PYTHON PROGRAMMING

INTRODUCTION

- Iulia Chiriac
- Contacts

- http://luxoft-training.ru
- http://luxoft-training.com

TRAINING ROADMAP: STRUCTURE

- 40 hours
- 10-15 mins breaks every 1.5 – 2 hours
- Lunch break
- In-class individual practice
- In-class group workshops

SECTION 1:
INTRODUCTION

WHY PYTHON?

- Very popular, top 10 programming languages

  over 1 million questions asked on stackoverflow.com;

  #1 by CodeEval, #3 by IEEE Spectrum

- Concise, clear, highly readable
- Dynamic, high level, interactive language
- Easy to embed as scripting
- Portable: Windows, Linux, Mac, Android, browser

EXECUTION MODEL VARIATION

- CPython – standard implementation
- PyPy – alternative interpreter and JIT compiler (EU funding)
- Jython – written in Java
- IronPython – written in C# and integrated with .NET
- Cython – Python + C = ♥

RUNNING PYTHON SCRIPTS: INTERACTIVE CONSOLE

- IDLE (local, usually installed by default)
- iPython (local)
- DreamPie (local)
- repl.it/languages/Python (online, inside a sandbox)
- live.sympy.org (online, inside a sandbox)
- shell.appspot.com (online, inside a sandbox)
- pythonanywhere.com (persistent sessions!)

EXECUTING PYTHON CODE

- Run a Python script as a file
  - Under Windows with CMD prompt
  - Under Linux/Unix/MAC
- Run a Python script from the interactive mode

```
Windows:
C:\Python27\python.exe C:\Scripts\script1.py

Linux/Unix:
chmod +x scripts/script1.py
python  scripts/script1.py



Windows:
C:\Users>python.exe
Python 2.7.10 (default, May 23 2015, 09:40:32) [MSC v.1500 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more information.
>>>

Linux:
user@hostname:~ python
Python 2.7.2+ (default, Oct  4 2011, 20:03:08) [GCC 4.6.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

BASIC SYNTAX

- Comments

```
# this is a comment
```

- Literal constants
- Variables
- Logical vs Physical line
- Indentation
- built-ins functions:

```
help() dir() type() print()
```

OPERATORS AND EXPRESSIONS

- expression: operators and operands
- operators:
  - arithmetic: +, - , *, /, %, **, //
  - comparison: ==, !=, <>, <, >, <=, >=
  - assignment: =, +=, -=, *=, /=, etc
  - bitwise: &, |, ^, ~, <<, >>
  - logical: and, or, not
  - membership test: in, not in
  - identity test: is, not is

BASIC TYPES: NUMERIC

- Integers – int()
  - Unlimited precision
- Float – float()
  - Precision : sys.float_info

BASIC TYPES: BOOL

- `bool()`
- subtype of int
- constants: True and False

```
>>> bool(1)
True
>>> bool(0)
False
```

BOOLEAN EXPRESSIONS

- Python values evaluated as False:
  - Numeric null values: 0, 0.0
  - None object
  - Empty string: '', "", str()
  - Empty sequences: (), [], {}
- ‘Short circuiting’ for evaluating logical expressions

NUMERIC TYPES: OPERATIONS

- 0b... are numbers in base 2 ; 0o… are octal ; 0x... are numbers in hex
- Built-in operations: sum, pow, round, min, max
- Math operations: trunc, floor, ceil, exp, log, sqrt, factorial, fsum, sin, cos
- Math constants: math.pi, math.e
- Conversion: int(x, base), float(x), complex(real, imag)
- Convert into base: bin(nr), oct(nr), hex(nr)
- Not all Python objects can be converted into numbers

BASIC TYPES: STRINGS

- List of characters, delimited by quotation marks: "" or ''
- Strings are immutable
- Multiline strings : """ """ or ''' '''
- Special characters should be escaped: ' , " , \
- \n – newline ; \t - tab

STRINGS: RAW STRINGS

- A string that ignores all escape characters and prints any backslash.
- Syntax:

```
s = r'Raw string - \'will be printed'
```

STRINGS: BYTESTRING VS STRING IN PYTHON3

- String
  - Sequence of characters, human readable
  - To write it on the disk it has to be converted to a bytestring
- Bytestring
  - Sequence of bytes, non-human readable
  - Similar to ‘unicode’ from Python2
- Methods: encode() , decode()

STRING TYPE: STRING MANIPULATION

- Accessing characters: `[index]`
- Slicing: `[index : index]` (count starts from 0, -1 last element)
- Concatenate: `string1 + string2` (use += for appending)
- Multiply: `string * number`
- Length: `len(string)`
- `in` and `not in` operators

STRING TYPE: STRING MANIPULATION METHODS

- Finding: find(word[,start[,end]]), startswith(prefix[,start[,end]]), endswith(sufix[,start[,end]])
- Removing space: strip([chars]), lstrip([chars]), rstrip([chars])
- Joining: join(iterable)
- Splitting: split([separator[,maxsplit]])
- Changing letters: replace(old,new[,count]), upper(), lower(), title(), capitalize()
- Verifying the string nature: isupper(), islower(), isalpha(), isalnum(), isspace(), isdigit()
- Counting: count()
- Justifying text: rjust(width[,fillchar]), ljust(width[,fillchar]), center(width[,fillchar])

STRING TYPE: STRING MANIPULATIONS

```
#Single/double quotes example:
str1 = 'String 1‘
str2 = "String's reloaded“
str3 = "\"Yes\", he said."

#'Raw' string example:
rstr = r"C:\Program Files"; print(rstr)
nrstr = "C:\Program Files"; print(nrstr)

#Concatenated strings:
word = 'Hello ' + ‘world';
print(word)
print(word*2) # Prints string two times
```

```
#Slice operator:
str1 = 'Hello World!'
print(str1[0]) # Prints first character of the string
print(str1[2:5]) # Prints characters starting from 3rd to 5th
print(str1[2:]) # Prints string starting from 3rd character


#String methods
str2 = " Hello World! "
print(str2.strip()) # Remove leading and trailing spaces
print(str2.upper()) # Letters are converted to upper case
print(str2.split()) # Split string by spaces
print(str2.replace("World","Europe")) # Replace characters
print(str2.count("l")) # Occurrences of substring 'l' in string
```

BASIC TYPES EXERCISES

1. Given an integer number, print its last digit.
2. Given a three-digit number. Find the sum of its digits.
3. Given the integer N - the number of minutes that is passed since midnight - how many hours and minutes are displayed on the 24h digital clock? The program should print two numbers: the number of hours (between 0 and 23) and the number of minutes (between 0 and 59).

   For example, if N = 150, then 150 minutes have passed since midnight - i.e. now is 2:30 am. So the program should print 2 30.

BASIC TYPES EXERCISES

4. Given the string s = "bandana":

- check if string "and" is contained in s
- find the index of the following strings: "n", "q"
- how many times does the string "an" appear in s?
- check if s is alphanumeric
- transform s to all uppercase
- check other string methods and try them out

BASIC TYPES EXERCISES

5. Given a string, print the following:

- In the first line, print the third character of this string.
- In the second line, print the second to last character of this string.
- In the third line, print the first five characters of this string.
- In the fourth line, print all but the last two characters of this string.
- In the fifth line, print all the characters of this string with even indices (remember indexing starts at 0, so the characters are displayed starting with the first).
- In the sixth line, print all the characters of this string with odd indices (i.e. starting with the second character in the string).
- In the seventh line, print all the characters of the string in reverse order.
- In the eighth line, print every second character of the string in reverse order, starting from the last one.

CONTROL FLOW

- if/elif/else
- while/else
- for/else

BASIC CONTROL STRUCTURES: IF/ELIF/ELSE

- Syntax

```
if (condition1):
    statement1
elif (condition2):
    statement2
elif (condition3):
    statement3
else:
    statement4
```

- else and elif are optional
- Single line if : *if (condition1):* statement 1

BASIC CONTROL STRUCTURES: WHILE/ELSE

- Syntax

```
while (conditions):
    statement1
    statement2
else:
    statement3
```

- Else is optional and will be executed when condition becomes false
- Single line while: *while (conditions):* statement 1

BASIC CONTROL STRUCTURES: FOR LOOP

- Syntax

```
for iter_variable in sequence:
    statement1
    statement2
else:
    statement3
```

- Else is optional and will be executed when iteration is completed

BASIC CONTROL STRUCTURES: LOOP CONTROL STATEMENTS

- break

  The loop terminates and the execution is transferred to the statement that follows the loop.

- continue

  The remaining statements will be skipped and the condition will be retested prior to reiterating.

- pass

  when a statement is required syntactically but you do not want any command or code to execute.

FUNCTION SYNTAX

- Syntax:

```
def functionname( parameters ):
   "function_docstring"
   function_suite
   return [expression]
```

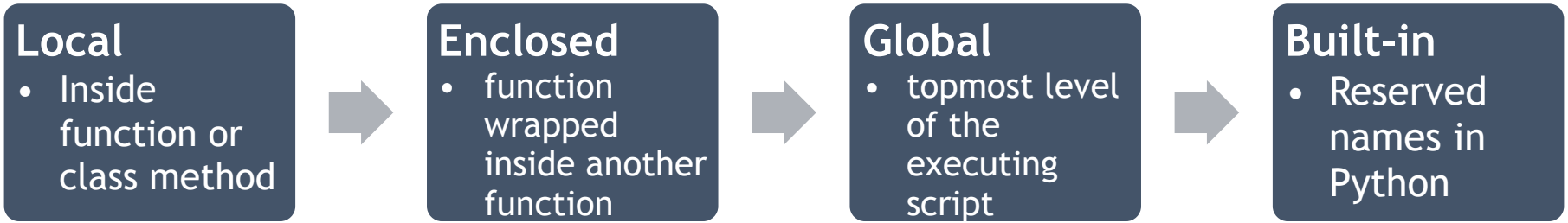
- return is optional

FUNCTION SYNTAX: ARGUMENTS

- Required arguments
  - Arguments passed to a function in correct positional order
  - The number of arguments in the function call should match exactly with the function definition
- Keyword arguments
  - They are related to the function calls
  - In a function call, the caller identifies the arguments by the parameter name
- Default arguments
  - An argument that assumes a default value if a value is not provided in the function call
- Variable-length arguments
  - They are used when you need to process a function for more arguments than you specified while defining the function.
  - These are not named in the function definition, unlike required and default arguments.

FUNCTION SYNTAX: SCOPES – THE LEGB RULE

- Scope – on which level Python searches for a variable name associated to an object



MODULE/PACKAGES

- Syntax for importing a module

  import module1[, module2[,... moduleN]

  from module_name import name1[, name2[, ... nameN]]

- The module is searched in the following:

  Current directory

  Each directory in shell variable PYTHONPATH

  Python default path (OS dependent – Unix /usr/local/lib/python)

- sys.path

MODULE/PACKAGES: CREATING SIMPLE MODULES

- Any file with the extension '.py' can be a Python module
- It can contain executable statements and function definitions
- A module is imported only once per session.
- To visualize changes in your modules, the interpreter must be restarted or called *reload()* function

MODULE/PACKAGES: CREATING PACKAGES

- sys.path – package subdirectory
- File __init__.py placed in a directory, makes Python treat it as a package
- Individual modules from the package can be imported using dots.

```
import package1.submodule
```

SIMPLE SCRIPTS

1. Write a function that takes a number as a parameter and **prints** its square.
2. Write another function that takes a number as a parameter and **returns** the square. Are the results of the two functions different?
3. Write a Python program which iterates the integers from 1 to 50. For multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz". For all the other numbers print the number.
4. Write a function that prints the odd numbers from a given interval. The default interval is 0-100.
5. Write a function for checking the speed of drivers. This function should have one parameter: speed.
   - If speed is less than 50, it should print “OK”.
   - Otherwise, for every 5 km above the speed limit (50), it should give the driver one demerit point and print the total number of demerit points. For example, if the speed is 60, it should print: “Points: 2”.
   - If the driver gets more than 12 points, the function should print: “License suspended”

SIMPLE SCRIPTS

6. Write a function that builds html tags. Apply html escaping for html special chars.

   The function will receive 2 parameters – tag type and tag content. It will return the generated html as text.

   e.g.: `f('b', 'Ham&Eggs')` returns `"<b>Ham&Eggs</b>"`

   HTML char escaping:

   - < becomes <
   - > becomes >
   - “ becomes "
   - & becomes &

7. Create a package and use it in Python shell

SIMPLE SCRIPTS

8. Describe the scope of the variables a, b, c and d in this example:

```
def my_function(a):
    b = a - 2
    return b
c = 3
d = None
if c > 2:
    d = my_function(5)
print(d)
```

- What is the lifetime of these variables? When will they be created and destroyed?
- Can you guess what would happen if we were to assign c a value of 1 instead?
- Why would this be a problem? Can you think of a way to avoid it?

SECTION 2:
ITERABLES

SEQUENCE TYPES: LIST, TUPLE

- List:
  - Members might have different types of data
  - Creating:

```
l = list()  # empty list
l = []  # empty list
l = list(iterable)  # new list initialized from iterable's items
l = [element1, element2]  # list with two elements
```

  - Mutable
- Tuple
  - Members might have different types of data
  - Creating:

```
tuple(), (), (element1, element2), tuple(iterable)
```

  - Immutable

SEQUENCE TYPES: LIST MANIPULATION

- Accessing elements: `list[index]`
- Modifying elements: `list[index] = new_element`
- Slicing: `list[start:stop:step]` (count starts from 0, -1 last element)
- Slice assignment: `list[start:stop] = new_list`
- Concatenate: `list1 + list2` (use += for appending)
- Multiply: `list * number`
- Length: `len(list)`
- `in` and `not in` operators

SEQUENCE TYPES: SLICE OPERATOR

```
l1 = ['abcd', 786, 2.23, 'john', 70.2]

print l1[0] # Prints first element of the list -> abcd
print l1[1:3] # Prints elements starting from 2nd till 3rd -> [786, 2.23]
print l1[2:] # Prints elements starting from 3rd element -> [2.23, 'john', 70.2]
print l1[::-1] # Reverses the list

#Example of slice assignment:
l1[0:2] = [1, 2] # Replace some items
print(l1) # Prints [1, 2, 2.23, 'john', 70.2]

l1[0:2] = [] # Remove some items
print(l1) # Prints [2.23, 'john', 70.2]

l1[1:1] = ['insert1', 'insert2'] # Insert some items
print(l1) # Prints [2.23, 'insert1', 'insert2', 'john', 70.2]

l1[:0] = l1 # Insert a copy of itself at the beginning
print(l1) # Prints [2.23, 'insert1', 'insert2', 'john', 70.2, 2.23, 'insert1', 'insert2', 'john', 70.2]

l1[:] = [] # Clear the list
print(l1) # Prints []
```

LIST/SEQUENCE FUNCTIONS: MANIPULATION ELEMENTS

- list.append(*x*)

  Add an item to the end of the list; equivalent to a[len(a):] = [x]

- list.extend(*L*)

  Extend the list by appending all the items in the given list; equivalent to a[len(a):] = L

- list.insert(*i*, *x*)

  Insert an item at a given position. (a.insert(0, x) inserts at the front of the list, and a.insert(len(a), x) is equivalent to a.append(x))

- list.remove(*x*)

  Remove the first item from the list whose value is *x*. It is an error if there is no such item.

- list.pop([*i*])

  Remove the item at the given position in the list, and return it.

  If no index is specified, a.pop() removes and returns the last item in the list.

LIST/SEQUENCE FUNCTIONS: SORTING AND COUNTING

- list.sort()

  Sort the items of the list, in place.

- list.reverse()

  Reverse the elements of the list, in place

- list.index(*x*)

  Return the index in the list of the first item whose value is *x*. It is an error if there is no such item.

- list.count(*x*)

  Return the number of times *x* appears in the list.

LISTS EXERCISES

1. Write a Python program to convert a list of characters into a string.
2. Write a function `filter_long_words()` that receives as parameters a list of words and an integer `n` and returns the list of words longer than `n`.
3. Write a function that receives two lists as parameters and returns True if they have at least one common element, False otherwise.
4. Given a list of numbers with all of its elements sorted in ascending order, determine and print the quantity of distinct elements in it.

SEQUENCE TYPES: SET, FROZENSET

- Unordered collection of hashable objects
- Used for testing memberships, removing duplicates, math operations
- Set type is mutable
- Frozenset type is immutable

SEQUENCE TYPES: SET, FROZENSET

- len(s) : number of elements
- x in s : verifies membership of x in s
- x not in s: verifies that x in not a member of s
- issubset() or s1 <= s2 or s1 < s2
- issuperset() or s1 > other or s1 >= other
- isdisjoint()
- union() or s1 | s2
- intersection() or s1 & s2
- difference() or s1 – s2
- copy()

SEQUENCE TYPES: SET, FROZENSET

- add()
- remove()
- discard()
- pop()
- clear()
- update() or s1 |= s2
- intersection_update() or s1 &= s2
- difference_update() or s1 -= s2

SET EXERCISES

1. Given two lists of numbers. Count how many distinct numbers occur in both the first list and the second.

   `f([1, 1, 2, 3], [2, 2, 3, 4]) -> 2` (2 and 3 occur in both)

2. Write a function that receives a text (a string) and returns the number of distinct words. A word=a sequence of characters that is not whitespace (space, newline, tab).

   `f('hello hello is there anybody in there') -> 5`

MAPPING TYPES: DICTIONARY

- A set of *key : value* pairs
- Dict order is guaranteed to be insertion order (since Python 3.7)
- Creating:

  Empty dict: d = dict(); d = {}

  Key by key: d = dict(); d[key] = value

  Compact : d = { key:value, key1:value1}

  Using keyword arguments: d = dict(key=value, key1=value1)

- Deleting:

  A key: del d[key]

THE DICTIONARY TYPE: RESTRICTIONS

- Key has to be hashable
  - Number
  - String
  - Tuple that contains only numbers, strings or tuples
- Key is unique within the dictionary
  - If storing an existing key, the old value will be overwritten

THE DICTIONARY TYPE: GETTING KEYS/VALUES/ITEMS

- key in d: check whether *d* has a key *key*
- dict.get(key, default=None): For *key* key, returns value or default if key not in dictionary
- dict.items() :Returns a list/view of *dict*'s (key, value) tuple pairs
- dict.keys() : Returns list/view of dictionary dict's keys
- dict.values(): Returns list/view of dictionary *dict2*'s values

THE DICTIONARY TYPE: CLEAR/COPY/UPDATE

- dict.clear(): Removes all elements of dictionary *dict*
- dict.copy(): Returns a shallow copy of dictionary *dict*
- dict.pop(key[, default]): If *key* is in the dictionary, remove it and return its value, else return *default*
- dict.update(dict2): Adds dictionary *dict2*'s key-values pairs to *dict*

THE DICTIONARY TYPE: DICTIONARY ITERATORS

- Loop the keys of a dictionary:

```
for key in dict:
for key  in dict.keys():
for key in dict.iterkeys():              * only in Python 2
```

- Loop keys and values of a dictionary:

```
for key, value in dict.items():
for key, value in dict.iteritems():      * only in Python 2
```

DICT EXERCISES

1. Given the following dictionary:

```
d = {'times': 100,
     'name': 'George',
     'hobbies': ['fishing', 'hiking']}
```

- add key *'friends'* to *d* with *['Andrei', 'Mihai', 'Alina']* as value
- sort value for key '*friends*'
- remove *'hiking'* from hobbies list
- remove *'times'* key from *d*

DICT EXERCISES

2. Given a list of strings build a dictionary that has each unique string as a key and the number of appearances as a value.

```
f(['hello', 'hello', 'is', 'there', 'anybody', 'in', 'there']) ->
{'hello': 2, 'is': 1, 'there': 2, 'anybody': 1, 'in': 1}
```

3. Write a python program to map two lists into a dictionary. Ignore extra elements if one of the lists is larger.

```
f([1, 2, 3, 4], ['Ana', 'Vali', 'Geo']) ->
{1: 'Ana', 2: 'Vali', 3: 'Geo'}
```

FUNCTIONS: VARIABLE LENGTH ARGUMENTS

```
def func(*args, **kwargs):
        for i in args:
                print(i)
        for k, v in kwargs.items():
                print(k, v)
```

- *args: variable-length positional arguments
- **kwargs: variable-length keyword arguments
- can be used when calling a function as well

VARARGS EXERCISES

1. Write a function `filter_long_words()` that receives as parameters a variable number of words (as args) and an integer `min_length` and returns the list of words longer than `min_length`.

   ```
   f('hello', 'how', 'is', 'Mary', min_length=3) ->
   ['hello', 'Mary']
   ```

STRINGS: PRINTING FORMATTED OUTPUT

- String objects have a specific method used for formatting – `format()`
- Format method uses formatters atoms to decide how to display data types along static strings
- Example:

```
"The result is: {formatter}".format(result_value)
```

STRINGS: PRINTING FORMATTED OUTPUT: FORMATTER

- Formatter syntax:

{[value]![conversion_method]:[fill][align][sign][width].[precision][format_type]}

- [value]: <empty>, Named, Indexed
- [conversion_method]: s – str(); r – repr()
- [align]: “<“ , “>“ , “=” , “^”
- [sign]: + , - ,‘ ’ [space]
- [format_type]:

  for strings(‘s’,None);

  for integers(‘b’,’c’,’d’,’o’,’x’,’X’,’n’,None);

  for float(‘e’,’E’,’f’,’F’,’g’,’G’,’n’,’%’)

STRINGS: PRINTING FORMATTED OUTPUT: FORMATTER

```
print 'Formatting with positional and named arguments'
print '{} {}'.format('one', 'two')
print '{1} {0}'.format('one', 'two')
print '{} {}'.format(1, 2)
print '{first_name} {last_name}'.format(last_name='Picard', first_name='Jean-Luc‘)

print 'Add some padding'
print '{:_<10}'.format('test')
print '{:_>10}'.format('test')
print '{:_^10}'.format('test')

print 'Add specific formatting to named value'
print '{result!s:_^20}'.format(result=(1, 2))
print '{result!r:_^20}'.format(result=(1, 2))

print '{result:_^ 20.2f}'.format(result=3.141)
print '{result:_^ 20.2f}'.format(result=-3.141)
```

FORMAT EXERCISE

1. Given following data structure:

```
[
   {'first_name': 'John', 'last_name': 'Cornwell', 'net_worth': 2632.345},
   {'first_name': 'Emily', 'last_name': 'Alton', 'net_worth': -4578.234},
   {'first_name': 'James', 'last_name': 'Bond', 'net_worth': 1000.07},
]
```

Generate an output formatted like:

```
-------------------------------------
| Cornwell          J. |   +2632.34 |
| Alton             E. |   -4578.23 |
| Bond              J. |   +1000.07 |
-------------------------------------
```

Last names are left aligned with padding to 15 chars

From first name display only first letter right aligned with padding, total width 2 chars, followed by '.'

Net worth column width is 10 chars, we display only first two decimals, and also sign for both positive and negative numbers

Data is wrapped up between '|' and '-' ASCII chars to print similar to a table

SECTION 3:
ADDITIONAL PYTHON FEATURES

TRY/EXCEPT/FINALLY

```
try:
      Statements to be executed
except(Exception1[, Exception2[,...ExceptionN]]]):
      If there is any exception from the given exception list,
      then this block is executed.
except  Exception1 as e:
       If Exception1 is raised, the exception instance is bound to ‘e’
variable.
except:
      If there is any exception, then this block is executed
      Use it with caution, because it might mask programming errors!
else:
      If there is no exception then this block is executed
finally:
     Always executed before leaving try, weather an exception has occurred or
not
```

- except, else or finally can be optional
- A try statement might have more that one except clause

EXERCISE

1. Write a program to read two numbers: `x` and `y` from standard input and print the result of `x/y`. If the user inputs invalid data, display an error message and exit gracefully.

RAISE, ASSERT

- raise: force a specified exception to occur

```
raise ValueError
```

- assert: insert debugging assertions into a program

```
assert condition
```

- raises AssertionError if condition is false

USING FUNCTIONS AS INPUT TO OTHER FUNCTIONS

- Everything in Python is an object
- Functions are objects
- Inspect a function's attributes: `dir(func)`
- Functions can be passed as arguments to other functions

```
def two_number_sum(a, b):
    return a + b

def two_number_diff(a, b):
    return a - b

def get_result(a, b, operation):
    return operation(a, b)
```

```
get_result(10, 2, two_number_sum)

get_result(10, 2, two_number_diff)
```

FUNCTIONS: LAMBDA

- anonymous functions

```
lambda x: x + 42
```

- throw-away functions, one purpose only
- can be used together with built-in functions like:

```
min(), max()
map(), filter()
sorted()
```

ITERABLES: BASIC AGGREGATIONS

- Min: returns smallest item
  - min(iterable[, key])
  - min(arg1, arg2,*args[,key])
- Max: returns larges item
  - max(iterable[, key])
  - max(arg1, arg2, *args[, key])
- Sum: returns the total of the items in an iterable, starting with start
  - sum(iterable[, start])
  - start is 0 by default, can be only number
  - Usually used for numbers

ITERABLES: FILTER&MAP

- Filter

  filter(function, iterable)

  iterable items for which *function* returns true

- Map

  map(function, iterable,…)

  The results of *function* applied to every *iterable* item.

  Additional iterables -> function arguments

ITERABLES: ZIP, SORTED, ENUMERATE

- Zip

  zip(iter1, iter2, …)

  Return a zip object of tuples where the i-th element comes from the i-th iterable argument

- Sorted

  sorted(iterable, key=None, reverse=False)

  Return a new sorted list from the items in iterable.

- Enumerate

  enumerate(iterable, start=0)

  Iterate over (index, value) pairs

AGGREGATIONS EXERCISES

1. Given a list of tuples `(product, price_eur)`, build the list of `(product, price_ron)`, knowing that the exchange rate is 4.75.
2. Write a function `filter_short_words(word_list, n)` that returns the words in `word_list` shorter than `n` as a list.
3. Write a function that receives any number of strings and returns the list of unique strings ordered by number of appearances (most frequent → least frequent).

```
f('hello', 'there', 'hello', 'hi', 'hi', 'hello') ->
['hello', 'hi', 'there']
```

COMPREHENSIONS

- List comprehensions

```
l1 = [ n*2 for n in range(10) ]
l2 = [ n for n in range(10) if n%2==0 ]
```

- Dictionary comprehensions

```
d1 = {n: n*2 for n in range(10)}
```

- Generator comprehensions

```
g1 = ( n*2 for n in range(10) )
```

- Embedded comprehensions

```
m1 = [[1, 2, 3], [1, 2, 3], [1, 2, 3]]
l1 = [ [ row[i] for row in m] for i in range(3)]
```

COMPREHENSIONS EXERCISES

1. Create a dict `{"a": 97, "b": 98, ... }` using comprehension. Keys range from "a" to "e".
2. Using the dictionary generated above, create another one where you swap keys and values.
3. Filter the above dictionary to contain only even keys.
4. Can you obtain dictionary from ex. 3 from the given string ("abcde") in a single dict comprehension?

```
import string
string.ascii_lowercase
Out[51]: 'abcdefghijklmnopqrstuvwxyz'
```

ITERABLES: ITERABLE OBJECTS, ITERATOR

- Python loves the idea of iterating over objects, generally called iterators;
- Create an iterator: `iter(array_object)`
- Every iterator has a method called 'next':

  `next` method returns elements in iterator one by one on every call.

  When the cycle is done, the `StopIteration` exception is raised ;

- Any object that can be iterated, can be used in a `for` or `while` cycle.

CREATING GENERATOR FUNCTIONS

- A generator is a function that returns an iterator.
- To create a generator, use `yield` keyword anywhere in a function
- Use `yield` keyword to generate a value. It saves the actual state of the generator function
- Get the next value using `next()`

```
def stepper(n):
    ret = 0
    while ret < n:
        yield ret
        ret += 1
```

GENERATORS EXERCISES

1. Create a generator function that receives a parameter `max_nr` and returns a random number between 1 and `max_nr`, indefinitely.
2. Write a generator function that yields unique elements from an iterable received as parameter.

SECTION 4:
ADVANCED TOPICS

DECORATOR: SIMPLE DECORATORS

- Dynamically modify or introduce code in methods and functions
- Syntax:

  @decorator_name
- Functions can be:

  assigned to a variable

  defined inside other function

  passed as parameter

  returned by other functions

DECORATOR: SIMPLE DECORATORS – FUNCTION

```
def duplicate(func):
   def func_wrapper(*arg,**kwargs):
      return 2 * func(*arg,**kwargs)
   return func_wrapper

@duplicate
def double_val(val):
   print("Double value " + str(val))
   return val

print(double_val(10))
```

DECORATOR: DECORATORS WITH ARGUMENTS

```
def multiply(nr):
    def multiply_decorator(func):
        def func_wrapper(*arg,**kwargs):
            return nr * func(*arg,**kwargs)
        return func_wrapper
    return multiply_decorator

@multiply(2)
def double_val(val):
    print("Double value " + str(val))
    return val

@multiply(3)
def triple_val(val):
    print("Triple value " + str(val))
    return val

print(double_val(10))
print(triple_val(10))
print(double_val(10))
```

DECORATOR: BUILT-IN FUNCTION/CLASS DECORATORS

- @property
- @staticmethod
- @classmethod
- @abstractproperty
- functools module

DECORATOR EXERCISES

1. Write a decorator that computes (and displays) execution time for a function.

   Hint: `time.time()` function returns current time in seconds

SECTION 5:
OBJECT ORIENTED PROGRAMMING

OOP BASICS: CLASSES AND INSTANCES

- Class (type):
  - template for creating objects
  - also referred to as type
  - consists of methods and attributes (generic: members) that define the type's behaviour
- Object (instance):
  - object created according to a template (class)

OOP BASICS: DEFINING A CLASS

- keyword `class`
- docstring
- class body

```
class MyClass:
    '''Class description'''
    pass
```

- calling the class will create a new instance

```
instance = MyClass()
```

CLASS MEMBERS: ATTRIBUTES

- Because of Python's dynamic nature, they can be set at runtime
- Class attributes: shared by all instances
- Instance attributes: owned by specific instances of a class

```
class MyClass:
    class_attr = 0

    def __init__(self, method_param):
        self.instance_attr = method_param
```

CLASS MEMBERS: METHODS

- Functions defined inside the class
- They receive the current instance as first parameter (self)
- Must be called in the context of an instance of a class

```
class MyClass:
   def method(self):
       pass

my_class_instance = MyClass()
my_class_instance.method()
```

CLASS MEMBERS: STATIC AND CLASS METHODS

- `@staticmethod` → neither the object, nor class is passed as the first argument
  → normal functions, called from an instance or from the class

- `@classmethod` → the class is the first argument of the method
  → called from an instance or from the class

MAGIC METHODS

- Constructor: `__new__()`
- Initializer: `__init__()`
- Destructor: `__del__()`

MEMBER ATTRIBUTES – ACCESS CONTROL SOLUTIONS

- `member` → public

  Prefixing member names with _ (underscore):

- `__member` → private
- `_member` → protected

- It's a convention based on the developer's responsibility;
- Name mangling for accessing outside the class: obj._classname__privateattr
- Useful in encapsulation.

MEMBER ATTRIBUTES – GETTER/SETTER METHODS

- Getter: method that gets an attribute value
- Setter: method that sets an attribute value
- Deleter: method that deletes an attribute value
- Also, used by property class or as decorators

  @attr.setter, @attr.getter, @attr.deleter

MEMBER ATTRIBUTES – THE PROPERTY CLASS

- Returns a property attribute
- Syntax:

  class property([fget[, fset[, fdel[, doc]]]])

  - fget: getting an attribute
  - fset: setting an attribute
  - fdel: deleting an attribute
  - doc: docstring for the attribute

  @property

OOP EXERCISES

1. Create a BankAccount class that receives two parameters on initialisation:
   - bank name (str)
   - amount of money (int)
2. Create two methods in this class, one to withdraw money and another one to deposit money into the account. The withdraw method will not allow withdrawing more money than available and it will raise an exception when you attempt to do that.
3. Create a class Employee with three instance attributes:
   - person name (str)
   - bank account (BankAccount)
   - salary (default 0) (int)

Salary should be private. Create a property to set salary; the getter should return None.

Create a method receive_salary that will deposit in the employee's bank account an amount equal to its salary.

PYTHON CLASS TEMPLATE: INHERITANCE

- Syntax:

```
class SubClassName(ParentClass1[, ParentClass2, …]):
    pass
```

- Subclass inherits all parent class members
- Same name method in subclass overwrites parent class method
- Calling parent class methods:

```
super().method_name(*args, **kwargs)
```

- Multiple class inheritance: C3 Method resolution order

PYTHON CLASS TEMPLATE: INHERITANCE

```
class ItemList(object):
  def __init__(self, name):
    self.name = name
    self.description = None
    self.items = list()

  def add_item(self, item):
    self.items.append(item)

  def set_items(self, items):
    self.items = items

  def get_items(self):
    return self.items
```

```
class TodoList(ItemList):
  def __init__(self, name, asignee):
    # Call Super constructor(name) -- Python2.x
    # super(TodoList, self).__init__(name)
    # Call Super constructor(name) -- Python3.x
    super().__init__(name)
    self.asignee = asignee
  def __str__(self):
      return "Override 'str' in TodoList class"

it1 = ItemList("Item1")
it1.add_item("t1")
print(it1.get_items())
td1 = TodoList("ToDo1","User1")
print(td1.get_items())
td1.add_item('todo1') ; print(td1.get_items())
print(str(td1)); print(str(it1))
```

OOP EXERCISES

1. Create a SpecialBankAccount class inherits BankAccount and receives one extra argument at initialisation which allows for the balance to go below zero (but not under `-overdraft`):

   ```
   overdraft (int)
   ```

2. Override parent `withdraw` method so that the new rule is implemented.
3. Place the two bank account classes in a Python module and the employee class in another Python module. Create a third module that uses the first two modules.

SPECIAL METHODS AND ATTRIBUTES

- A class can implement certain operations that are invoked by special syntax
- Used in: operator overloading
- Attempts to execute an operation raise an exception when no appropriate method is defined
- User-defined types can emulate built-in types

OPERATORS – RELATED FUNCTIONS

- comparison:__eq__, __ne__, __lt__, __gt__, __le__, __ge__
- unary operators: __pos__, __neg__, __abs__, __invert__, __round__, __floor__, __ceil__, __trunc__
- arithmetic operators: __add__, __sub__, __mul__, __floordiv__, __div__, __truediv__, __mod__, __divmod__, __pow__, __lshift__, __rshift__, __and__, __or__, __xor__

STRING REPRESENTATION OF OBJECTS

- `__str__` vs `__repr__`
  - `__str__` is to be human readable (more for clients)
  - `__repr__` is to be unambiguous (more for developers)
  - `__repr__` is backup for `__str__`
- `__call__`
  - Allows the class instance to be called as a function

MEMBER ATTRIBUTES – THE DESCRIPTOR MODEL

- An object that defines any of `__get__()`, `__set__()` or `__delete__()` methods
- Data descriptors: `__get__` and `__set__` definitions
- Non-data descriptors: only `__get__` definition
- Invoked by `__getattribute__()` method

MEMBER ATTRIBUTES – THE DESCRIPTOR MODEL

```
class RevealAccess(object):
    """A data descriptor that sets and returns values normally and prints a message logging their access. """
    def __init__(self, initval=None, name='var'):
        self.val = initval
        self.name = name
    def __get__(self, obj, objtype):
        print 'Retrieving', self.name
        return self.val
    def __set__(self, obj, val):
        print 'Updating', self.name
        self.val = val
class MyClass(object):
    x = RevealAccess(10, 'var "x"')
    y = 5

m = MyClass()
print(m.x) # Prints: # Retrieving var "x" # 10
m.x = 20 # Prints: # Updating var "x"
print(m.x) # Prints: # Retrieving var "x" # 20
print(m.y) # Prints: 5
```

SLOTS

- `__slots__` class variable
- Can be a string, iterable, or sequence of strings with variable names used by instances
- Used for saving space
- Reserves space for the variables
- Dictionary for class attribute storage is not created automatically
- The action is limited to the class where it is defined

PYTHON CLASS TEMPLATE: ABSTRACT TEMPLATE CLASSES

- `abc` module
- `ABCMeta` metaclass
- Class needs to implement all abstract methods to become concrete class
- Register a class with an abstract base class

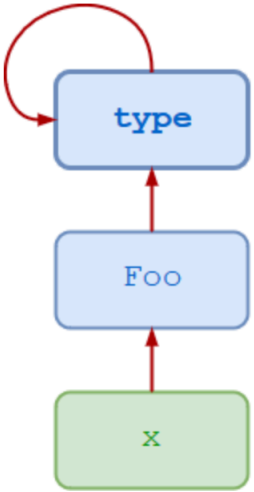
PYTHON CLASS TEMPLATE: ABSTRACT TEMPLATE CLASSES

```
from abc import ABC, abstractmethod

class Person(ABC):
    @abstractmethod
    def get_name(self):
        pass


class Employee(Person):
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def get_name(self):
        print("Name: %s" % self.name)

e1 = Employee("Pers1", 1000)
e1.get_name()
print(isinstance(e1, Person)); print(type(e1)); print(type(Person)); print(type(Employee))
```

PYTHON CLASS TEMPLATE: METACLASS

- The class of a class
- Default metaclass is `type`

- `Metaclass -> __call__`
- `Class -> __new__ __init__`



OOP EXERCISES

- Create abstract class `PublicPlace`. It will have abstract method `get_address()`
- Create class `Restaurant`, that inherits `PublicPlace` class.
  - Instance attributes are: `category(str), rating(int), address(str).`
  - Init will receive category, rating and address
  - Class attribute: `count`
  - Cannot create any `Restaurant` object with rating below 1 or over 5 (create descriptor/property to validate)

- Print how many `Restaurant` objects were created (use class attribute `count`)?
- Print the details of each `Restaurant` object (implement `__str__`)
- Update `rating` for at least one `Restaurant` object
- Compare two `Restaurant` objects. The object that has the higher `rating` will be greater.

OOP EXERCISES

- Create class `Dish` - instance attributes: `id(int), name(str), price(int)`
- Create class `Menu` - instance attributes: `dishes(list of Dish objects)`
- Implement appropriate methods so that `Menu` objects support accessing dishes by index and `in` operator:

```
d = Dish(0, 'Lasagna', 20)
m = Menu()
m.add_dish(d)
print(m[0])
print(d in m)
```

SECTION 6:
I/O OPERATIONS

BASIC FILE OPERATIONS: OPEN

- Open a file

```
file_handler = open(file, mode='r', buffering=-1, encoding=None,
   errors=None, newline=None, closefd=True, opener=None)
```

CONTEXT MANAGER AND WITH/AS

- Properly manage resources
- Syntax:

  with *action* as *var*:
- Used for manipulating files, archives, temporary files
- contextlib module: tools for creating and working with context managers

BASIC FILE OPERATIONS: ITERATING FILE OBJECTS

- File handler is an iterator

```
with open("test.txt", "r") as fout:
    for ln in fout:
        print ln

print(fout.closed) # File is automatically closed after exiting the context
```

BASIC FILE OPERATIONS

```
# Open file
fin = open("test.txt", "w")

# Write file
fin.write( "Python is a great language.\nYeah its great!!\n")

# Close file
fin.close()

# Read file
with open("test.txt", "r") as fout:
    print(fout.read()) # read the entire file -> 'Python is a great language.\nYeah its great!!\n"
    fout.seek(0,0) # position cursor at the beginning to start reading again
    print(fout.read(10)) # read first 10 bytes -> 'Python is '
    print(fout.read(10)) # read the following 10 bytes -> 'a great la'

print(fout.closed) # File is automatically closed after exiting the context
```

IO OPERATIONS EXERCISES

1. Write a Python program that reads file content and displays the number of lines that were read.
2. Write a Python program to append text into a file and displays the new content.
3. Write a function `grep` that receives `text` and `file` as parameters and returns a list with all the lines in the file containing `text`.
4. Write another function `grepinto` that receives `text`, `infile` and `outfile` as parameters and writes to `outfile` the lines in `infile` that contain `text`. Open both files within one `with` statement.

PATH OPERATIONS : OS MODULE

- `os`: Operating system functions
  - `os.listdir`: List the content of a directory
  - `os.walk`: Walks the directory tree. Returns (dirpath, dirnames, filenames)
- `os.path`: Functions on path names

PATH OPERATIONS : OS MODULE

```
import os

path = r'C:\temp\test' ;
path_to_add = os.path.join(path,'new_dir') ; path_to_add2 = os.path.join(path,'new_dir2','dir3') ; path_to_add3 = os.path.join(path,'new_dir3')

print(os.getcwd() ) #Returns the current working directory.
print(os.listdir(path)) #Return a list of the entries in the directory given by path.

os.mkdir(path_to_add) #Create a directory named path.
os.makedirs(path_to_add2) #Create directory recursively, by adding eventual missing directories

os.walk(path_to_add) #Walks the directory tree.

os.removedirs(path_to_add2) #Remove directories recursively.
os.rename(path_to_add, path_to_add3)  #Rename the file or directory.
os.rmdir(path_to_add3) #Remove (delete) the directory path.

os.system('dir') #Executes a command in the system shell
```

PATH OPERATIONS : OS.PATH MODULE

```
path = r'C:\temp\test\d1'

print("'%s' exists: %s" % (path, os.path.exists(path)))
print("'%s' is a file: %s" % (path, os.path.isfile(path)))
print("'%s' is a directory: %s" % (path, os.path.isdir(path)))
print("'%s' is a link: %s" % (path, os.path.islink(path)))

print("'%s' split: %s" % (path, os.path.split(path)))

print("The directory name for '%s': %s" % (path, os.path.dirname(path)))
print("The base name for '%s': %s" % (path, os.path.basename(path)))

print("Size of directory '%s': %s" % (path, os.path.getsize(path)))
print("Add 'new_var' dir to '%s': %s" % (path, os.path.join(path,"new_var")))
```

PATH OPERATIONS : GLOB MODULE

- glob.glob

  Returns a list of path names that are matching the pattern

- glob.iglob

  Returns an iterator of path names that are matching the pattern

```
print("Print all files with 'py' extension: %s" % glob.glob('*.py'))

for dr in glob.iglob("*.py"):
    print dr
```

COMMUNICATE WITH EXTERNAL PROCESSES : SUBPROCESS MODULE

- subprocess.run(*popenargs, input=None, capture_output=False, timeout=None, check=False, **kwargs)
- subprocess.run replaces subprocess.call and subprocess.check_output in Python 2
- subprocess.Popen - for advanced uses

COMMUNICATE WITH EXTERNAL PROCESSES : SUBPROCESS MODULE

```
import subprocess

r = subprocess.call(['ls', '-la']) # Returns the exit code
print r # 0

r = subprocess.check_output(['ls', '-la']) # Returns the string
print r # 'total 40\ndrwxr-xr-x [...]'

# The most important parameters are: cwd, env and shell
subprocess.call('ls -la', cwd='/home', env={'X': '1'}, shell=True)

proc = subprocess.Popen('ps ax | grep python && sleep 10', shell=True)
print 'PID:', proc.pid
print proc.returncode

proc.poll() # Starts the process and if it's done, gets the return code. NOT BLOCKING.
proc.wait() # Blocking. Waits for the proc to finish and returns the code.
print proc.returncode
```

PARSING COMMAND LINE ARGUMENTS: ARGPARSE MODULE

- Parser for command line options and arguments
- `ArgumentParser(prog=None, usage=None, description=None, epilog=None, parents=[], formatter_class=argparse.HelpFormatter, prefix_chars='-', fromfile_prefix_chars=None, argument_default=None, conflict_handler='error', add_help=True, allow_abbrev=True)`
- ArgumentParser object methods:

```
add_argument(name or flags...[, action][, nargs][, const][, default][,
type][, choices][, required][, help][, metavar][, dest])

print_help()

parse_args(args=None, namespace=None)
```

IO OPERATIONS EXERCISES

1. Write a Python program that prints all '.py' files from a directory.

   Give the file extension as a parameter.

   You can use the Python installation directory (e.g C:\Python27)

2. Write a Python program that pings localhost. Use subprocess module. Save command output.

   Command: 'ping localhost' or 'ping 127.0.0.1'

SECTION 7:
REGULAR EXPRESSIONS

REGULAR EXPRESSION ELEMENTS – OPTION FLAGS

Modifier	Description
re.I	Performs case-insensitive matching.
re.L	Makes \w, \W, \b, \B, \s and \S dependent on the current locale.
re.M	Makes "^" and "$" matches the beginning, respectively the end of the string and of the line. By default, it only matches at the beginning/end of string.
re.U	Makes \w, \W, \b, \B, \d, \D, \s and \S dependent on the Unicode character properties database.
re.X	Allows writing regular expressions that look nicer and are more readable. You can visually separate logical sections of the pattern and add comments
re.S	Make the '.' special character match any character at all, including a newline; without this flag, '.' will match anything *except* a newline.

REGULAR EXPRESSION ELEMENTS – ANCHORS

Pattern	Description
^	Matches beginning of line.
$	Matches end of line.
\A	Matches beginning of string.
\Z	Matches end of string. If a newline exists, it matches just before newline.
\z	Matches end of string.
\b	Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.
\B	Matches nonword boundaries.

REGULAR EXPRESSION ELEMENTS – GROUPING

Pattern	Description
[...]	Matches any single character in brackets.
[^...]	Matches any single character not in brackets
(re)	Groups regular expressions and remembers matched text.

REGULAR EXPRESSION ELEMENTS – REPETITION

Pattern	Description
re*	Matches 0 or more occurrences of preceding expression.
re+	Matches 1 or more occurrence of preceding expression.
re?	Matches 0 or 1 occurrence of preceding expression.
re{ n}	Matches exactly n number of occurrences of preceding expression.
re{ n,}	Matches n or more occurrences of preceding expression.
re{ n, m}	Matches at least n and at most m occurrences of preceding expression.
a| b	Matches either a or b.

REGULAR EXPRESSION ELEMENTS – SPECIAL SYNTAX

Pattern	Description
(?: re)	Groups regular expressions without remembering matched text.
(?#...)	Comment.
(?= re)	Specifies position using a pattern. Doesn't have a range.
(?! re)	Specifies position using pattern negation. Doesn't have a range.
(?> re)	Matches independent pattern without backtracking.

REGULAR EXPRESSION ELEMENTS – SPECIAL CHARS

Pattern	Description
.	Matches any single character except newline. Using m option allows it to match newline as well.
\w	Matches word characters.[A-Za-z0-9_]
\W	Matches nonword characters.[^A-Za-z0-9_]
\s	Matches whitespace. Equivalent to [\t\n\r\f].
\S	Matches nonwhitespace.
\d	Matches digits. Equivalent to [0-9].
\D	Matches nondigits.
\n, \t, etc.	Matches newlines, carriage returns, tabs, etc.

RE MODULE FUNCTIONS – MATCH VS SEARCH

- re.match attempts to match pattern to string
- re.search finds the first occurrence of pattern within the string
- Syntax:

  re.search(pattern, string, flags=0)

  re.match(pattern, string, flags=0)

  - *pattern* : Regular expression to be matched
  - *string*: String that would be searched to match the pattern
  - *flags:* Different flags using bitwise OR (|).
- Returns a match object on success, None on failure

RE MODULE FUNCTIONS – FINDALL VS FINDITER

- Returns all non-overlapping matches of *pattern* in *string*.
- re.findall as a list of strings
- re.finditer as an iterator of MatchObject instances
- Syntax:

re.findall(pattern, string, flags=0)

re.finditer(pattern, string, flags=0)

RE MODULE FUNCTIONS – SPLIT

- Split *string* by the occurrences of *pattern*
- Syntax:

  re.split(pattern, string, maxsplit=0, flags=0)

  - *maxsplit:* If *maxsplit* is nonzero, at most *maxsplit* splits occur, and the remainder of the string is returned as the final element of the list

RE MODULE FUNCTIONS – SUB

- Replaces all occurrences of a pattern in a string with a given replacement
- Syntax:

  re.sub(pattern, repl, string, max=0)

  - *repl:* Replacement
  - *max:* The maximum number of times to replace

RE MODULE FUNCTIONS – COMPILE

- Compile a regular expression pattern into a regular expression object.
- Used when RE matching is used many times in a single program
- The compiled versions of recent patterns are cached
- Syntax:

re.compile(pattern, flags=0)

USE CASES FOR FLAGS – MULTI-LINE MATCH

- re.DOTALL

  '.' matches all characters, including new line

- re.MULTILINE

  '^' and '$' match the beginning and end of any line within a string

ADDITIONAL FEATURES – GROUPS

- Capture groups
  - Using ()
- referencing groups
  - groups() and group([index])

- named groups
  - (?P<group_name>…)
- embedded groups
  - Numbering starts with the group that has the leftmost parenthesis

ADDITIONAL FEATURES – GROUPS

```
import re
s = "This is a regex example"

m = re.search(r"This is a (\w+) example",s) ; # capture 'regex'
print("Whole pattern: %s" % (m.group())) # or group(0)
print("Captured text: %s" % (m.group(1)))
print("List with captured groups(len = %d): %s" % (len(m.groups()),m.groups()))

m = re.search(r"This is a (?P<g1>\w+) example",s) ; # name captured group 'g1'
print("Captured text using index: %s" % (m.group(1)))
print("Captured text using name : %s" % (m.group('g1')))

m = re.search(r"This is a ((\w+) example)",s) ; # embedded groups
print("Captured text: %s" % (m.group(1))) # outer group
print("Captured text: %s" % (m.group(2))) # inner group
print("Captured text: " + repr(m.group(1,2))) # multiple groups at a time
```

ADDITIONAL FEATURES – GREEDINESS

- Greedy regular expression: .*
- Non-greedy expression: *?, +?, ??, or {m,n}?

  Will match as little text as possible

```
s = "<html> Text</html>"
print(re.match(r"<.*>",s).group()) # greedy => <html> Text</html>
print(re.match(r"<.*?>",s).group()) # non-greedy => <html>
```

REGULAR EXPRESSIONS EXERCISES

1. Write a Python program that will parse the output returned by ‘ping google.com’ command

   Print how many packets were sent, received, lost

   Print the TTL min and max value

2. Using the output above, hide the IP address of the receiver

   E.g “Pinging google.com [216.58.214.206] with 32 bytes of data:” => “Pinging google.com [x.x.x.x] with 32 bytes of data:”

SECTION 8:
PYTHON DEVELOPMENT TOOLS

DOCUMENT YOUR CODE

- Code and also code documentation readability should be one of the main concerns of any developer

- Why document my code?

  You will probably need to use it again in few months, without docs it will be hard to remember implementation logic

  You want people to be able to use your code

  You want people to be able to help out with your code

DOCSTRINGS

- Strings specified in source code similar with comments
- Used to document specific segment of code
- Unlike conventional source code comments, docstrings are parsed and loaded at runtime into the interpreter, added as metadata to runtime classes/functions/variables
- Docstrings are added as simple strings, right after class/function declaration
- Full info in PEP257 - https://www.python.org/dev/peps/pep-0257/

DOCSTRING EXAMPLE

```
class Triangle(object):
    """This is a class implementing a specific geometric form
This is the second line of the docstring
    """
```

```
print Triangle.__doc__
```

```
This is a class implementing a specific geometric form
This is the second line of the docstring
```

```
help(Triangle)
```

```
Help on class Triangle in module __main__:

class Triangle(__builtin__.object)
 |  This is a class implementing a specific geometric form
 |  This is the second line of the docstring
 |
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

SPHINX

- Sphinx is a doc framework
- In digests reStructuredText files and generates HTML/PDF/Epub outputs
- reStructuredText is plain text that uses simple and intuitive constructs to indicate the structure of a document

http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html

USING SPHINX

1. Install Sphinx and do quick setup

```
> pip install sphinx
> sphinx-quickstart
```

2. Update index.rst and point to your project/module [test]

```
.. automodule:: test
    :members:
    :undoc-members:
    :inherited-members:
    :show-inheritance:
```

3. Document your module and objects using reStructuredText syntax

```
* A thing.
* Another thing.

or

1. Item 1.
2. Item 2.
3. Item 3.
```

4. Generate your docs in ./_build folder

```
> Make html
```

TESTING FRAMEWORKS

- Testing your code is a very important step release cycle
- These are few topics that should be covered while testing your code:

  Test your code with a representative number of use cases to make sure everything works end-to-end and your module interaction with other modules works also

  Regression test your code – make sure no bugs where introduces. Also make sure that your code did not impact existing modules and functionality

  Stress test – make sure your code can handle high loads when system becomes busy

TESTING FRAMEWORKS: UNITTEST

- Unit testing framework for Python
- Part of the Python Standard Library as of Python 2.1

- The smallest building blocks of unit test is a test case implemented in unittest.Testcase class
- It can run one or several test methods along with a setup and a tear down method

http://pyunit.sourceforge.net/pyunit.html

TESTING FRAMEWORKS: UNITTEST

- Sample Unittest test case

```
import unittest


class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget("The widget")

    def tearDown(self):
        self.widget.dispose()
        self.widget = None

    def testDefaultSize(self):
        self.assertEqual(self.widget.size(), (50, 50), 'incorrect default size')

    def testResize(self):
        self.widget.resize(100, 150)
        self.assertEqual(self.widget.size(), (100, 150), 'wrong size after resize')
```

UNITTEST SUITES

- Suites are logical components that group several test cases, suites, or a mix of both of them

```
widgetTestSuite = unittest.TestSuite()
widgetTestSuite.addTest(WidgetTestCase("testDefaultSize"))
widgetTestSuite.addTest(WidgetTestCase("testResize"))

if __name__ == "__main__":
    runner = unittest.TextTestRunner()
    runner.run(widgetTestSuite())
```

TESTING FRAMEWORKS: PYTEST

- third-party framework: *pip install pytest*
- tests can be methods or functions
- error highlighting, code snippet
- parametrized tests
- custom markers
- fixtures
- See more at: https://docs.pytest.org/en/latest/index.html

TESTING EXERCISES

1. Write a generator function that searches in a file according to a regex pattern and returns all matched strings.
2. Write unit tests for the function
   - consider all cases (empty file, no matches, multiple matches across the file, multiple matches on a single line, etc)
   - do not use actual files in the tests, mock the open function.

```
import sys

flexmock(sys.modules['builtins'])
```

LOGGING: MODULE LOGGING

- Very complete, easy to use and customize logging system, replicated by other programming languages (PHP)

- Supports multiple Loggers
- One logger supports a list of Handlers
- One Handler has an associated Formatter

LOGGING: FORMATTERS

- Defines the log message format using generic terms/syntax
  - name
  - levelname
  - funcName
  - thread
  - message
  - asctime
- Supports optional parameter datefmt for date formatting in strftime format (Eg: %Y-%m-%d %H:%M:%S)
- formatter1 = logging.Formatter(fmt="[%(asctime)s %(name)-5s %(levelname)-5s %(funcName)20s() %(thread)d] %(message)s", datefmt="%Y-%m-%d %H:%M:%S")

LOGGING: HANDLERS

- Defines how log event will be processed
- Needs an associated formatter to define log event formatting
- Ignores event logs with severity lower than a configured severity

- Needs 2 parameters: Formatter, Minimum Log Severity
- Some loggers need some extra parameter depending on log event processing type. Eg: FileHandler, SyslogHandler

- ConsoleHandler
- FileHandler
- SyslogHandler
- HttpHandler
- RotatingFileHandler

LOGGING: QUICK LOGGING

```
import logging

logging.basicConfig(format='%(asctime)s %(levelname)s: %(message)s', level=logging.DEBUG,
                    datefmt='%m/%d/%Y %I:%M:%S %p')

logging.debug('This message should appear on the console')
logging.info('So should this')
logging.warning('And this, too')
```

LOGGING: COMPLEX LOGGING

```
import logging

# create simple formatter
simple_formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

# create a more detailed formatter
detailed_formatter = logging.Formatter('[%(asctime)s - %(name)s %(funcName)20s() %(process)d %(thread)d] %(levelname)s: %
(message)s')

# create a console handler
console_handler = logging.StreamHandler()
console_handler.setLevel(logging.DEBUG)

# create a file handler
file_handler = logging.FileHandler("app.log")
file_handler.setLevel(logging.DEBUG)

# We set simple log formatter for console and the detailed formatter to file handler
console_handler.setFormatter(simple_formatter)
file_handler.setFormatter(detailed_formatter)

# setup a new logger
logger = logging.getLogger("script-003-logger")
logger.setLevel(logging.DEBUG)

# Map existing handlers to this logger
logger.addHandler(console_handler)
logger.addHandler(file_handler)
```

LOGGING: COMPLEX LOGGING

```
# === Start using logger
def do_something():
    logger = logging.getLogger('example-logger')
    logger.info("Hello, this is my first informational log. This will go to console and file but with different
formatting")

def do_something_else():
    logger = logging.getLogger('example-logger')
    logger.error("Hello, this is my second log. This time it's an error log")
```

LOGGING: LOGGING CONFIGURATION

- Logging configuration can be loaded from a standalone configuration file
- The file must contain sections called [loggers], [handlers] and [formatters] which identify by name the entities of each type which are defined in the file

https://docs.python.org/2.7/library/logging.config.html

LOGGING: LOGGING FILE CONFIGURATION - OBJECTS

```
[loggers]
keys=log01,log02,log03

[handlers]
keys=hand01,hand02

[formatters]
keys=form01,form02,
```

LOGGING: LOGGING FILE CONFIGURATION - FORMATTERS

```
[formatter_form01]
format=F1 %(asctime)s - %(name)s - %
(levelname)s - %(message)s
datefmt=
class=logging.Formatter

[formatter_form02]
format=F1 [%(asctime)s - %(name)s %
(funcName)20s() %(process)d %(thread)d] %
(levelname)s: %(message)s
datefmt=
class=logging.Formatter
```

LOGGING: LOGGING FILE CONFIGURATION - HANDLERS

```
[handler_hand01]
class=StreamHandler
level=NOTSET
formatter=form01
args=(sys.stdout,)

[handler_hand02]
class=FileHandler
level=DEBUG
formatter=form02
args=('python.log', 'w')

[handler_hand03]
class=handlers.SocketHandler
level=INFO
formatter=form03
args=('localhost',
handlers.DEFAULT_TCP_LOGGING_PORT)
```

LOGGING: LOGGING FILE CONFIGURATION - LOGGERS

```
[logger_log02]
level=DEBUG
handlers=hand01
propagate=1
qualname=compiler.parser
```

PARALLEL PROCESSING: THREADING MODULE

- threading.Thread
- Represents an activity that is run in a separate thread of control
- Activity can be specified by passing a callable to constructor or by overriding run() method in a subclass
- Activity should be started by calling start() method of Thread instance
- Threads can be named by setting public attribute ‘name’
- There is also an integer identifier attribute ‘ident’ defaults to None and it’s set when activity is launched

PARALLEL PROCESSING: THREADING MODULE

- start() – Starts thread activity. Can be called only one time for a thread. It launches threads's run() method in a separate thread of control
- run() – Method representind threads's activity. Defaults to callable passed in constructor as 'target' parameter
- join([timeout]) – Blocking call that waits for specific thread to terminate
- is_alive() – Queries thread status. Returns 'True' unitl 'run()' method terminates

PARALLEL PROCESSING: THREADING MODULE

- threading.Lock
- Lowest level synchronization primitive available

- Lock.acquire([blocking=1]) – Acquire a lock. Sets Lock state to blocked. Returns only when thread owns the lock. When instantiated with blocking=0 return False immediately instead of blocking. Very useful for polling Lock.
- Lock.release() – Release a lock. Sets Lock state to unblocked. When called on an unblocked lock throws RuntimeError

PARALLEL PROCESSING: MULTIPROCESSING MODULE

- Multiprocessing module is a package that supports spawning processes using an API similar to the threading module
- Multiprocessing module allows the programmer to fully leverage multiple processors on a given machine. It runs on both Unix and Windows
- Supports communication channels between processes
- Supports sharing states between processes. This is highly not recommended. It is usually best to avoid this kind of practices.

PARALLEL PROCESSING: MULTIPROCESSING MODULE

- multiprocessing.Process
- Object used to spawn processes. Follows the API of threading.Thread

PARALLEL PROCESSING: MULTIPROCESSING MODULE

- multiprocessing.Pool
- Convenient interface for parallel execution of a function across multiple input values distributing input data and execution across several processes

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    p = Pool(5)
    print(p.map(f, [1, 2, 3]))

# When executed would return
[1, 4, 9]
```

PARALLEL PROCESSING EXERCISES

1. Write a function that returns all files with an extension (optional - if file extension is not provided, search in all files) at a certain location (recursive) as a list of strings (file names).
2. Write a function that searches in a file according to a regex pattern. Return the list of strings that match the pattern.
3. Apply the second function to all files returned by the first function.
4. Parallelise this operation, both with threading and multiprocessing. Which version runs faster?

SECTION 9:
DATABASE API

DATABASE API: CONNECTING TO DB API

- DB-API v 2.0
- Syntax:

```
db_conn =
dbmodule.connect(dbname="",host="",database="",user="",password="")
conn = db_conn.cursor()
```

- *dbmodule* can be:

  PostgreSQL: psycopg2

  Mysql: MySQLdb

  SQLite: sqlite3

DATABASE API: CREATING AND POPULATING TABLES

- Create tables

```
conn.execute ("""CREATE TABLE table_name (param1, param2)""")
```

- Populate tables

```
conn.execute ("INSERT INTO table_name VALUES (‘val1', ‘val2')")
conn.execute ("INSERT INTO table_name (param1) VALUES (‘val3')")
conn.commit()
```

DATABASE API: RETRIEVING DATA RECORDS

- Select rows

  conn.execute("SELECT * FROM table_name")

- Get rows

  All rows: rows = conn.fetchall()

  Next row: row = conn.fetchone()

DATABASE API: EXECUTING PARAMETRIZED QUERIES

- Syntax:

  conn.execute("SELECT * FROM table_name WHERE param1 = %s", "val1")

  sql = "SELECT * FROM table_name WHERE param1 = (%s)"; data = (“val1”,); cursor.execute(sql,data)

- Parameters passing

  a list: .execute ("... col = ?", ["value"])

  a tuple: .execute ("... col = ?", ("value"))

  variable arguments: .execute ("... col = ?", "value")

  a dictionary: .execute ("... col = :arg", {'arg': "value"})

  keyword args: .execute ("... col = :arg", arg = "value")

- Note! DO NOT USE ‘%’ concatenation operator

SECTION 10:
GRAPHICAL USER INTERFACES USING PYQT5

PYQT5

- Python bindings for Qt5 application framework
- available for Python 2 and 3
- implemented as a set of Python modules.
- over 620 classes and 6000 functions and methods
- multiplatform toolkit which runs on all major operating systems, including Unix, Windows, and Mac OS.
- installation:

```
pip install pyqt5
```

PYQT5

- **QApplication** manages the GUI application's control flow and main settings
- one QApplication per application, no matter whether the application has 0, 1, 2 or more windows at any given time

- **QWidget** - base class of all user interface objects
- the widget is the atom of the user interface: it receives mouse, keyboard and other events from the window system, and paints a representation of itself on the screen
- every widget is rectangular
- a widget is clipped by its parent and by the widgets in front of it
- a widget that is not embedded in a parent widget is called a window

PYQT5 - LAYOUT MANAGEMENT

- Absolute positioning
  - size and the position of a widget don't change if we resize a window
  - applications might look different on various platforms
  - if we decide to change our layout, we must completely redo our layout
- Box layout
  - horizontal: the boxes are placed in a row, with suitable sizes
  - vertical: boxes are placed in a column, with suitable sizes
- Grid layout
  - most universal layout class
  - divides the space into rows and columns

PYQT5 - EVENTS AND SIGNALS

- GUI applications are event-driven
- **event source**: the object whose state changes and generates events
- **event object**: encapsulates the state changes in the event source
- **event target**: the object that wants to be notified

- **signal and slot mechanism**
- signals and slots are used for communication between objects.
- a signal is emitted when a particular event occurs.
- a slot can be any Python callable.
- a slot is called when its connected signal is emitted.

SECTION 11:
INTRODUCTION TO THE PYTHON DJANGO WEB APPLICATION FRAMEWORK

DJANGO

- Django is Python
- Supported by Django Software Foundation
- Implements MVC philosophy
- Big community that started with 4 members
- Opened to contributors

WHO IS USING DJANGO

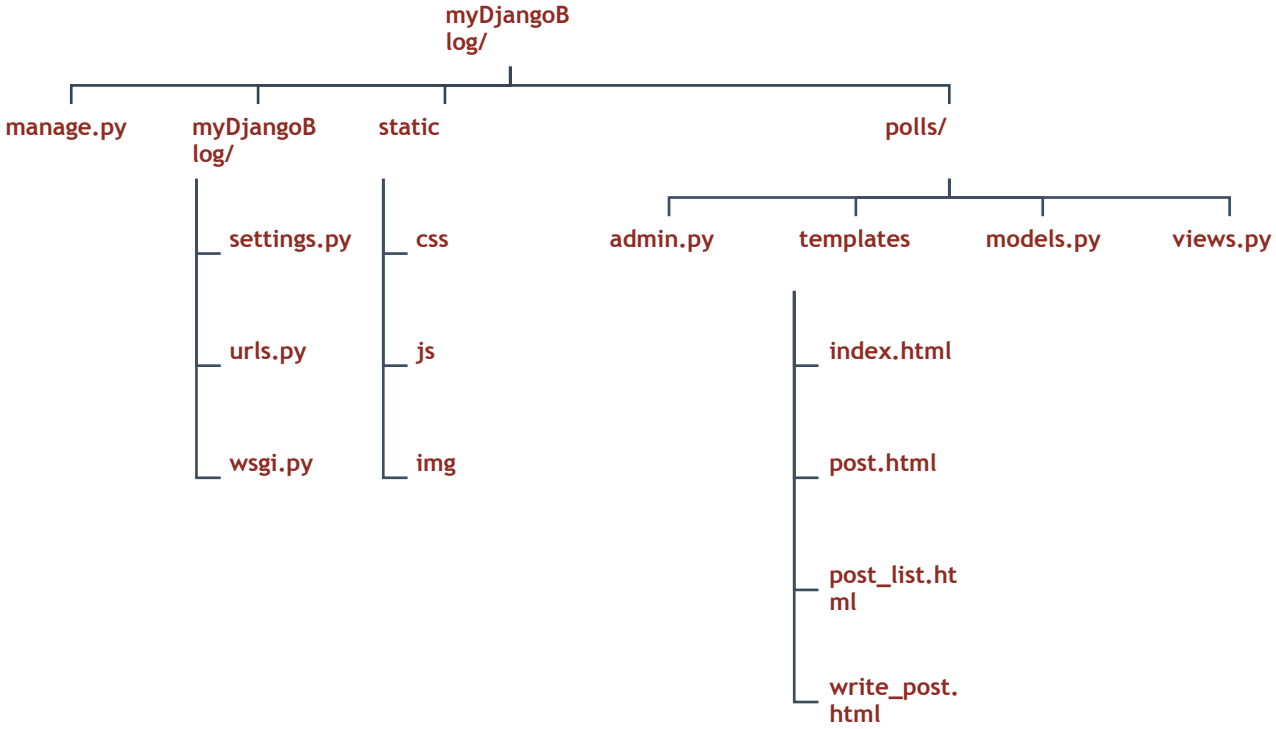
- Disqus [comment system] – 1billion unique visits per month (March, 2015)
- Instagram – over 300 million monthly active users (December, 2014)
- The Washington Post - 52.2 million unique visitors (March, 2015)
- The Guardian - 41,6 million unique visitors (October, 2014)
- Pinterest - 72.8 million users (April, 2015)
- Mozilla
- National Geographic
- Spotify
- Nasa
- Bitbucket
- PlayFire

Source: https://www.quora.com/What-are-some-well-known-sites-running-on-Django

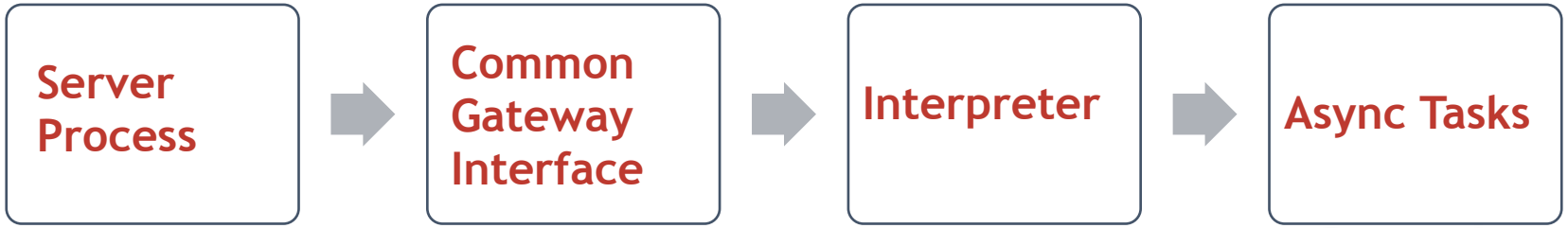
DJANGO PROJECTS AND APPLICATIONS

- Project
  - Describes an entire web application
  - Defined primarily by a settings module
  - Contains other things – static files, templates
- Application
  - Python package that provides a specific set of features
  - Application may be reused in various projects
  - Usually has specific purpose and interacts with limited parts of the framework

DJANGO FILE STRUCTURE



THE “ANATOMY” OF A WEB SERVER



- The process that interfaces with client requests. It’s process that owns the socket where client requests are coming to
- Intermediate level that uses standardized data representation to allow interpreters communicate with server process
- Worker process that process the request. Manages threads, access rights, etc.
- Extra tools used to interact with local system or launch asynchronous tasks

SERVER PROCESS

Apache

NGINX

Nginx

COMMON GATEWAY INTERFACE

- CGI
  - Fast CGI
  - Gunicorn
  - Uwsgi
  - Wsgi
- mod_php
- mod_wsgi

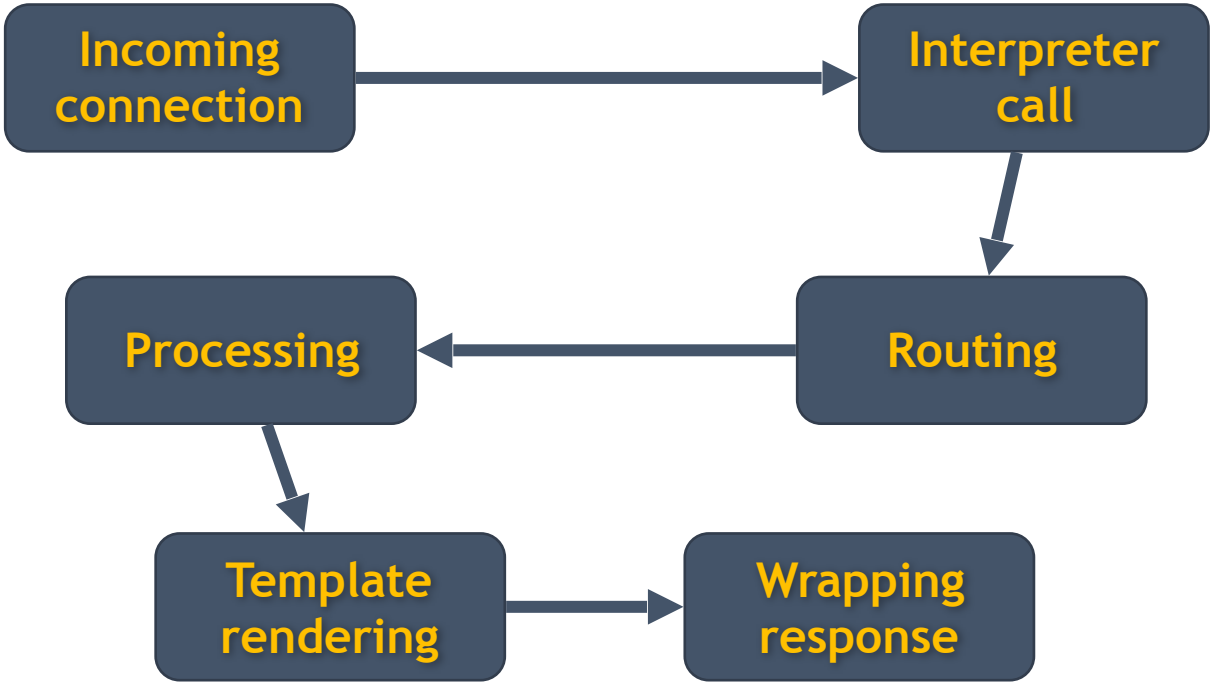
INTERPRETERS

- PHP
- Python
- Ruby
- Node (JS)

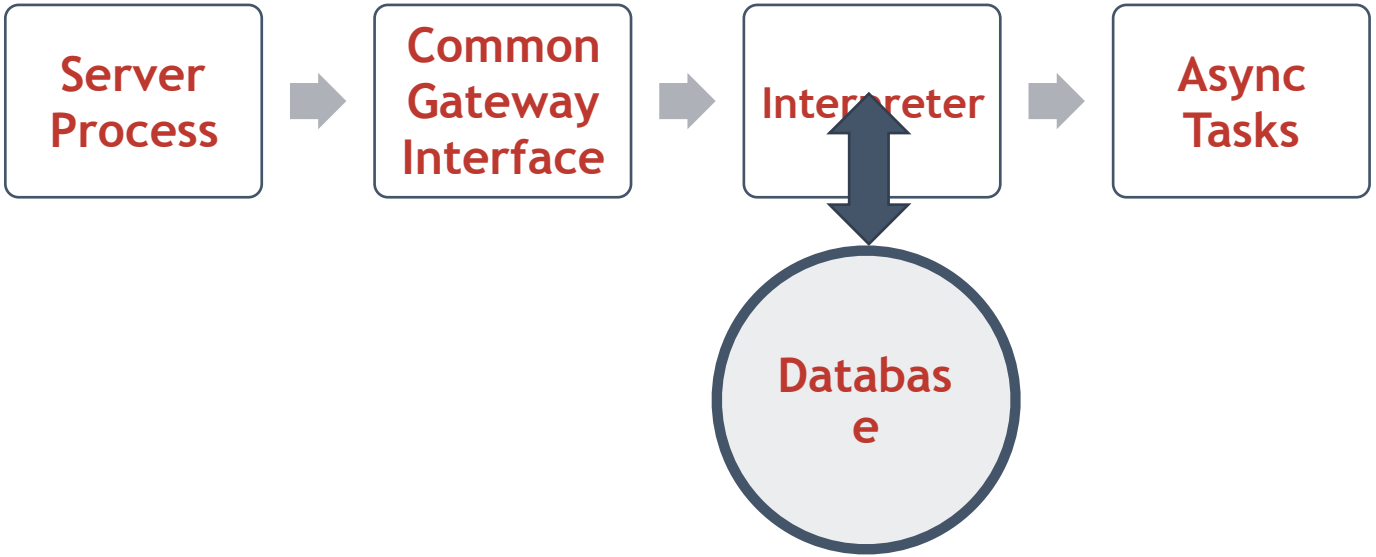
ASYNC TASK WORKERS

- Shell access - crontab
- BeanstalkD
- React
- Celery

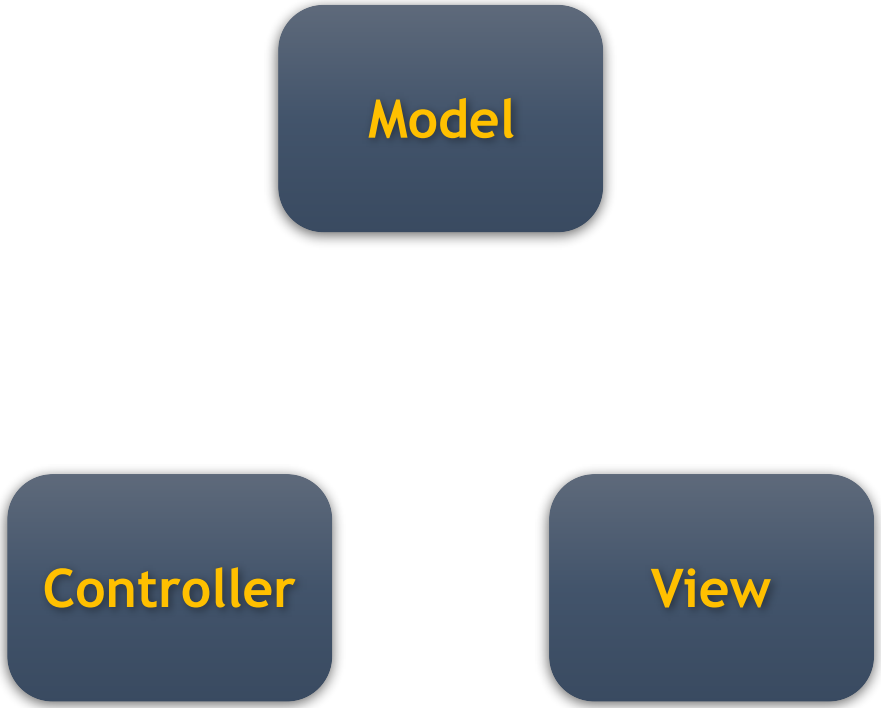
LIFECYCLE OF A HTTP REQUEST



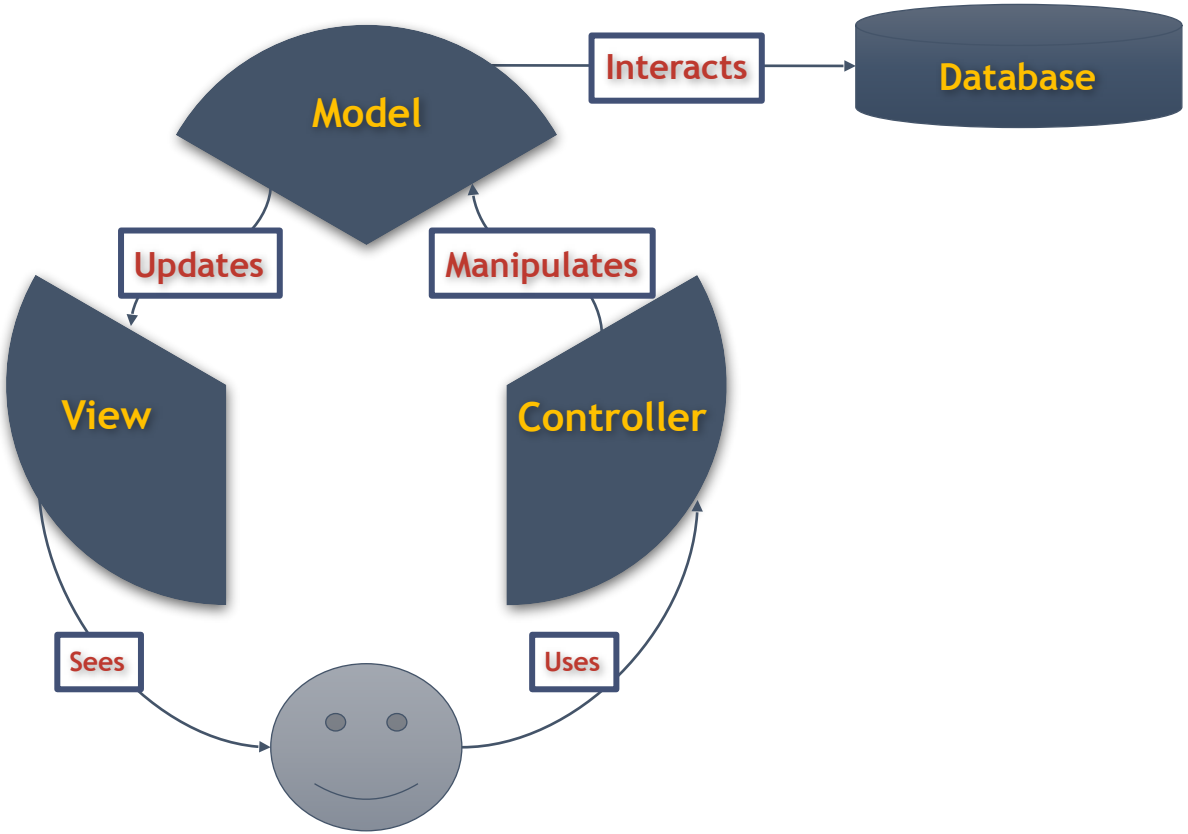
“ANATOMY” OF A WEB SERVER [EXTENDED]



MVC MODEL



MVC INTERACTIONS



CONTROLLER

- URL Dispatcher
- Strip input parameters from HTTP request
- Session / Authentication / Authorization / Validation
- User input data validation
- Implements application logic
- Handling errors exceptions
- Updates model's state
- Select view template to render
- Change view's presentation of the model

MODEL

- Maps real world model into data structures
- Contains the essential fields and behaviors of the application data
- Manages application data
- Manages database interaction
- Keeps sync between in memory app data and persistent data storage (DB)
- Responds to instructions to change state (usually from Controller)
- Responds to requests about it's state (usually from View)
- Implements in memory caching of data
- Generally, each model maps to a single database table

VIEW

- Manages the display of information
- Contains all the code needed to render models and their relations
- Usually supports specific customized programming language used for formatting and rendering controller computation results
- The most common approach relies on templates

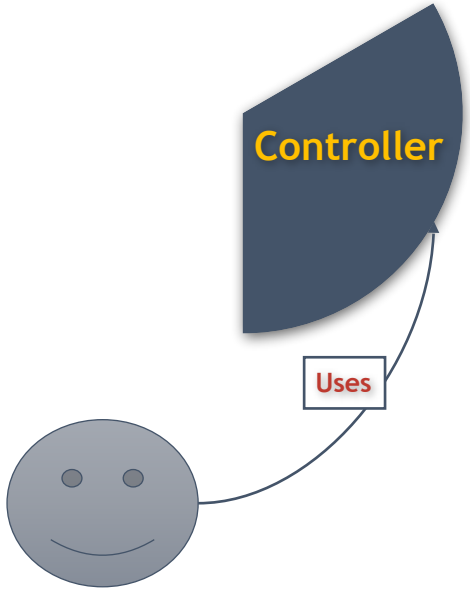
VIEW - TEMPLATES

- ❑ static parts of the desired HTML output
- ❑ special syntax describing how dynamic content will be inserted

- A convenient way to generate HTML dynamically
- Rendering templates means interpolating the template with context data and returning the resulting string
- A template contains:
  - static parts of the desired HTML output
  - special syntax describing how dynamic content will be inserted

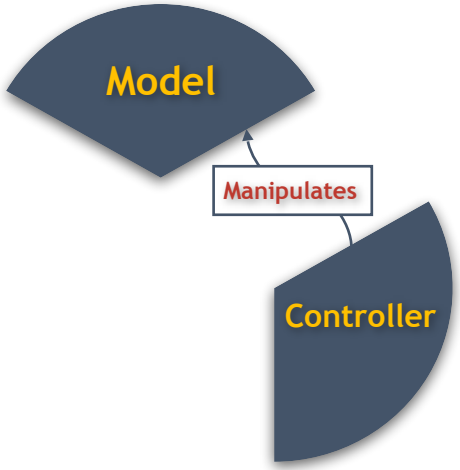
USER → CONTROLLER

- User input to controller is a combination of:
  - URIs
  - HTTP Methods
  - POST/GET parameters
- Input mechanisms
  - URL request
  - HTML forms
  - JS async requests



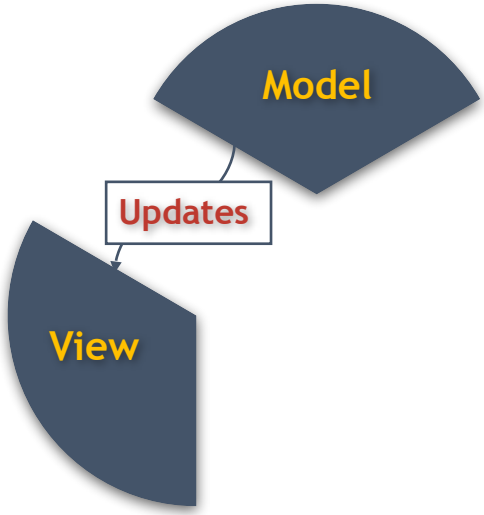
CONTROLLER → MODEL

- Instantiate handlers to application data models
- Usually this means instantiate model classes
- CRUD operations
- Read / write models attributes – (getters/setters)
- Call on model behavior – (direct method call)
    - Filter
    - Group
    - Sort



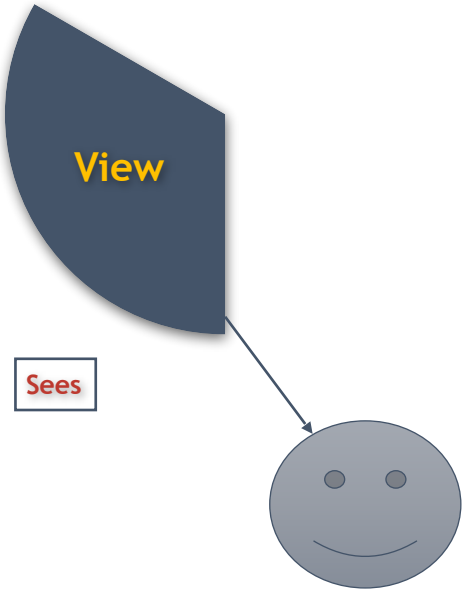
MODEL → VIEW

- Selection of appropriate template
  - Object list
  - Object display
  - 404 Page template
  - 403 Forbidden template
- Computation results to be used as dynamic data in render engine



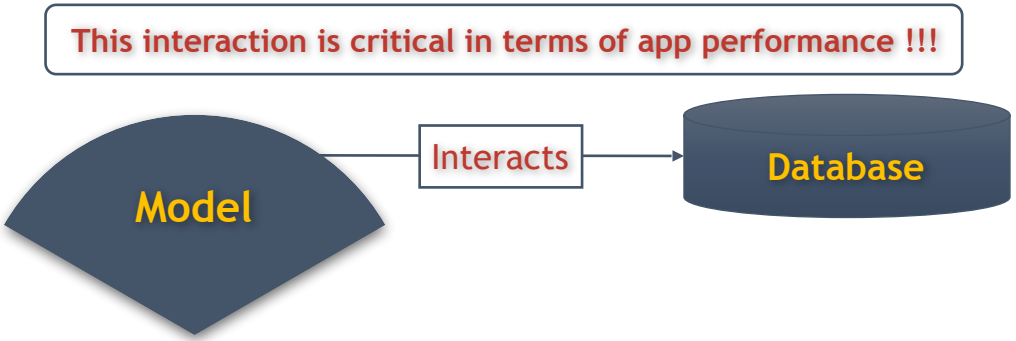
VIEW → USER

- Response pages
  - HTTP Codes
  - HTTP Pages
- XML response
- JSON response

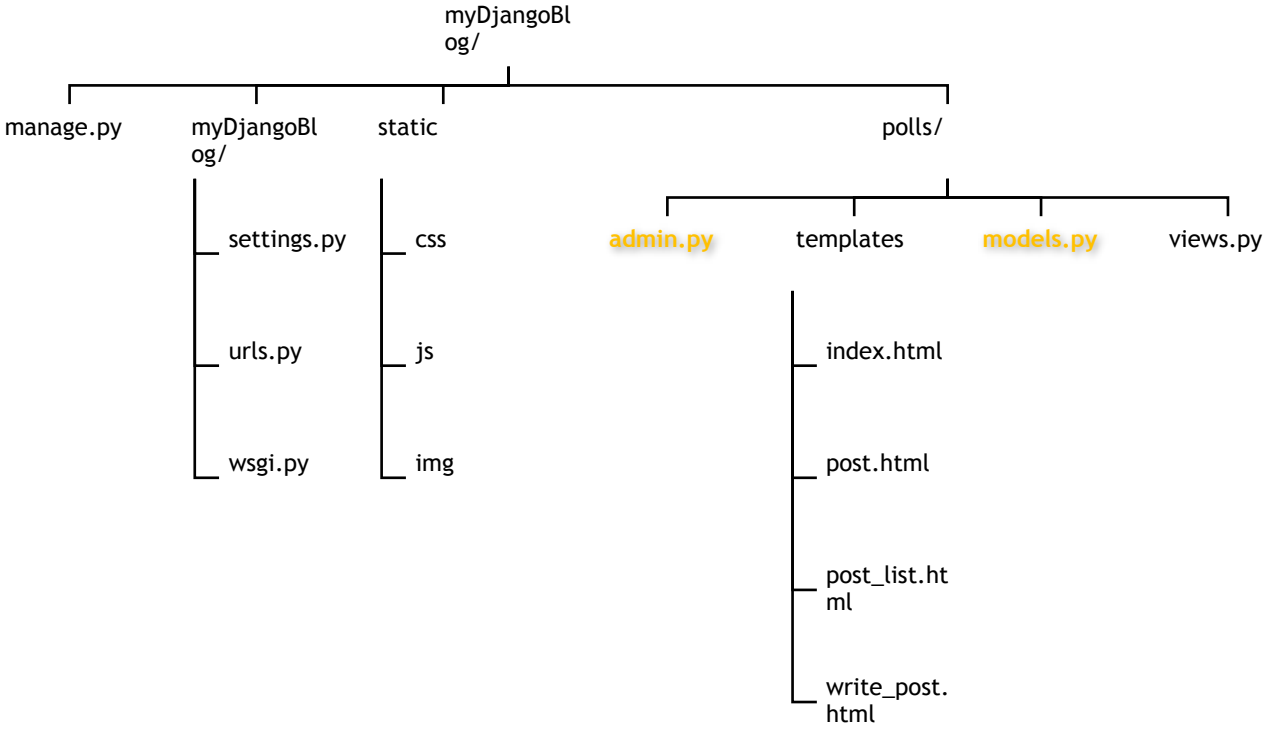


MODEL → DATABASE

- Structured query commands – usually SQL-like
- Specific CRUD directives
- Cache directives
- Data store directives – DB schema



DJANGO - MODEL



MODEL DEFINITION

- Python native class definition
- Shapes real life entities involved in interactions that application manages
- Inherits from models.Model class

Model definition is the same regardless the persistent data storage engine

MODEL DEFINITION

models.py

```
from django.db import models


class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')

    def __unicode__(self):
        return "Question %s from %s" % (self.id, self.pub_date)




class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)

    class Meta:
        ordering = ['question', 'votes']
```

MODEL DEFINITION

models.py

```
from django.db import models


class Band(models.Model):
    """A model of a rock band."""
    name = models.CharField(max_length=200)
    can_rock = models.BooleanField(default=True)


class Member(models.Model):
    """A model of a rock band member."""
    name = models.CharField("Member's name", max_length=200)
    instrument = models.CharField(choices=(
            ('g', "Guitar"),
            ('b', "Bass"),
            ('d', "Drums"),
        ),
        max_length=1
    )
    band = models.ForeignKey("Band")
```

MODEL FIELD TYPES

- AutoField
- BooleanField
- CharField
- TextField
- DateField
- DateTimeField
- FloatField
- IntegerField
- BooleanField
- BinaryField
- DecimalField
- EmailField
- GenericIPAddressField
- FileField
- ImageField
- CommaSeparatedIntegerField
- SlugField

MODEL CRUD OPERATIONS

```
class Band(models.Model):
    """A model of a rock band."""
    name = models.CharField(max_length=200)
    can_rock = models.BooleanField(default=True)

new_band = Band(name='Django Rocks', can_rock=False)
### new_band.id returns None
new_band.save()
### new_band.id returns Integer

new_band.can_rock = True
new_band.save()
### new_band.id was not None so UPDATE was used

new_band.delete()
```

ONE-TO-ONE RELATIONSHIPS

```
class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)

    def __str__(self):              # __unicode__ on Python 2
        return "%s the place" % self.name

class Restaurant(models.Model):
    place = models.OneToOneField(Place)
    serves_hot_dogs = models.BooleanField(default=False)
    serves_pizza = models.BooleanField(default=False)


restaurant1.place = {returns Place model}
place1.restaurant = {returns Restaurant model}
```

ONE-TO-MANY RELATIONSHIPS

```
class Reporter(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    email = models.EmailField()

class Article(models.Model):
    headline = models.CharField(max_length=100)
    pub_date = models.DateField()
    reporter = models.ForeignKey(Reporter, on_delete=models.CASCADE)

article1.reporter = {Returns Reporter model}
reporter1.article_set.all() = {Returns QuerySet of Article models}
```

- article_set dynamic attribute is used to span one-to-many relationship

MANY-TO-MANY RELATIONSHIPS

```
class Publication(models.Model):
    title = models.CharField(max_length=30)


class Article(models.Model):
    headline = models.CharField(max_length=100)
    publications = models.ManyToManyField(Publication)

article1.publications.all() = {returns QuerySet of
Publication models}
publication1.article_set.all() = {returns QuerySet of
Article models}
```

QUERYING MODELS

- Django uses an intuitive system:
  - A model class represents a database table
  - An instance of that class represents a particular record in the database table

- Retrieving specific objects with filters uses QuerySets API
- QuerySet is Python class with complex behavior
- It uses DB connection for reading purposes
- QuerySet can be constructed, filtered, sliced, and generally passed around without actually hitting the database
- No database activity actually occurs until you do something to evaluate the queryset

QUERYSET METHODS

- Get – Return one instance
- Get_or_create – Get mathing entry or create a new one

- All – Selects all entries
- Filter – Select only entries that match the filter
- Exclude – Exclude matching entries
- Exists – Checks if there is at least one result
- Order_by – Order entries by fields in list
- Reverse – Reverse the order of entries retrieved
- Distinct – Select distinct entries by a specific field
- Defer – Do not fetch specified fields
- Only – Fetch only specified fields

QUERYSET COMPLEX METHODS

- select_for_update – Returns a queryset that will lock rows until the end of the transaction
- select_related – Returns a QuerySet that will “follow” foreign-key relationships

- Raw – Takes a raw SQL query, executes it, and returns a django.db.models.query.RawQuerySet instance

FILTER MAGICS

- The QuerySet ‘filter’ method supports lots of lookup types

  exact

  - Entry.objects.get(headline__exact="Cat bites dog")

  iexact – case insensitive lookup

  - Blog.objects.get(name__iexact="beatles blog")

  contains – case sensitive containment test

  - Entry.objects.get(headline__contains='Lennon')

  Startswith, endswith, istartswith, iendwith

More at:
https://docs.djangoproject.com/en/1.9/ref/models/querysets/#field-lookups

FILTERS THAT SPAN RELATIONSHIPS

- Retrieves all Entry objects with a Blog whose name is 'Beatles Blog'

```
Entry.objects.filter(blog__name='Beatles Blog')
```

- Retrieves all Blog objects which have at least one Entry whose headline contains 'Lennon'

```
Blog.objects.filter(entry__headline__contains='Lennon')
```

- If there was no author associated with an Entry, it would be treated as if there was also no name attached, rather than raising an error

```
Blog.objects.filter(entry__authors__name='Lennon')
```

- However, if you need Blog entries whose Entry author is not None

```
Blog.objects.filter(entry__author__isnull=False)
```

More at:
https://docs.djangoproject.com/en/1.9/topics/db/queries/

QUERYSET CACHING

```
all_bands = Band.objects.all()
# Does not hit database

first_band = all_bands.first()
# Still not hitting database

print first_band.name
# Hits database

print first_band.name
# Does not hit database again

first_band.refresh_from_db()
# Forces hitting DB again
```

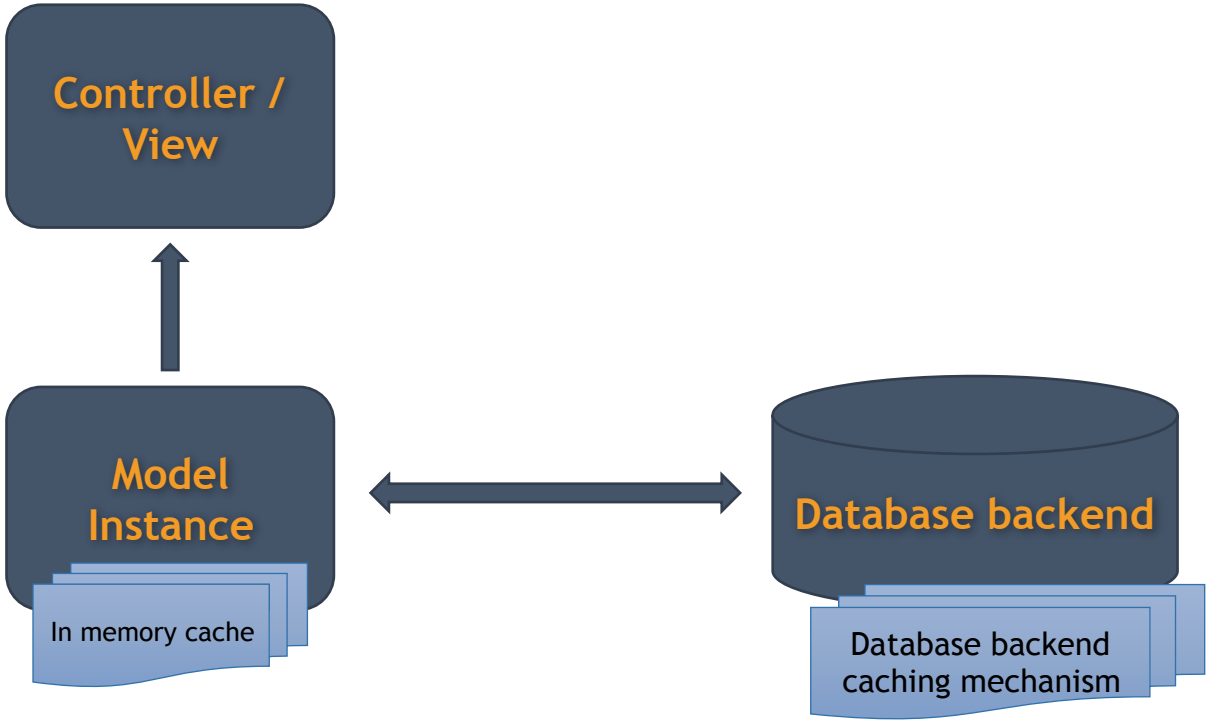
MODEL DATABASE BACKEND DEFINITION

settings.py

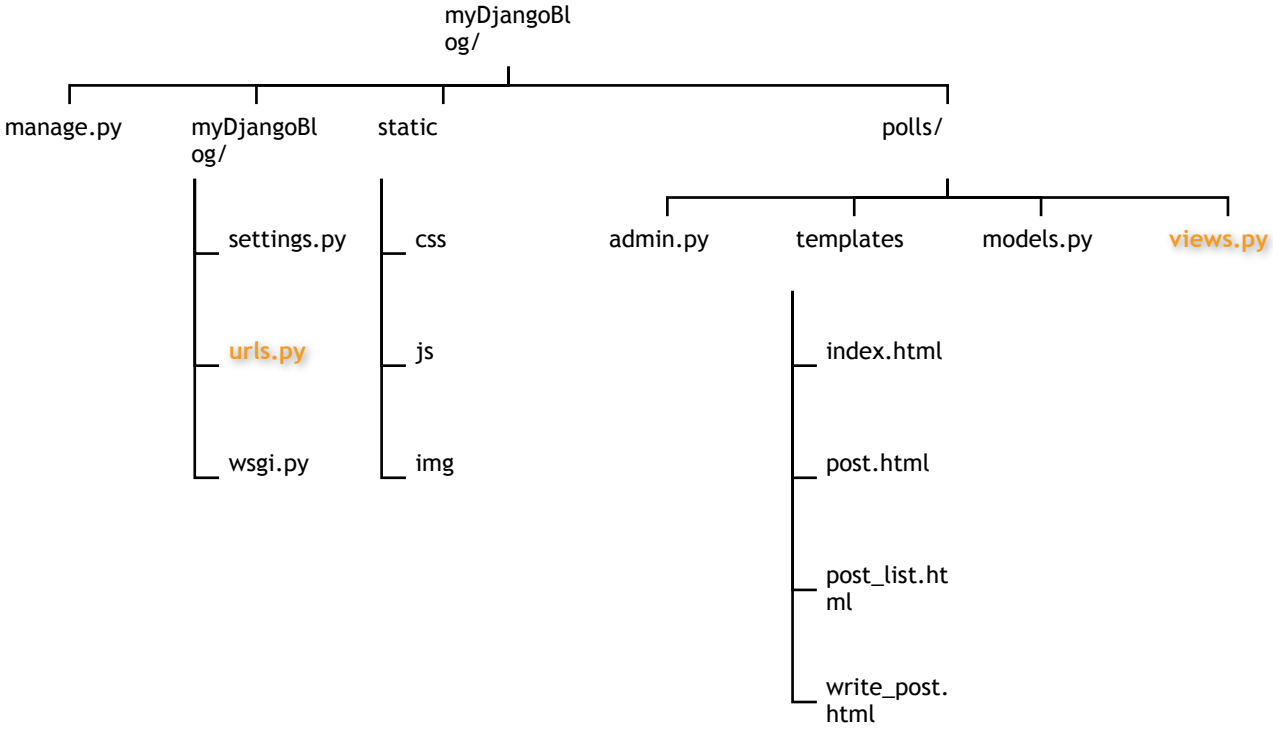
```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'database_name',
        'USER': 'database_username',
        'PASSWORD': 'database_password',
        'HOST': 'database.mydomain.com',    # Or an IP Address that your DB is hosted on
        'PORT': '3306',
    }
}
```

- Supported backends
  - PostgreSQL
  - MySQL
  - SQLite
  - Oracle

MODEL CACHING



DJANGO - CONTROLLER



URL DISPATCHER

- Clean, elegant URL scheme
- Doesn’t put any cruft in URLs, like .php or .asp
- Like a table of contents for your app
- Mapping between URL patterns and your views
- Because it’s pure Python code, it can be constructed dynamically

Cool URIs don't change https://www.w3.org/Provider/Style/URI

URL DISPATCHER DEFINITION

urls.py

```
from django.conf.urls import url

from . import views

urlpatterns = [
    # ex: /polls/
    url(r'^$', views.index, name='index'),
    # ex: /polls/5/
    url(r'^(?P<question_id>[0-9]+)/$', views.detail, name='detail'),
    # ex: /polls/5/results/
    url(r'^(?P<question_id>[0-9]+)/results/$', views.results, name='results'),
    # ex: /polls/5/vote/
    url(r'^(?P<question_id>[0-9]+)/vote/$', views.vote, name='vote'),
]
```

URL DISPATCHER ADVANTAGES

- Using a URL dispatcher mechanism allows you to have loose couple between source files and URL

- PHP traditional way:

  mysite.com/post.php?year=2016&month=11

- URL Dispatcher way:

  mysite.com/post/?year=2016&month=11

  mysite.com/post/2016/11

CONTROLLER - VIEWS

- Simply a Python function that takes a Web request and returns a Web response
- Response can be the HTML contents of a Web page, or a redirect, or a 404 error, or an XML document, or an image

- HTTP Codes, content, headers are fully customizable

VIEW SAMPLES

```
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)
```

```
from django.http import HttpResponse, HttpResponseNotFound

def my_view(request):
    # ...
    if foo:
        return HttpResponseNotFound('<h1>Page not found</h1>')
    else:
        return HttpResponse('<h1>Page was found</h1>')
```

```
from django.http import HttpResponse

def my_view(request):
    # ...

    # Return a "created" (201) response code.
    return HttpResponse(status=201)
```

URL DISPATCHER – VIEW INTERACTION

urls.py

```
from django.conf.urls import url
from . import views
urlpatterns = [
    # ex: /polls/
    url(r'^$', views.index, name='index'),
    # ex: /question_detail/
    url(r'^/question_detail/(?P<question_id>[0-9]+)/$', views.display_question_detail,
name='detail'),
    # ex: /polls/5/results/
    url(r'^(?P<question_id>[0-9]+)/results/$', views.results, name='results'),
    # ex: /polls/5/vote/
    url(r'^(?P<question_id>[0-9]+)/vote/$', views.vote, name='vote'),
]
```

views.py

```
from .models import Question

def display_question_detail(request, question_id):
    print "Looking for question %s details" % question_id
    # Rest of view processing
```

CONTROLLER - MODEL ACTION

- ‘question_id’ argument passed from URL dispatcher
- Object lookup
- Return HTTP 404 if object not found
- Render appropriate template

```
from django.http import Http404
from django.shortcuts import render

from .models import Question
# ...
def question_detail(request, question_id):
    try:
        question = Question.objects.get(pk=question_id)
    except Question.DoesNotExist:
        raise Http404("Question does not exist")
    return render(request, 'polls/detail.html', {'question': question})
```

CONTROLLER - MODEL ACTION

```
from django.shortcuts import get_object_or_404, render

from .models import Question
# ...
def question_detail(request, question_id):
    question = get_object_or_404(Question,
pk=question_id)
    return render(request, 'question_detail.html',
{'question': question})
```

- Does the something using a shortcut

HTTP METHOD SPECIFIC PROCESSING

```
def add_question(request):
    if request.method == "GET":
        # Display add question form template
        return render(request, 'add_question_form.html')
    elif request.method == "POST":
        # Add question request
        # Process POSTed data and add new entry to DB
        # When finished redirect to display template on new
created question
        return redirect("/display_question/%s" % new_question.id)
```

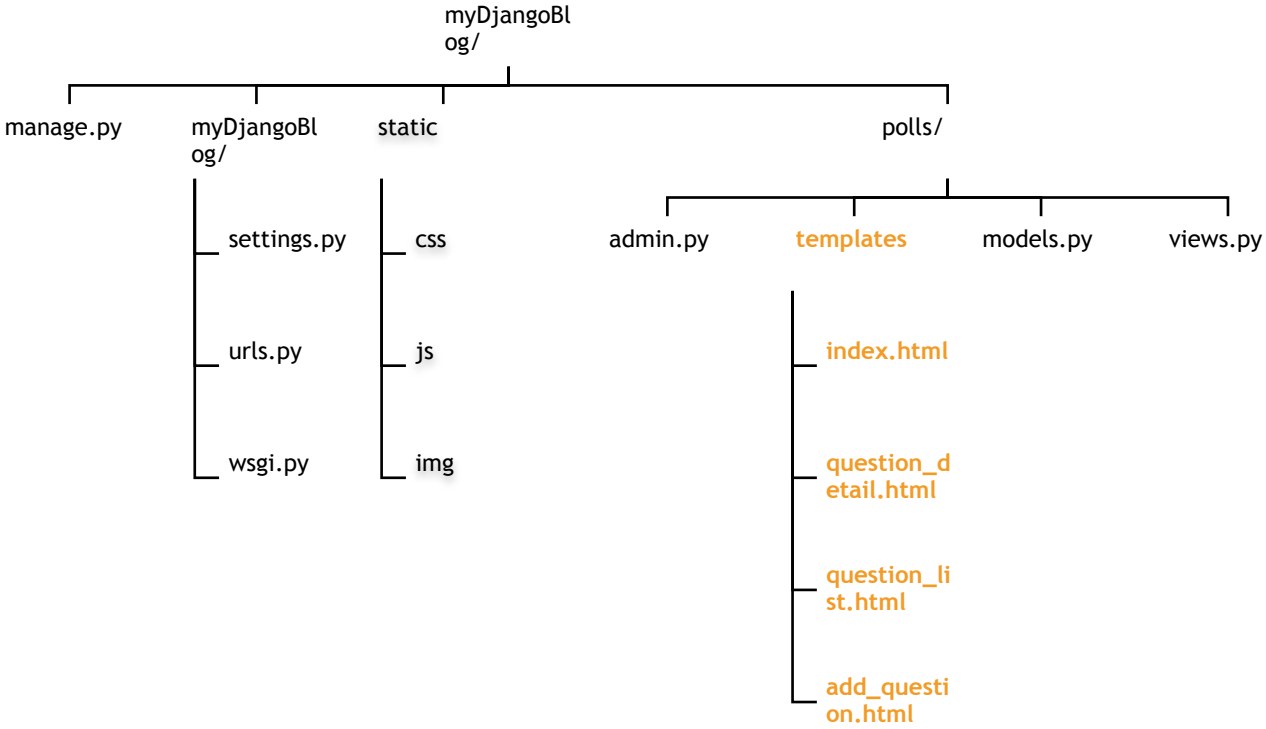
- Allows you to do different processing based on incoming HTTP request method
- Useful when implementing RESTful applications

PASSING DATA TO TEMPLATE ENGINE

```
def my_questions(request, count):
    view_context = dict()

    logged_in_user = request.user
    question_list = Question.objects.filter(created_by=logged_in_user)
    sub_list = question_list[:count]

    view_context['question_list'] = sub_list

    return render(request, 'question_list.html', context = view_context)
```

- Context variable is the bridge of communication between Controller and View
- It must be a dict structure
- Context values can be any data structures that Python supports
- Context keys must be string type

DJANGO - VIEW


myDjangoBl
og/
manage.py
myDjangoBl
og/
static
polls/
settings.py
urls.py
wsgi.py
css
js
img
admin.py
templates
models.py
views.py
index.html
question_d
etail.html
question_li
st.html
add_questi
on.html

VIEW

- Convenient way to generate HTML dynamically
- A template contains the static parts of the desired HTML output as well as some special syntax describing how dynamic content will be inserted

- Django ships built-in backends for its own template system
- Django defines a standard API for loading and rendering templates regardless of the backend
- Loading consists of finding the template for a given identifier and preprocessing it, usually compiling it to an in-memory representation
- Rendering means interpolating the template with context data and returning the resulting string

CONFIGURING TEMPLATE ENGINE

settings.py

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            # ... some options here ...
        },
    },
]
```

TEMPLATE SYNTAX - VARIABLES

My first name is {{ first_name }}. My last name is {{ last_name }}.

With a context of:

{'first_name': 'John', 'last_name': 'Doe'}

Will Render to:

My first name is John. My last name is Doe.

TEMPLATE SYNTAX - LOOKUPS

- Dictionary lookup
- Attribute lookup
- List-index lookups

```
{{ my_dict.key }}

{{ my_object.attribute }}

{{ my_list.0 }}
```

TEMPLATE SYNTAX – CONDITIONAL RENDERING

```
<span>
    Found {{ question_count}}
    {% if question_count == 0 %}
        question
    {% else %}
        questions
    {% endif %}
</span>
```

TEMPLATE SYNTAX – LOOP RENDERING

```
<table>
    <th class="table_header">
        <td>ID</td>
        <td>Content</td>
        <td>Created By</td>
    </th>
    {% for question in question_list %}
        <tr>
            <td>question.id</td>
            <td>question.content</td>
            <td>question.created_by</td>
        </tr>
    {% endfor %}
</table>
```

TEMPLATE SYNTAX – TAGS

- Useful for data formatting in human readable format

```
{{ django|title }}
```

With a context of **{'django': 'the web framework for perfectionists with deadlines'}**
Renders to: The Web Framework For Perfectionists With Deadlines

```
{{ my_date|date:"Y-m-d" }}
```

```
{% if messages|length >= 100 %} You have lots of messages today! {% endi
```

```
{{ value|default:"nothing" }}
```

```
{{ value|default_if_none:"nothing" }}
```

```
{{ blog_date|timesince:comment_date }}
```

TEMPLATE SYNTAX – TAGS [CONTINUED]

```
{{ value|yesno:“Yes,No,Not Available" }}


{% for book in books|dictsort:"author.age" %}
    * {{ book.title }} ({{ book.author.name }})
{% endfor %}

[ {'title': '1984', 'author': {'name': 'George', 'age': 45}},
  {'title': 'Timequake', 'author': {'name': 'Kurt', 'age': 75}},
  {'title': 'Alice', 'author': {'name': 'Lewis', 'age': 33}}, ]

* Alice (Lewis)
* 1984 (George)
* Timequake (Kurt)

{{ title|escape }}

{{ value|escapejs }}
```

Full list + examples:
https://docs.djangoproject.com/en/1.9/ref/templates/builtins/

TEMPLATE INHERITANCE

dashboard_base.py

```
{% block title %}
    Default title to display if not set
{% endblock %}

{% block content %}
{% endblock %}
```

manager_dashboard.py

```
{% extends “dashboard_base.html" %}
{% block title %}
    {{ section.title }}
{% endblock %}

{% block content %}
    …Render whatever needs to be rendered on Manager’s dashboard…
{% endblock %}
```

TEMPLATE – ACCESSING METHOD CALLS

- Most method calls attached to objects are also available from within templates
- Templates have access to much more than just class attributes and variables passed in from views

```
{% for person in person_list %}
    {{ person.get_full_name }}
{% endfor %}
```

```
{% for article in reporter.article_set.all %}
    {{ article.title }}
{% endfor %}
```

The template system is meant to express presentation, not program logic