keys using the ***auto()*** function from the ***enum*** module. To use this, you would set a key equal to ***auto()***, ex: ***PEAR = auto()***
- ➔ To use as hash values, see code ⇨

Class String Functions (representation):

- ➔ Converting different values into string representations, like the ***str*** formatting function
- ➔ Four main functions used to convert objects to strings:
  - ***object.__str__(self)*** ⇨ called when ***str(object), print(object), "{0}", .format(object)***
    - Informal representation of the object, nicely formated and human-readable, not expectation that it is a valid Python expression
  - ***object.__repr__(self)*** ⇨ called when ***repr(object)***
    - This function should try to return a Python expression that could be used to recreate the object with the same value, not always possible with complex objects, in which case it should just contain useful data about the object (often used for debugging purposes)
    - If you override ***.__repr__()*** but not ***.__str__(), .__str__()*** will be used to represent objects as strings
  - ***object.__format__(self, format_spect)*** ⇨ called when ***format(object, format_spec)***
    - Takes a format spec as an argument, called to format an object with that spec.
    - Most classes just delegate to the ***.__str__()***
  - ***object.__bytes__(self)*** ⇨ called when ***bytes(object)***
    - Not exactly a string function. It handles the conversion of an object to a bytes object. Used when you want to pass data as a set of bytes.
- ➔ By default, a Python class will represent the following data this way, printing an object, representing the class name and its place in memory:

```
class Person():
    def __init__(self):
        self.fname = "Joe"
        self.lname = "Marini"
        self.age = 25
```

⇨

```
<__main__.Person object at 0x7fc3f81cfee0>
<__main__.Person object at 0x7fc3f81cfee0>
Formatted: <__main__.Person object at 0x7fc3f81cfee0>
```

- ➔ But the .__repr__() function can be overridden as so, but without the .__str__() being overridden as well, so .__repr__() is being called in both cases. These are the results:

```
def __repr__(self):
    return "<Person Class - fname:{0}, lname:{1}, age{2}>".format(
        self.fname, self.lname, self.age)
```

⇨

```
<Person Class - fname:Joe, lname:Marini, age25>
<Person Class - fname:Joe, lname:Marini, age25>
Formatted: <Person Class - fname:Joe, lname:Marini, age25>
```

- ➔ With the .__str__() function overridden as well, we get these results:

```
def __str__(self):
    return "Person ({0} {1} is {2})".format(self.fname, self.lname, self.age)
```

⇨

```
<Person Class - fname:Joe, lname:Marini, age25>
Person (Joe Marini is 25)
Formatted: Person (Joe Marini is 25)
```

- ➔ The ***.__repr__()*** gives a nice, informational format that can be used for debugging, and the ***.__str__()*** gives a nice, clean, and readable version

- ➔ Lastly, overriding the bytes function, creating a string to hold the objects data, or values. This also uses the string class's encode function to ***encode()*** into bytes, this time UTF-8
  - This returns a UTF-8 encoded string preceded by a b' to tell you it is in the bytes format

Class Attribute Functions:

- ➔ Methods that Python provides that can be used to control how attributes are accessed on an object. Whenever attributes are retrieved or set, Python calls one of the following functions:
  - ***object.__getattribute__(self, attr)*** ⇨ called when ***object.attr*** → called unconditionally, every time an attribute name is requested and also called by Python any time it goes to look up any methods on your class, so be very careful with it.
  - ***object.__getattr__(self, attr)*** ⇨ called when ***object.attr*** → when a requested attribute cannot be found on an object → can be used to redefine the attributes already given to objects
  - ***object.__setattr__(self, attr, val)*** ⇨ called when ***object.attr = val*** → when an attribute value is set on the object
    - REMINDER: when overriding setattr, always have an else statement calling the super class to avoid raising attribute exceptions
  - ***object.__delattr__(self)*** ⇨ called when ***del object.attr*** → to delete an attribute
  - ***object.__dir__(self)*** ⇨ called when ***dir(object)*** → when the dir function is called, enabling a developer to discover dynamically what attribute your object supports

```
(50, 75, 100)
#324b64
(125, 200, 86)
#7dc856
125
['hexcolor', 'rgbolor']
```

- ➔ Adding support for attributes can really extend the features that your classes support and provide a way to reuse existing attributes in new ways.

```python
class myColor():
    def __init__(self):
        self.red = 50
        self.green = 75
        self.blue = 100

    # use getattr to dynamically return a value
    def __getattr__(self, attr):
        if attr == "rgbcolor":
            return (self.red, self.green, self.blue)
        elif attr == "hexcolor":
            return "#{0:02x}{1:02x}{2:02x}".format(
                self.red, self.green, self.blue)
        else:
            raise AttributeError

    # use setattr to dynamically return a value
    def __setattr__(self, attr, val):
        if attr == "rgbcolor":
            self.red = val[0]
            self.green = val[1]
            self.blue = val[2]
        else:
            super().__setattr__(attr, val)

    # use dir to list the available properties
    def __dir__(self):
        return ("rgbolor", "hexcolor")


def main():
    # create an instance of myColor
    cls1 = myColor()
    # print the value of a computed attribute
    print(cls1.rgbcolor)
    print(cls1.hexcolor)

    # set the value of a computed attribute
    cls1.rgbcolor = (125, 200, 86)
    print(cls1.rgbcolor)
    print(cls1.hexcolor)

    # access a regular attribute
    print(cls1.red)

    # list the available attributes
    print(dir(cls1))
```

Class Numerical Operators:

- ➔ Using special class methods, you can give your classes abilities that they do not natively have from Python but that other built-in objects have, such as the ability to emulate numeric values in order to support operations like addition, etc. by overriding class methods. (The left table shows normal operations, and the right is for in-place operations

➔ See the code below for examples

Numeric Function	Called When
`object.__add__(self, other)`	self + other
`object.__sub__(self, other)`	self – other
`object.__mul__(self, other)`	self * other
`object.__div__(self, other)`	self / other
`object.__floordiv__(self, other)`	self // other
`object.__pow__(self, other)`	self ** other
`object.__and__(self, other)`	self & other
`object.__or__(self, other)`	self | other

Numeric Function	Called When
`object.__iadd__(self, other)`	self += other
`object.__isub__(self, other)`	self –= other
`object.__imul__(self, other)`	self *= other
`object.__itruediv__(self, other)`	self /= other
`object.__ifloordiv__(self, other)`	self //= other
`object.__ipow__(self, other)`	self **= other
`object.__iand__(self, other)`	self &= other
`object.__ior__(self, other)`	self |= other

```
class Point():
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return "<Point x:{0},y:{1}>".format(self.x, self.y)

    # implement addition
    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    # implement subtraction
    def __sub__(self, other):
        return Point(self.x - other.x, self.y - other.y)

    # implement in-place addition
    def __iadd__(self, other):
        self.x += other.x
        self.y += other.y
        return self
```

```
def main():
    # Declare some points
    p1 = Point(10, 20)
    p2 = Point(30, 30)
    print(p1, p2)

    # Add two points
    p3 = p1 + p2
    print(p3)

    # subtract two points
    p4 = p2 - p1
    print(p4)

    # Perform in-place addition
    p1 += p2
    print(p1)
```

```
<Point x:10,y:20> <Point x:30,y:30>
<Point x:40,y:50>
<Point x:20,y:10>
<Point x:40,y:50>
```

Class Comparison Operators:

Comparison Function	Called When
`object.__gt__(self, other)`	self > other
`object.__ge__(self, other)`	self >= other
`object.__lt__(self, other)`	self < other
`object.__le__(self, other)`	self <= other
`object.__eq__(self, other)`	self == other
`object.__ne__(self, other)`	self != other

➔ It is also possible to use special methods to allow objects to compare themselves to one another. Each compares the object it is called on to the object that is referred to as other

➔ This also offers a way to use the built-in ***.sorted()***, which will also use these comparisons to sort

```
def main():
    # define some employees
    dept = []
    dept.append(Employee("Tim", "Sims", 5, 9))
    dept.append(Employee("John", "Doe", 4, 12))
    dept.append(Employee("Jane", "Smith", 6, 6))
    dept.append(Employee("Rebecca", "Robinson", 5, 13))
    dept.append(Employee("Tyler", "Durden", 5, 12))

    # Who's more senior?
    print(bool(dept[0] > dept[2]))
    print(bool(dept[4] < dept[3]))

    # sort the items
    emps = sorted(dept)
    for emp in emps:
        print(emp.lname)
```

```
False
True
Doe
Sims
Durden
Robinson
Smith
```

```
class Employee():
    def __init__(self, fname, lname, level, yrsService):
        self.fname = fname
        self.lname = lname
        self.level = level
        self.seniority = yrsService

    # implement comparison functions by emp level
    def __ge__(self, other):
        if self.level == other.level:
            return self.seniority >= other.seniority
        return self.level >= other.level

    def __gt__(self, other):
        if self.level == other.level:
            return self.seniority > other.seniority
        return self.level > other.level

    def __lt__(self, other):
        if self.level == other.level:
            return self.seniority < other.seniority
        return self.level < other.level

    def __le__(self, other):
        if self.level == other.level:
            return self.seniority <= other.seniority
        return self.level <= other.level

    def __eq__(self, other):
        return self.level == other.level
```

Logging:

- ➔ Logging is important, because it enables your program to record events as it is executing code, thus capturing and recording events for later analysis.
- ➔ It is like the black box of an airplane, always recording data about how things are functioning. If something unexpected happens, you can use the data to address issues in your program and is highly customizable and can provide as much or as little detail as is needed.

```
• import logging
• logging.debug("debug-level message")
• logging.info("info-level message")
• logging.warning("warning-level message")
```

Basic Logging:

Message Level	Logging API	Description
DEBUG	`logging.debug()`	Diagnostic information useful for debugging
INFO	`logging.info()`	General information about program execution results
WARNING	`logging.warning()`	Something unexpected, or an approaching problem
ERROR	`logging.error()`	Unable to perform a specific operation due to problem
CRITICAL	`logging.critical()`	Program may not be able to continue, serious error

- ➔ Use ***logging.basicConfig()*** to set the default logging level, the minimum level you want it to report, as well as setting the location to which you want it to log, as opposed to your terminal standard-out window.
- ➔ You can also config the filemode, which by default is append mode, meaning it will just continue to append log messages every time the program is run, rather than starting over with a new log. “w” sets it to rewrite mode.

Custom Logging:

`%(asctime)s`	Human readable date format when the log record was created
`%(filename)s`	File name where the log message originated
`%(funcName)s`	Function name where the log message originated
`%(levelname)s`	String representation of the message level (DEBUG, INFO, etc.)
`%(levelno)d`	Numeric representation of the message level
`%(lineno)d`	Source line number where the logging call was issued (if available)
`%(message)s`	The logged message string itself
`%(module)s`	Module name portion of the filename where the message was logged

- ➔ ***logging.basicConfig()*** can take two additional arguments, ***format*** and ***datefmt***
- ➔ ***format*** → specifies a string that controls the precise formatting of the output message that is sent to the log
- ➔ ***Datefmt*** → if the ***format*** contains a date specifier, then this is used to format how the date and time are represented
  Ex: ***datestr = “%m/%d/%Y %I: %M %S %p”***

```
extData = {'user': 'joem@example.com'}

def anotherFunction():
    logging.debug("This is a debug-level log message",
                  extra=extData)

def main():
    # set the output file and debug level, and
    # use a custom formatting specification
    fmtStr = "%(asctime)s: %(levelname)s: %(funcName)s " \
             "Line:%(lineno)d User:%(user)s %(message)s"
    dateStr = "%m/%d/%Y %I:%M:%S %p"
    logging.basicConfig(filename="output.log",
                        level=logging.DEBUG,
                        format=fmtStr,
                        datefmt=dateStr)

    logging.info("This is an info-level log message",
                 extra=extData)
    logging.warning("This is a warning-level message",
                    extra=extData)
    anotherFunction()
```

```
10/04/2022 03:02:43 PM: INFO: main Line:22 User:joem@example.com This is an info-level log message
10/04/2022 03:02:43 PM: WARNING: main Line:24 User:joem@example.com This is a warning-level message
10/04/2022 03:02:43 PM: DEBUG: anotherFunction Line:8 User:joem@example.com This is a debug-level log message
```

- ➔ The previous example is how to set up the datetime format to show month, day, year and then hour, minute, seconds, and am or pm.

➔ Each logging function takes a parameter called *extra*, which can be set to a dictionary object that contains key-value pairs to be included in the output for information such as user info

---

Python Comprehensions:

⇨ can be applied to lists, sets, and dictionaries and work similarly to the ***.map()*** and ***.filter()*** functions to apply functions to a collection of items: ***[output expression - variables]***

List Comprehensions:

- ➔ Compare the map/filter function to the list comprehension version ⇨ ⇨ ⇨
- ➔ Predicate condition ⇨ an if statement or other kind of statement that specifies under what conditions to perform the expression supplied at the beginning of the comprehension

```
def main():
    # define two lists of numbers
    evens = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
    odds = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

    # Perform a mapping and filter function on a list
    evenSquared = list(
        map(lambda e: e**2, filter(lambda e: e > 4 and e < 16, evens)))
    print(evenSquared)

    # Derive a new list of numbers frm a given list
    evenSquared = [e ** 2 for e in evens]
    print(evenSquared)

    # Limit the items operated on with a predicate condition
    oddSquared = [e ** 2 for e in odds if e > 3 and e < 17]
    print(oddSquared)
```

Dictionary Comprehensions:

- ➔ Used when you need a dictionary so you have a key for each value you need to save inside of a collection of values.
- ➔ The first half of the comprehension sets up the key: value pair to be created
- ➔ Avoid having more than two expressions in any one comprehension. The dict comp example here is about as complicated as a comprehension should ever get

```
ctemps = [0, 12, 34, 100]

# Use a comprehension to build a dictionary
tempDict = {t: (t * 9/5) + 32 for t in ctemps if t < 100}
print(tempDict)
print(tempDict[12])

# Merge two dictionaries with a comprehension
team1 = {"Jones": 24, "Jameson": 18, "Smith": 58, "Burns": 7}
team2 = {"White": 12, "Macke": 88, "Perce": 4}
newTeam = {k: v for team in (team1, team2) for k, v in team.items()}
print(newTeam)
```

newTeam = {k: v for team in (team1, team2) for k, v in team.items()}

↑ Here we have two loops, one over the two dicts and one over the players in the dicts

Set Comprehensions:

- ➔ Reminder: sets contain only unique items, no duplicates, so set comprehensions are good when you want to weed out duplicates. Here we compare a list to a set comprehension
- ➔ Use {} to create a set. When you do not provide key: value pairs, it will create a set.

```
ctemps = [5, 10, 12, 14, 10, 23, 41, 30, 12, 24, 12, 18, 29]

# build a set of unique Fahrenheit temperatures
ftemps1 = [(t * 9/5) + 32 for t in ctemps]
ftemps2 = {(t * 9/5) + 32 for t in ctemps}
print(ftemps1)
print(ftemps2)

# build a set from an input source
sTemp = "The quick brown fox jumped over the lazy dog"
chars = {c.upper() for c in sTemp if not c.isspace()}
print(chars)
```