Python Track

Containers, Built-in functions, and Classes

Lecture Flow

- Classes
- Iterators & Generators
- Collections & Containers
- Python Built-in Functions
- Common Pitfalls and Best Practices

Classes

Classes and Objects

- Python is an object oriented programming language.
- Almost everything in Python is an object, with its properties and methods.
- A Class is like an object constructor, or a "blueprint" for creating objects.

Class - Example

```
class MyClass:
    def __init__(self):
        self.x = 5

p1 = MyClass()
print(p1.x) # 5
```

The __init__() function

- Classes have a function called **__init__()**, which is always executed when the class is being initiated.

- Used to assign values to object properties

The __init__() function (Continued)

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
p1 = Person("John", 36)
```

- The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

Object Methods

- Objects can also contain methods. Methods in objects are functions that belong to the object.

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc() # Hello my name is John
```

Iterators & Generators

Iterators and Iterables

- Iterable is an object, that one can iterate over. It generates an Iterator when passed to **iter()** method.
- Iterators have the **__next__()** method, which returns the next item of the object.
- Strings, Lists, tuples, dictionaries and sets are iterables.

Iterators - Example

```
my_tuple = ("apple", "banana", "cherry")

my_iter = iter(my_tuple)

print(next(my_iter)) ?
```

Iterators - Example

```
my_tuple = ("apple", "banana", "cherry")

my_iter= iter(my_tuple)

print(next(my_iter)) # apple

print(next(my_iter)) ?
```

Iterators - Example

```
my_tuple = ("apple", "banana", "cherry")

my_iter= iter(my_tuple)

print(next(my_iter)) # apple

print(next(my_iter)) # banana

print(next(my_iter)) ?
```

Iterators - Example

```
my_tuple = ("apple", "banana", "cherry")

my_iter= iter(my_tuple)

print(next(my_iter)) # apple

print(next(my_iter)) # banana

print(next(my_iter)) # cherry
```

Iterators - Example

```
my_tuple = ("apple", "banana", "cherry")

my_iter = iter(my_tuple)

print(next(my_iter)) # apple

print(next(my_iter)) # banana

print(next(my_iter)) # cherry

print(next(my_iter)) # ?
```

Iterators - Example

```
my_tuple = ("apple", "banana", "cherry")

my_iter = iter(my_tuple)

print(next(my_iter)) # apple

print(next(my_iter)) # banana

print(next(my_iter)) # cherry

print(next(my_iter)) # StopIteration exception
```

Generators

- Python Generator functions allow you to declare a function that behaves likes an iterator

- The state of generators are maintained through the use of the keyword yield and works much like using return but it has some important differences.

Generators (Continued)

- The code in generator functions only execute when next() is called on the generator object.

- The next time the function is called, execution continues from where it left off, with the same variable values it had before yielding, whereas the return statement terminates the function completely.

Generators - Examples

```
# with out generators
def square_nums(nums):
    result = []
    for i in nums:
        result.append(i*i)
    return result

my_nums = square_nums([1,2,3])

for square in my_nums:
    print(square)
```

```
# with generator
def square_nums(nums):
    for i in nums:
        yield i*i

my_nums = square_nums([1,2,3])

for square in my_nums:
    print(square)
```

Generators - Examples

```
# with out generators
def square_nums(nums):
    result = []
    for i in nums:
        result.append(i*i)
    return result

my_nums = square_nums([1,2,3])

print(my_nums)
```

```
# with generator
def square_nums(nums):
    for i in nums:
        yield i*i

my_nums = square_nums([1,2,3])

print(next(my_nums)) # 1
print(next(my_nums)) # 4
print(next(my_nums)) # 9
```

Collections & Containers

Collections and Containers

- The collections module in Python provides different types of containers.
- A container is an object that is used to store different objects and provide a way to access the contained objects and iterate over them.
- Some of the built-in containers are Tuple, List, Dictionary, etc.

Collections - Example

- Commonly used containers provided by the collections module:

Defaultdict	`from collections import defaultdict`
Counter	`from collections import Counter`
Deque	`from collections import deque`

defaultdict

- It is used to provide some default values for the key that does not exist and without raising a KeyError.

- The default value is specified when creating the defaultdict object using the default_factory parameter.

defaultdict - Example

```
from collections import defaultdict

d = defaultdict(int)

L = [1, 2, 3, 4, 2, 4, 1, 2]

for i in L:

    d[i] += 1

print(d)
```

Counter

- It is used to keep the count of the elements in an iterable in the form of an unordered dictionary
- The key represents the element in the iterable
- The value represents the count of that element in the iterable.

Counter - Example

```
from collections import Counter


print(Counter(['B','B','A','B','C','A','B','B','A','C']))

?
```

Counter - Example

```
from collections import Counter


print(Counter(['B','B','A','B','C','A','B','B','A','C']))
# {‘B’: 5, ‘A’ : 3, ‘C’ : 2}
```

Counter - Example

```
from collections import Counter


print(Counter("ABCDEFGABCDE"))
```

?

Counter - Example

```
from collections import Counter


print(Counter("ABCDEFGABCDE"))

# {'A': 2, 'B': 2, 'C': 2, 'D': 2, 'E': 2, 'F': 1, 'G': 1}
```

Deque (Doubly Ended Queue)

- Deque (Doubly Ended Queue) is the optimized list for quicker append and pop operations from both sides of the container.
- It provides O(1) time complexity for append and pop operations as compared to list with O(n) time complexity.

deque - Example

```
from collections import deque

de = deque([1,2,3])

de.append(4)

print(de)  # ?

de.popleft()

print(de) # ?
```

deque - Example

```
from collections import deque

de = deque([1,2,3])

de.append(4)

print(de)  # [1, 2, 3, 4]

de.popleft()

print(de) # [2, 3, 4]
```

Python Built-In Functions

Built-in Functions

Function	Use	Example
`all()`	Returns True if all items in an iterable object are true	`x = all([True, True, True])`
`any()`	Returns True if any item in an iterable object is true	`x = any([False, True, False])`
`chr()`	Returns a character from the specified Unicode code	`x = chr(97)`
`ord()`	Get the ASCII value of a character	`x = ord("h")`

Built-in Functions (Continued)

Function	Use	Example
`eval()`	Evaluates and executes an expression	`x = 'print(55)'` `eval(x)`
`type()`	Returns the type of an object	`b = "Hello World"` `y = type(b)`

Built-in Functions (Continued)

Function	Use	Example
`map()`	Apply a function to elements in an iterable	`func = lambda a: len(a)` `x = map(func,('a','ba','che'))`
`filter()`	Filter elements in an iterable based on a function	`func = lambda a: a%2` `x = filter(func, [1,2,3,4])`
`zip()`	Combine elements from multiple iterables into tuples	`x = zip(['a','b','c','d'], [1,2,3,4])`

Built-in Math Functions

Function	Use	Example
`min()`	Find the lowest in an iterator	`x = min(5, 10, 25)`
`max()`	Find the highest in an iterator	`y = max(5, 10, 25)`

`sum()`	Returns the sum of all elements in a list or iterable	`total = sum(5, 10, 25)`
		`nums = [5, 10, 25]`
		`total = sum(nums)`

`abs()`	Returns the absolute (positive) value of the specified number	`x = abs(-7.25)`

Built-in Math Functions (Continued)

Function	Use	Example
`sqrt()`	Returns the square root of a number:	`import math` `x = math.sqrt(64)`
`ceil()`	Rounds a number upwards to its nearest integer	`import math` `x = math.ceil(1.4)`
`floor()`	Rounds a number downwards to its nearest integer	`import math` `x = math.floor(1.4)`

Built-in String Functions

Function	Use	Example
`len()`	Returns the length of a string	`s = "Hello world!"` `s_length = len(s)`
`split()`	Splits a string into a list based on a delimiter	`s = "a,b, c"` `result = s.split(",")`
`join()`	Concatenates elements of a list into a single string	`words = ['Hello','world']` `result = " ".join(words)`
`strip()`	Removes leading and trailing whitespaces	`s = "  Hello, World!\t\n  "` `s.strip()`
`replace()`	Replaces occurrences of a substring with another	`s = "Hello, a2sv! How is everyone in a2sv?"` `s.replace("a2sv", "A2SV")`

Common Pitfalls

- Iteratively concatenating strings using the + operator. It can be inefficient due to string immutability. Use "".join() for efficient string concatenation.

- Forgetting to include self as the first parameter in instance methods of a class.

Best Practices

- Explore the collections module for specialized data structures (e.g., Counter, defaultdict) instead of reinventing the wheel.
- Look for opportunities to use built-in functions like map, filter, and list comprehensions to avoid unnecessary loops.
- When iterating over multiple iterables in parallel, use zip for clean and efficient code.
- Initialize all necessary attributes in the __init__ method to ensure a well-defined object state.

Practice Problems

- ★ Two sum
- ★ Contains duplicate
- ★ Majority Element
- ★ Majority Element ii

Quote of the day

"The difference between ordinary and extraordinary is that little extra. Add a touch of **class** to everything you do."

– Jimmy Johnson