

Python For Engineers II



Where we left off...

- Last workshop, we covered basics of Python.
 - Integer, float objects.
 - Basic strings.
 - Control Flow.
 - Functions.
- In this workshop, we're going to learn:
 - Extra string features.
 - Lists, Tuples, and Dictionaries.
 - Object-Oriented Programming.
 - Libraries and how to use them.

Strings – Formatting with F-Strings

- Python provides many ways to format data and strings.
- F-strings are a fast and convenient way to take multiple, existing data and put it all together in a single string.

```
# structure  
# put expression between '{' and '  
f'characters {<expression>}'  
f"characters {<expression>}"
```

```
# f-strings allow you to inline format  
# variables into a string without having to  
# convert them or use weird specifiers.
```

```
a, b, c = 1, 1.0, '1!'  
print(f"a = {a}, b = {b}, c = {c}")  
pwr = 3  
s = f' {(a+a)} ^ {pwr} = {(a+a)**pwr} '  
print(s)
```

Lists I

- Python's way of doing arrays.
 - A container that can hold multiple pieces of data under one name. A table if you will.
- Can hold **any** Python object, even lists themselves as lists are also objects.
- Lists can dynamically grow so you can keep adding more and more data onto them as long as your system has enough RAM.
- Since lists can hold multiple objects at once, they are a Sequenced Object.

structure on list creation.

```
<var name> = [ <comma separated objects> ]
```

example

```
l1 = [] # empty list
```

```
l2 = [1] # contains one element.
```

lists can hold ANY kind of object.

```
l3 = [True, l2, 99, 6e-3]
```

lists can be added together

```
l1 += l2 + l3
```

```
print(l1)
```


Lists II

```
# accessing individual list data  
# is done by indexing.  
print(l1[2])
```

```
# lists can be looped over!  
for item in l1:  
    → print(item)
```

```
# print how many objects are in the list  
print( len(l1) )
```

```
# another way of looping by using 'len' and indexing.  
for i in range(len(l1)):  
    → print(l1[i])
```

```
# 'in' expression searches for values.  
if 5 in l1:  
    → print('l1 has a 5!')
```

Tuples

- Similar to Lists but Tuples are not allowed to overwrite their elements.
- You can add tuples like lists.
- Most times you use a tuple is for returning multiple items from a function.

```
# structure of tuple creation  
<var name> = ( <comma separated objects> )
```

```
# example  
point_3D = (1, 5.0, 3)  
print(point_3D)
```

```
# like lists, tuples can be indexed.  
print(f'x axis: {point_3D[0]}')
```

```
# like lists, tuples can be looped.  
for coord in point_3D:  
    → print(coord)
```

```
# like lists, tuples can be added together  
double_point = point_3D + (1.0, 1.0, 10.0)
```

```
# tuple of tuples!  
line = (point_3D, (2, -10.5, 9))
```

```
# unlike lists, tuple values CANNOT be changed!  
point_3D[2] = 10.0 # ERROR.
```

Slice Expressions

- Sometimes you only want to access a portion of a string, tuple, or list. How can we do that without having to loop through a specific range?
 - Answer: **Slicing**.
- Basic idea: gives you only a portion of a sequenced object that you need.
- Slicing an object gives you the same object type back.
 - IE if you slice a list, the slice expression gives back a list object.

```
name = 'peter griffin'
print(name[1:])      # eter griffin
print(name[:7])      # peter g
print(name[0:10:2])  # ptrgi
```

```
# structure of a slice expression.
# start, stop, and step represent expressions.
<seq-obj> [ <start> : <stop> : <step> ]
<seq-obj> [ <start> : <stop> ]
<seq-obj> [ <start> : ]
<seq-obj> [ <start> :: <step> ]
<seq-obj> [ : <stop> : <step> ]
<seq-obj> [ : <stop> ]
<seq-obj> [ : ]
```

```
nums = [ 1,2,3,4,5,6,7,8,9,10 ]
print(nums[5:])    # slice from index 5 and till end.
print(nums[:5])    # slice from first till index 5.
print(nums[3:5])   # slice from index 3 till index 5.
```

```
# print every odd number
print(nums[::2])   # slice from start to end, every 2 indexes.
```

```
# print every even number
print(nums[1::2])  # slice from index 1 to end, every 2 indexes.
print(nums[::])    # simply prints whole list
```

Sequence Repeating

- We saw last workshop that strings can be repeated by multiplying them with an integer number.
- You can also do the same for other sequencing objects like lists and tuples!

```
# useful in presizing a list but with initial values
```

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

```
print(nums)
```

```
num_players = int( input('how many players?: ') )
```

```
scores = [0] * num_players # start everybody off with 0 points.
```

```
print(scores)
```

```
cheer = 'sis boom bah! '
```

```
print(cheer * 4)
```


Dictionaries I

- Allows us to map keys* (as objects) to values (also as objects).
- Mostly used to map string keys or other sequential data to other data in such that it establishes a relationship as a key-value pair.
- One of the most valuable object types in the software world.

***lists can't be used as keys, tuples are allowed.**

```
# structure of dictionary creation.
# expr is short for expression.
<var name> = { <key expr> : <value expr> }
<var name> = {
    —><key expr1> : <value expr1> ,
    —>...
    —><key exprN> : <value exprN> ,
}
```

```
# example
empty_dict = {}
```

```
spouses = {
    —>'peter griffin': 'lois pewterschmidt',
    —>'homer simpson': 'marge bouvie',
    —>'stan smith': 'francine "ling" dawson',
    —>'bob belcher': 'linda genarro',
}
```

```
# adding a new entry.
spouses['phillip j fry'] = 'turanga leela'
```

```
# oops, we misspelled marge's maiden name! let's fix it.
spouses['homer simpson'] = 'marge bouvier'
```

```
# remove entries.
del spouses['phillip j fry']
```

Dictionaries II

check if an entry exists.

```
print('peter griffin' in spouses)
```

dictionaries can be looped over.

loop gives the key, not value.

```
for guy in spouses:
```

```
    → print(f'{guy} is married to {spouses[guy]}')
```

dictionaries have a length as well.

```
print(len(spouses))
```

Lambda Functions

- Useful when you need a small function that only returns a singular expression!
- No need for a return statement!
- Creates a direct, unnamed function object.
- Lambdas are technically expressions.

structure

```
lambda <parameter names> : <expression>
```

example

```
sqr = lambda x: x**2
```

```
print(sqr(9))    # prints 81
```

```
pwrize = lambda a,b: a**b * a
```

```
print(pwrize(2,3)) # prints 16
```

Object-Oriented Programming Overview

- Everything in Python is an object.
- Object-oriented is when:
 - Objects are paired with special functions that provide actions for a type of object. These special functions are called **methods**.
 - The programmer can create custom object types with their own custom methods.
- Custom object types must be modelled by a construct known as a **class**.
- Named, individual pieces of data that custom objects hold are called **fields, members, or properties**.
 - Python usually calls them **attributes**.
 - Ex: the **real** and **imag** in `-1j.real` & `1j.imag`
- Each unique object holds their own independent copy of the properties defined by the class they model.
- *Think of a class like a blueprint describing an object type, where properties describe the data of the objects and methods describe the actions that you can do with that data.*

Object Oriented Programming – Structure of Classes

- Important to know: `self` in class methods ALWAYS refers to the object that's using/calling the method.

`# structure`

`class <name>:`

→ `<optional named props & their expressions>`

→

→ `# constructor method. optional but usually necessary`

→ `# for custom property initializing.`

→ `def __init__(self, <parameters>):`

→ `self.<prop1> = <param1>`

→

→ `...`

→ `self.<propN> = <paramN>`

→

→ `# a class can have as many methods or none.`

→ `def <method nameN>(self, <parameters>):`

→ `<statements>`

Object Oriented Programming – Class Example I

```
class Dog:
```

```
    kind = 'canine' # property shared by all objects of  
    this class.
```

```
    def __init__(self, name):  
        self.name = name # property unique to each  
        independent object.
```

```
d = Dog('Scooby Doo') # this is calling '__init__' in Dog
```

```
e = Dog('Air Bud')
```

```
print(d.kind, d.name)
```

```
print(e.kind, e.name)
```

Object Oriented Programming – Class Example II: Objects & Methods

```
class Dog:
    → kind = 'canine'
    → def __init__(self, name):
    →     → self.name = name
    →
    → # method that defines an action for the object type.
    → # in this case, Dogs bark.
    → def bark(self):
    →     → print(f'{self.name}: "woof!"')

fiddo = Dog('fido')

# Calling a method is the combination of
# a function call and accessing properties.
fiddo.bark()
```

Object Oriented Programming – Inheritance

- What happens if we want to model objects that have shared as well as distinct qualities?
- The answer: **Inheritance**
- Create a class(es) that contains all the qualities the objects have in common.
- Create the class(es) that model the differences.

structure of class inheritance

single inheritance.

```
class <name> (<class_name>):
```

```
    → ... <remaining_class_structure>
```

multiple inheritance

```
class <name> (<class_name1>, ..., <class_nameN>):
```

```
    → ... <remaining_class_structure>
```

Object Oriented Programming – Inheritance Example I

- Classes that inherit from other classes gain their methods and properties.

```
class Weapon:
```

```
→ # the var names in params are NOT  
→ # the same var for self.<var name>  
→ def __init__(self, power, weight):  
→     self.damage = power  
→     self.weight = weight
```

```
# class Bow & Sword inherit from Weapon
```

```
class Bow(Weapon):
```

```
→ def __init__(self, power, weight, wood_type):  
→     # 'super' sets up call to __init__  
→     # of the class we inherit from.  
→     super().__init__(power, weight)  
→     self.material = wood_type
```

```
class Sword(Weapon):
```

```
→ def __init__(self, power, weight, two_handed):  
→     super().__init__(power, weight)  
→     self.hold_type = two_handed
```

Object Oriented Programming – Inheritance Example II

- This is an example where a class is derived from two existing classes.
- Multi-inheritance is typically not recommended due to complexity and maintainability of the code.
- Use it only if necessary.
- When you have single inheritance, use **super()**.
- For multi-inheritance, you must manually call constructors

```
class Weapon:
    → def __init__(self, power, weight):
    →     self.damage = power
    →     self.weight = weight
```

```
class Magic:
    → def __init__(self, magic_power):
    →     self.magic_power = magic_power
    →
    → def cast(self, multiplier, intercept):
    →     return (self.magic_power * multiplier) + intercept
```

```
class MagicStaff(Weapon, Magic):
    → def __init__(self, pwr, weight, magic_pwr):
    →     # for multi-inheriting, use manual constructor calls.
    →     Weapon.__init__(self, pwr, weight)
    →     Magic.__init__(self, magic_pwr)
```

```
staff = MagicStaff(10, 2, 3.5)
print(f'with this staff, I cast a spell and do {staff.cast(staff.damage, 35)} damage!')
```


Object Oriented Programming – Polymorphism I

- What do you do when you have to model objects that not only have shared qualities but actions as well? Answer: **Polymorphism**.
- There's two kinds of polymorphism.
 - Method overriding – Change the code of an existing method for a derived class.
 - Duck Typing – Run code based on the **type** of the object.
- The example to the right is an example of method overriding.
- Method Overriding is best when you only need polymorphism for custom objects.
- Duck Typing is best when you have to model objects that have shared actions but little-to-no shared qualities.

```
class Dog:
    → kind = 'canine'
    → def __init__(self, name):
    →     → self.name = name
    →
    → def bark(self):
    →     → print(f'{self.name}: "woof!"')
```

```
class StBernard(Dog):
    → def __init__(self, name):
    →     → super().__init__(name)
    →
    → # override 'bark' from 'Dog' class.
    → def bark(self):
    →     → print(f'{self.name}: "ruff!"')
```

```
class Chihuahua(Dog):
    → def __init__(self, name):
    →     → super().__init__(name)
    →
    → def bark(self):
    →     → print(f'{self.name}: "arff!"')
```

```
Dog('fido').bark()
StBernard('beethoven').bark()
Chihuahua('eevee').bark()
```

Object Oriented Programming – Polymorphism II: Duck Typing

```
class Car:
```

```
    def __init__(self):
```

```
        self.gas = 100.0
```

```
    def drive(self):
```

```
        if self.gas <= 0.0:
```

```
            return
```

```
        # drain gas when we drive
```

```
        self.gas -= 10.0
```

```
        print('vroom')
```

```
class Airplane:
```

```
    def __init__(self):
```

```
        self.gas = 100.0
```

```
    def fly(self):
```

```
        if self.gas <= 0.0:
```

```
            return
```

```
        self.gas -= 10.0
```

```
        print('phwee')
```

two ways to do duck typing: check type or check properties/attributes
better to just check by type unless you're doing something special.

```
def move1(vehicle):
```

```
    if isinstance(vehicle, Car):
```

```
        vehicle.drive()
```

```
    elif isinstance(vehicle, Airplane):
```

```
        vehicle.fly()
```

```
def move2(vehicle):
```

```
    if hasattr(vehicle, 'drive'):
```

```
        vehicle.drive()
```

```
    elif hasattr(vehicle, 'fly'):
```

```
        vehicle.fly()
```

```
move1(Car())
```

```
move1(Airplane())
```

```
def product(obj):
```

```
    if isinstance(obj, int):
```

```
        return obj * 2
```

```
    elif isinstance(obj, float|complex):
```

```
        return obj * 3
```

```
    elif isinstance(obj, str):
```

```
        return obj * 4
```

```
    elif isinstance(obj, list|tuple):
```

```
        return sum(obj) * 10
```

```
print( product(1) )
```

```
print( product(2j) )
```

```
print( product('h') )
```






















```
print( product([1,2,3]) )
```

```
print( product((2,4,6)) )
```

Builtin List Methods

- Special methods exclusively for the List object.
- Example works by starting out every list with two burgers and a box of fries.
- The food items represent different possible objects.
- ***Notice what some of the methods return.***
- Tuples only have **count** and **index** for builtin methods, so no special slide for tuple methods.

PYTHON LIST METHODS

	<code>.append()</code> → 
	<code>.count()</code> → 2
	<code>.copy()</code> → 
	<code>.index()</code> → 2
	<code>.reverse()</code> → 
	<code>.remove()</code> → 
	<code>.insert(1, )</code> → 
	<code>.pop(1)</code> → 
	<code>.pop()</code> → 

Builtin Dictionary Methods

```
student_gpas = {  
    → 'ahmed smithy': 3.6,  
    → 'austin powers': 2.3,  
    → 'johanne rodriguez': 3.5,  
    → 'simon jarrett': 2.9,  
    → 'lisa simpson': 5.0,  
    → 'minerva baskil': 3.33,  
}  
  
# removes all keys and values.  
student_gpas.clear()  
  
# makes a copy of the dictionary values.  
print(student_gpas.copy())  
  
# 'fromkeys(keys)' or 'fromkeys(keys, value)'  
# creates a dictionary from existing data.  
# class method as opposed to object method.  
staff = dict.fromkeys(('edna krabapple', 'armin tamzarian', 'groundskeeper willy'), 4.0)  
print(staff)  
  
# get(keyname, value)  
# 'value' parameter is optional.  
# It's for giving a default value if key isn't found.  
print(student_gpas.get('lisa simpson', 2.0))  
  
# this gets a viewing object of the dictionary's keys and values.  
# the viewing object is convertible to a list/tuple.  
print(list(student_gpas.keys()))  
print(list(student_gpas.values()))
```

Scoping – Visibility of Objects

- An object only exists from inside the region it is created. This is called the **scope** [of that object].
- Objects created inside a function have Local scope.
- **Python's Scoping rules are complex.**
- If-statements, loops **do not** have their own scopes.
- Classes do have their own scope.

```
times_f_called = 0
```

```
def f(a):
```

```
→ # to access a global variable.
```

```
→ # you need 'global' keyword.
```

```
→ global times_f_called
```

```
→ times_f_called += 1
```

```
→ return a * 2
```

```
f(10)
```

```
print(times_f_called) # prints 1
```

```
# objects created here are called 'global  
variables'.
```

```
# they have global scope, meaning they can be  
used in any lower or equivalent scope.
```

```
a = 1
```

```
# functions have their own scope.
```

```
def f(a):
```

```
→ a *= 2 # this 'a' is separate from 'a'  
outside the function.
```

```
f(a)
```

```
print(a) # prints 1
```


Exceptions – Structure

structure of a try-except statement

try:

→<statements>

except (<optional exception types>):

→<statements that handle error>

else: # optional part

→<statements when no error happens>

finally: # also optional part

→<statements when **try-except** block finishes>

raising an exception

raise **Exception**('message')

- Sometimes you need to run code with the possibility of an error happening.
- We sometimes need to handle that error when it occurs.
- **Try-Exception** Statements do the job.
- When we also need, we can raise our own exceptions back to the user!

Exceptions - Example

```
data = input('enter a number: ')\ntry:\n    → i = int(data)\nexcept ValueError:\n    → print(f'{data} can\'t be converted to int. ')\nelse:\n    → if i < 0:\n        → raise ValueError('no negative numbers. ')\n    → print(i)
```

Libraries & Importing Library Modules – Importing Structure

- Libraries are a set of ready-to-use Python code that is packaged under a name.
- Accessing these libraries requires a special action called *importing*.

structure of importing libraries.

```
import <library name>
```

```
import <library name1 , ... , library nameN>
```

```
from <library name> import <object_name>
```

```
from <library name> import <object_name1 , ... , object_nameN>
```

structure of accessing objects within a library.

```
<library name> . <object_name>
```

Libraries & Importing Library Modules – Importing Example

```
# example
import os
# print out the current, working directory.
print(os.getcwd())

# alternate equivalent
from os import getcwd
print(getcwd())
```



End of Python For Engineers II

- Thank you for attending.
- Next time: **Python for Engineers III.**
 - Learn about the Python Standard library.
 - Word problems to build up our skills.
 - Using PIP to get use third-party libraries.
 - Reading syntax and runtime errors.