Django Class Notes

Clarusway

Python Review Session Class Notes

Summary

Introduction to Python

- Comments in Python start with the hash character, #
- The equal sign (=) is used to assign a value to a variable
- \ can be used to escape
- Indentation 4 spaces.
- Overall code syntax rules are stated at PEP8. There are online tools. PEP8 online check

Using Python as a Calculator

The operators +, -, * and / work just like in most other languages:

```
2 + 2
# 4
```

Data Types

String

Assign string to a variable: a = "Hello"

Expressing string with quotes, '', "", ''' ''', """ """

Multiline Strings with ''' ''' or """ """:

```
a = """Python
class"""
```

Last one is called docstring. A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. Such a docstring becomes the **doc** special attribute of that object.

Strings are immutable, which means a string value cannot be updated.

We can verify this by trying to update a part of the string which will led us to an error.

- String Length:

```
a = "Hello, World!"
print(len(a))
```

Indexes (positive and negative)

- Individual characters in a string may be chosen. If the string has length L, then the indices start from 0 for the initial character and go to L-1 for the rightmost character.
- Negative indices may also be used to count from the right end, -1 for the rightmost character through -L for the leftmost character.
- Call char by index:

```
a = "Hello World!"
print(a[1])
```

- Slicing
  - stringReference [start : pastEnd]
  - stringReference [: pastEnd]
  - stringReference [start :]
  - stringReference [:]
  - Change the order: s[::-1]
- Methods

```
upper() # Returns an uppercase version of the string
lower() # Returns a lowercase version of the string
title()
capitalize()
count(item)  # Returns the number of repetitions of the substring sub inside s.
split() # Splits at any sequence of whitespace (blanks, newlines, tabs) and
returns the remaining parts of s as a list
split(sep)  # Separator that gets removed from between the parts of the list.
startswith() # True/False
endswith() # True/False
strip([chars]) # Returns a copy of the string with both leading and trailing
characters removed
```

```
string3 = "ol eyyyyy  "

# prints the string by removing leading and trailing whitespaces
print(string3.strip())

# prints the string by removing geeks
```

```
string3 = "ol eyyyyy  "
string3.strip("y")
```

Integer

- Literal integer values may not contain a decimal point.
- Integers may be arbitrarily large and are stored exactly.
- Integers have normal operations, with usual precedence (highest listed first):
    - **: exponentiation (5**3 means 5*5*5)
    - *, multiplication
    - /, division with float result
    - //, floor divisions result int, integer division (ignoring any remainder)
    - %, modulus just the remainder from division
    - +, -: addition, subtraction

Boolean

- The Boolean data type can be one of two values, either True or False.
- The values True and False will also always be with a capital T and F
- Comparison operators are used to compare values and evaluate down to a single Boolean value of either True or False

```
==  Equal to
!=  Not equal to
<   Less than
>   Greater than
<=  Less than or equal to
>=  Greater than or equal to
```

```
x = 5
y = 8

print("x == y:", x == y)
print("x != y:", x != y)
print("x < y:", x < y)
print("x > y:", x > y)
print("x <= y:", x <= y)
print("x >= y:", x >= y)
```

Logical Operators

```
and True if both are true   x and y
or  True if at least one is true    x or y
not True only if false not x
```

- For example, they can be used to determine if the grade is passing and that the student is registered in the course. If both of these cases are true, then the student will be assigned a grade in the system.

```
print((9 > 7) and (2 < 4))  # Both original expressions are True
print((8 == 8) or (6 != 6)) # One original expression is True
print(not(3 <= 1))          # The original expression is False
```

- Falsy expressions:

```
# [] {} 0 None  # False

if []:
    print("This is true")

# Will not print anything since [] is falsy.
```

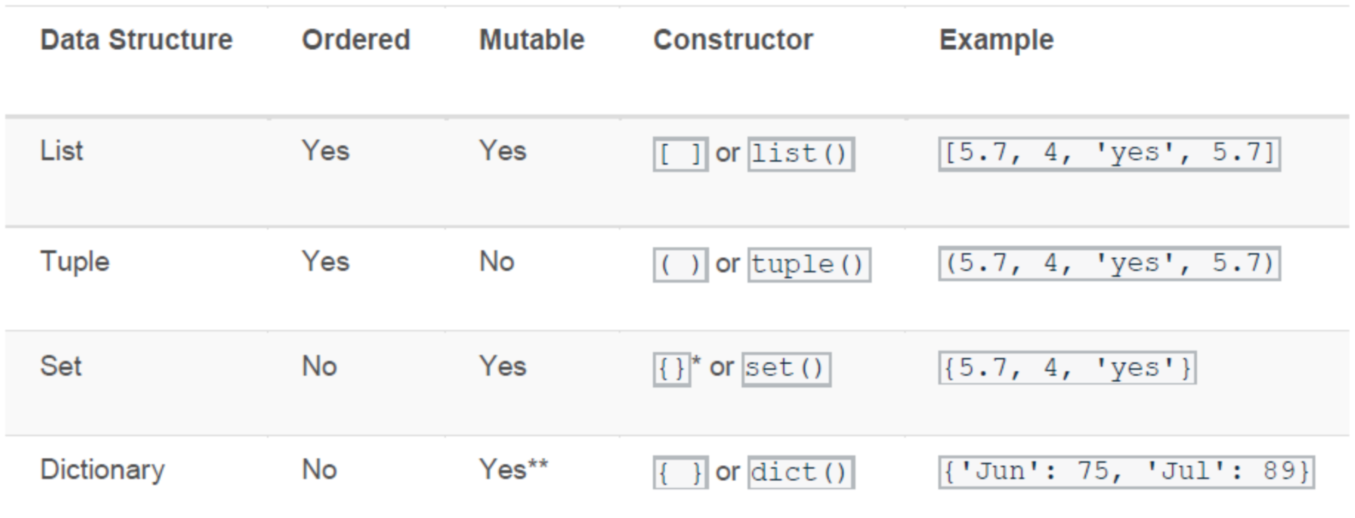
- Operational order: not and or

Float

Nothing special.

Collection Data Types

- List
- Dictionary
- Tuple
- Set



Data Structure	Ordered	Mutable	Constructor	Example
List	Yes	Yes	`[ ]` or `list()`	`[5.7, 4, 'yes', 5.7]`
Tuple	Yes	No	`( )` or `tuple()`	`(5.7, 4, 'yes', 5.7)`
Set	No	Yes	`{}`* or `set()`	`{5.7, 4, 'yes'}`
Dictionary	No	Yes**	`{ }` or `dict()`	`{'Jun': 75, 'Jul': 89}`

List

Lists are mutable sequences, typically used to store collections of items (where the precise degree of similarity will vary by application).

```
list1 = ["Apple", 1, 3.5, True, [1, 3], {"Ready": "yes"}]
```

Lists may be constructed in several ways:

- Using a pair of square brackets to denote the empty list: []
- Using the type constructor: list() or list(iterable)

```
a = list()
print(a)
```

The constructor builds a list whose items are the same and in the same order as iterable’s items.

- list('abc') returns ['a', 'b', 'c']
- list((1, 2, 3)) returns [1, 2, 3]

Python has a set of built-in methods that you can use on lists:

```
append()  # Adds a -single element- at the end of the list
```

```
stack = []

for i in range(5):
    stack.append(i)
    print(stack)

while len(stack):
    print(stack.pop())
```

```
clear()   # Removes all the elements from the list
copy()    # Returns a copy of the list
```

- The copy() method in Python returns a copy of the List. We can copy a list to another list using the = operator, however copying a list using = operator means that when we change the new list the copied list will also be changed, if you do not want this behaviour then use the copy() method instead of = operator.

```
count()   # Returns the number of elements with the specified value
extend()  # Add the elements of a list (or any iterable), to the end of the
current list
index()   # Returns the index of the first element with the specified value
insert(index, elem)  # Adds an element at the specified position
```

```
pop(index)    # Removes the element at the specified position, assigning the
popped item to a new variable
remove()  # Removes the item with the specified value (the first value)
reverse() # Reverses the order of the list
```

reverse() actually reverses the elements in the container. reversed() doesn't actually reverse anything, it merely returns an object that can be used to iterate over the container's elements in reverse order.

```
sort()    # Sorts the list
```

How to reach a value inside a list:

```
# Reach to the index no 1 inside inner list:
a = ['ali', 2, [1,2], 7]
a[2][1]
2
```

How to join two list:

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]

list3 = list1 + list2
print(list3)

# ['a', 'b', 'c', 1, 2, 3]
```

Tuples

- Less memory
- Stable, unchanged values
- Faster

Tuples are immutable sequences, typically used to store collections of heterogeneous data.

Tuples may be constructed in a number of ways:

- Using a pair of parentheses to denote the empty tuple: ()
- Using a trailing comma for a singleton tuple: "a", or ("a",)
- Separating items with commas: "a", "b", "c" or ("a", "b", "c")
- Using the tuple() built-in: tuple() or tuple(iterable)

Tuple items are ordered, unchangeable, and allow duplicate values.

To create a tuple with only one item, you have to add a comma after the item

```
thistuple = ("apple",)
print(type(thistuple))

# NOT a tuple
thistuple = ("apple")
print(type(thistuple))

# A tuple with strings, integers and boolean values:
tuple1 = ("abc", 34, True, 40, "male")

# Print the second item in the tuple:
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

Python has two built-in methods that you can use on tuples.

```
count() # Returns the number of times a specified value occurs in a tuple
index() # Searches the tuple for a specified value and returns the position of
where it was found
```

```
# Join two tuples:
tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)

tuple3 = tuple1 + tuple2
print(tuple3)

# ('a', 'b', 'c', 1, 2, 3)
```

Dictionary

Dictionaries are used to store data values in key:value pairs.

A dictionary is a collection which is ordered, changeable and does not allow duplicates.

Dictionaries are written with curly brackets, and have keys and values:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
# {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}

# You can access the items of a dictionary by referring to its key name, inside
```

```
square brackets:
x = thisdict["model"]
print(x)
# Mustang

# If there is not such a key, it will raise key error. To avoid this use get()
method.


# There is also a method called get() that will give you the same result:
x = thisdict.get("model")
print(x)
# Mustang

# If not, then it will return None (if get() is used with only one argument).
# Default None can be changed to a worning message:
thisdict.get("bran", "There is no such a key!")


# The keys() method will return a list of all the keys in the dictionary.
x = thisdict.keys()
print(x)
# dict_keys(['brand', 'model', 'year'])
# Keys can be displayed on a loop:
for key in  thisdict.keys():
  print(key)


# Add a new item to the original dictionary, and see that the keys list gets
updated as well:
thisdict["color"] = "white"


# You can change the value of a specific item by referring to its key name:
thisdict["year"] = 2022

# The update() method will update the dictionary with the items from the given
argument. The argument must be a dictionary, or an iterable object with key:value
pairs.
thisdict.update({"year": 2020})

# Adding an item to the dictionary is done by using a new index key and assigning
a value to it:
thisdict["color"] = "red"

# The pop() method removes the item with the specified key name:
thisdict.pop("model")

# The popitem() method removes the last inserted item
thisdict.popitem()


# The del keyword removes the item with the specified key name:
del thisdict["model"]
```

```
# The del keyword can also delete the dictionary completely.

# The clear() method empties the dictionary.
```

- Reaching list item is the same with lists

Dictionary Methods

Python has a set of built-in methods that you can use on dictionaries.

```
clear() # Removes all the elements from the dictionary
get()   # Returns the value of the specified key
items() # Returns a list containing a tuple for each key value pair
keys()  # Returns a list containing the dictionary's keys
pop()   # Removes the element with the specified key
popitem()   # Removes the last inserted key-value pair
update()    # Updates the dictionary with the specified key-value pairs
values()    # Returns a list of all the values in the dictionary
```

Set

Sets are used to store multiple type of items in a single variable. A set is a collection which is both unordered and unindexed. Sets are written with curly brackets. A set can contain different data types: You cannot access items in a set by referring to an index or a key.

It is possible to use the set() constructor to make a set.

```
# Create a Set
thisset = {"apple", "banana", "cherry"}

# Get the number of items in a set:
print(len(thisset))

# To add one item to a set use the add() method.
thisset.add("orange")

# To remove an item in a set, use the remove(), or the discard() method.
thisset.remove("banana")
thisset.discard("banana")

# If the item to remove does not exist, discard() will NOT raise an error.

# Remove the last item by using the pop() method:

# Sets are unordered, so when using the pop() method, you do not know which item
that gets removed.
```

Python has a set of built-in methods that you can use on sets.

```
add()   # Adds an element to the set
clear() # Removes all the elements from the set
copy()  # Returns a copy of the set
**difference()**    # Returns a set containing the difference between two or more
sets
difference_update() # Removes the items in this set that are also included in
another, specified set
discard()   # Remove the specified item
**intersection()**  # Returns a set, that is the intersection of two other sets
**intersection_update()**   # Removes the items in this set that are not present
in other, specified set(s)
isdisjoint()    # Returns whether two sets have a intersection or not
issubset()  # Returns whether another set contains this set or not
issuperset()    # Returns whether this set contains another set or not
pop()   # Removes an element from the set
remove()    # Removes the specified element
**symmetric_difference()**s # Returns a set with the symmetric differences of two
sets
symmetric_difference_update()   # inserts the symmetric differences from this set
and another
**union()** # Return a set containing the union of sets
update()    # Update the set with the union of this set and others
```

```
a = {1, 2, 3, 10, 32, 100}
b = {1, 2, 32}

a.difference(b)
a.intersection(b)
a.union(b)
```

Sets vs Lists and Tuples

Sets, unlike lists or tuples, cannot have multiple occurrences of the same element and store unordered values.

Usecase: Remove duplicate values from a list or tuple and to perform common math operations like unions and intersections.

Conditional - If

Check if the list is empty or not:

```
list11 = [1]
if list11:
    print("The list is not empty!")

# Structure:
```

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")

# You can also have an else without the elif

##############OPTIONAL#######################################
# If you have only one statement to execute, you can put it on the same line as
the if statement:
# if a > b: print("a is greater than b")

# if num_1 > num_2:
#     print(f"{num_1} is greater than {num_2}")

# a = 2
# b = 330
# print("A") if a > b else print("B")

# a = 330
# b = 330
# print("A") if a > b else print("=") if a == b else print("B")
##############OPTIONAL#######################################
```

While using if, put the most specific condition to the top.

Loops

While

With the while loop we can execute a set of statements as long as a condition is true.

```
# Ask for password until user enters correct secret:
secret = 'swordfish'
pw = ''

while pw != secret:
    pw = input("What's the secret word? ")

# Print i as long as i is less than 6:
i = 1
while i < 6:
  print(i)
  i += 1

# With the break statement we can stop the loop even if the while condition is
true
```

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

- Some libraries of python may be called like random and time. No code is perfect at the begginning. Modify as you need. Google when you can't remember. Use cheat sheets.

```
# import time

# i = 10
i = 0
# while i >= 0:
while i < 5:
    print(f"Countdown {i}")
    i += 1
    # i -= 1
    # time.sleep(1)

# print("!!!!! Fire !!!!!")
```

For

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

```
# Print each fruit in a fruit list:

fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
  print(fruit)

# Loop through the letters in the word "banana":

for x in "banana":
  print(x)


### break

# With the break statement we can stop the loop before it has looped through all
the items.

# Exit the loop when x is "banana":
  if x == "banana":
    break
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)


### continue

# With the continue statement we can stop the current iteration of the loop, and
continue with the next:

# Do not print banana:
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    continue
  print(x)
```

The range() Function

To loop through a set of code a specified number of times, we can use the range() function.

The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

```
# Note that range(6) is not the values of 0 to 6, but the values 0 to 5.
range(6)


# Using the range() function:
for x in range(6):
  print(x)


# The range() function defaults to 0 as a starting value, however it is possible
to specify the starting value by adding a parameter: range(2, 6), which means
values from 2 to 6 (but not including 6):

for x in range(2, 6):
  print(x)


# The range() function defaults to increment the sequence by 1, however it is
possible to specify the increment value by adding a third parameter: range(2, 30,
3):

for x in range(2, 30, 3):
  print(x)
```

Nested Loops

A nested loop is a loop inside a loop.

The "inner loop" will be executed one time for each iteration of the "outer loop".

```
# Print each adjective for every fruit:
adjs = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for adj in adjs:
  for fruit in fruits:
    print(adj, fruit)
```

The pass Statement

for loops cannot be empty, but if you for some reason have a for loop with no content, put in the pass statement to avoid getting an error.

```
for x in [0, 1, 2]:
  pass
```

While vs For

Pushup analogy:

- For ----> Do pushup 10 times
- While --> Do pushup until I say stop!

Functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

Naming functions varies, but in django you will genarally see snake_case.

```
# In Python a function is defined using the def keyword:
def my_function():
  return "Hello from a function"

# To call a function, use the function name followed by parenthesis:
my_function()
```

Arguments

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses.

You can add as many arguments as you want, just separate them with a comma.

```
# The following example has a function with one argument (name). When the function
is called, we pass along a name, which is used inside the function to print the
full name.
def my_function(name):
    retrun f"Hello {name} how are you?"

my_function("John")


# This function expects 2 arguments, and gets 2 arguments:
def my_function(name, last_name):
    retrun f"Hello {name} {last_name} how are you?"

my_function("John", "Smith")
```

Arbitrary Arguments, *args

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

This way the function will receive a tuple of arguments, and can access the items accordingly:

If the number of arguments is unknown, add a * before the parameter name:

```
def my_function(*kids):
  return "The youngest child is " + kids[0]

my_function("Hans", "Angel", "Lilly")
```

Keyword Arguments

You can also send arguments with the key = value syntax.

```
# This way the order of the arguments does not matter.
def my_function(child3, child2, child1):
  return "The youngest child is " + child3

my_function(child1 = "Hans", child2 = "Angel", child3 = "Lilly")
```

Arbitrary Keyword Arguments, **kwargs

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition.

This way the function will receive a dictionary of arguments, and can access the items accordingly:

```
# If the number of keyword arguments is unknown, add a double ** before the
parameter name:

def my_function(**kids):
    for kid in kids.values():
        print(f"Hello {kid}")

my_function(name = "John", last_name = "Smith")
```

Ordering Arguments in a Function

The correct order for your parameters is:

- Standard arguments
- *args arguments
- **kwargs arguments

```
# correct_function_definition.py
def my_function(a, b, *args, **kwargs):
    pass
```

Default Parameter Value

If we call the function without argument, it uses the default value:

```
def my_function(country = "Norway"):
  return "I am from " + country

my_function("Sweden")
my_function()
```

The pass Statement

Function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the pass statement to avoid getting an error.

```
def myfunction():
  pass
```

Decorators

A decorator is a design pattern in Python that allows a user to add new functionality to an existing object without modifying its structure.

Decorator takes another function as an argument, extending the behavior of that function without explicitly modifying it.

Decorators are usually called before the definition of a function you want to decorate.

Decorators dynamically alter the functionality of a function, method or class without having to make subclasses or change the source code of the decorated class.

Thanks to this our code will be more cleaner, more readable, maintainable (Which is no small thing), and reduce the boilerplate code allowing us to add functionality to multiple classes using a single method.

Basic Template of Decorators

```python
def my_decorator(func):
    def wrapper():
        # Do something before
        result = func()  # or only func()
        # Do something after
        return result  # or erase this line, with only func() above
    return wrapper

@my_decorator
def func():
    pass
```

Here is a simple decorator example:

```python
def make_posh(func):
    def wrapper():
        print('+---------+')
        print('|         |')
        result = func()
        print(result)
        print('|         |')
        print('+=========+')
        return result
    return wrapper

@make_posh
def pfib():
    '''Print out Fibonacci'''
    return ' Fibonacci '
```

Here is another example:

```python
import time

def elapsed_time(func):
    def wrapper():
        t1 = time.time()
        func()
        t2 = time.time()
        print(f'Elapsed time: {(t2 - t1) * 1000} ms')
    return wrapper

@elapsed_time
def big_sum():
    num_list = []
    for num in (range(0, 100)):
        num_list.append(num)
    print(f'Big sum: {sum(num_list)}')

big_sum()
```

```python
from django.contrib.auth.decorators import login_required

@login_required
def my_view(request)
    pass

# Each time that a user try to access to my_view, the code inside login_required
will be executed.

# A simple view – A view function, or view for short, is a Python function that
takes a Web request and returns a Web response.
```

Exception Handling

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

These exceptions can be handled using the try statement:

- The try block lets you test a block of code for errors.
- The except block lets you handle the error.
- The finally block lets you execute code, regardless of the result of the try- and except blocks.

```python
# The try block will generate an exception, because x is not defined:

try:
  print(x)
except:
  print("An exception occurred")
```

```
# Since the try block raises an error, the except block will be executed.
```

Many Exceptions

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error.

```
try:
  print(unknown_var)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")


# You can use the else keyword to define a block of code to be executed if no
errors were raised:
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")

# The finally block, if specified, will be executed regardless if the try block
raises an error or not.
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

Raise an exception

As a Python developer you can choose to throw an exception if a condition occurs.

This type of error occurs whenever syntactically correct Python code results in an error.

To throw (or raise) an exception, use the raise keyword.

```
# Raise an error and stop the program if x is lower than 0:

x = -1

if x < 0:
  raise Exception("Sorry, no numbers below zero")
```

```
# The raise keyword is used to raise an exception.

# You can define what kind of error to raise, and the text to print to the user.

#Raise a TypeError if x is not an integer:
x = "hello"

if not type(x) is int:
  raise TypeError("Only integers are allowed")
```

pip

pip is the standard package manager for Python. It allows you to install and manage additional packages that are not part of the Python standard library.

pip is already installed if you are using Python 2 >=2.7.9 or Python 3 >=3.4 downloaded from python.org or if you are working in a Virtual Environment created by virtualenv or venv. Just make sure to upgrade pip.

https://pip.pypa.io/en/stable/installing/

```
# Check if pip is installed or not.
pip --version

# See the commands available with pip
pip help

# Upgrade pip
python -m pip install --upgrade pip

# Install any package with pip
pip install <package name>
pip install python-decouple
# https://pypi.org/project/python-decouple/


# See if it is installed or not:
pip list  # List installed packages. or:
pip freeze  # Output installed packages in requirements format.


# Send packages needed in the project to a specific file
pip freeze > requirements.txt


# Install all needed packages with pip
pip install -r requirements.txt
```

Built-in Functions in Python

https://docs.python.org/3/library/functions.html

Modules

In Python, Modules are simply files with the ".py" extension containing Python code that can be imported inside another Python Program.

Examples: `os`, `random`,`time`

Python Module Index

https://docs.python.org/3/tutorial/modules.html

☺ **Happy Coding!** ✍

Clarusway