Learning Python

Hamdi NEHDI

Table of contents

Python: General Overview

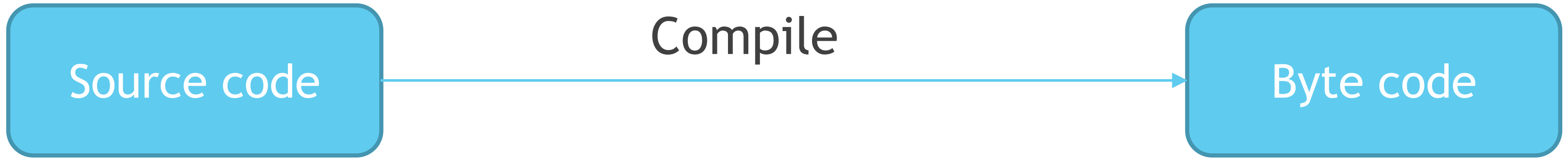
- High level programming language + dynamic typing + dynamic binding.
- A general-purpose programming language.
- Often applied in scripting roles.
- Support for OOP (Object-Oriented Programming)

Python: Advantages

- Software quality: Readability + Coherence.
- Developer productivity: Less code to type (1/5 size of C++).
- Portability: Cross computer platforms (Linux/Windows).
- Support libraries: Standard libraries + Third party libraries.

Python: Downsides

- Execution speed may not always be as fast as that of fully compiled and lower-level languages like C/C++.



Provides portability

Python: Types and Operations

- In Python data takes the form of objects (built-in objects or created objects using classes).

- The Python conceptual hierarchy:

    1. Programs are composed of modules.
    2. Modules contain statements.
    3. Statements contain expressions.
    4. Expressions create and process objects.

Python: Built-in types

- Make programs easy to write.
- Built-in objects are components of extensions: We may need to provide our own objects using Python classes and these same objects may use built-in types.
- Built-in objects are often more efficient than custom data structures.

Python’s core data types

1. Numbers:

   Integers, floating-point numbers, complexe numbers with imaginary parts, decimals, rationals …

- ✓ Numbers in Python support normal mathematical operations, for instance:

```
>>> 102 + 103                    # Integer addition
205
>>> 2.5 * 4                      # Floating-point multiplication
10.0
>>> 2 ** 20                       # 2 to the power of 20
1048576
```

Python’s core data types

1. Numbers:

✓ Numeric modules that we can use:

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.sqrt(66)
8.12403840463596
```

Python’s core data types

2. Strings:

- ✓ Strings are used to record textual information (ex: name)
- ✓ Strings are used to record arbitrary collection of bytes (ex: image file’s content)
- ✓ String in Python is a sequence: a posiotionally ordered collection of other objects.

```
>>> S = ‘Spam’
>>> len(S)
4
```

```
>>> S[0]
‘S’
>>> S[2]
‘a’
```

```
>>>S[-1]
‘m’
>>>S[-2]
‘a’
```

Python’s core data types

2. Strings:

❖ Slicing:

Extract an entire section (slice) from a string in a single step.

General form: X[I, J]

“ Give me everything in X from offset I up to but not including offset J”

I: defaults to zero

J: defaults to the length of the string

Python’s core data types

2. Strings:

- Slicing:

```
>>> S[1:3]
‘pa’
>>> S[1:]
‘pam’
```

```
>>> S[:3]
‘Spa’
>>> S[:-1]
‘Spa’
```

```
>>> S[:]
‘Spam’
```

Python’s core data types

2. Strings:

❖ Immutability:

Every string operation is defined to produce a new string as its result BECAUSE string is IMMUTABLE in Python (it cannot be changed in place after they are created).

```
>>> S[0] = ‘z’                    # Immutable objects cannot be changed
...error text omitted...
TypeError: 'str' object does not support item assignment
>>> S = ‘z’ + S[1:]               # But we can run expressions to create new objects
‘zpam’
```

Python’s core data types

2. Strings:

❖ Unicode strings:

Python has a full Unicode support required for processing text in internationalized characters sets (non ASCII text: Simple kind of Unicode).

```
>>> 'sp\xc4m'              # 3.X: normal str strings are Unicode text
'spÄm‘

>>> print u'sp\xc4m'       # 2.X: Unicode strings are a distinct type
'spÄm‘
```

Python's core data types

3. Lists:

- ✓ Lists are posiotionally ordered collections of arbitrary typed objects and they have no fixed size (but we cannot access items that are not present).
- ✓ Lists are mutable unlike strings.
- ✓ Lists can be indexed and sliced just like strings.

```
>>> L = [123, 'spam', 1.23]          # A list of three different-type objects
>>> len(L)                           # Number of items in the list
3
```

Python’s core data types

3. Lists:

```
>>> L[0] = 50                    # Mutability
>>> L[:-1]                       # Slicing a list returns a new list
[50, 'spam']
>>> L + [4, 5, 6]                # Concat/repeat make new lists too
[50, 'spam', 1.23, 4, 5, 6]
>>> L * 2
[50, 'spam', 1.23, 50, 'spam', 1.23]
>>> L                            # We're not changing the original list
[50, 'spam', 1.23]
```

Python’s core data types

3. Lists:

```
>>> L.append('NI')              # Growing: add object at end of list
>>> L
[50, 'spam', 1.23, 'NI']
>>> L.pop(2)                    # Shrinking: delete an item in the middle
1.23
>>> L                           # "del L[2]" deletes from a list too
[50, 'spam', 'NI']
```

Python’s core data types

4. Dictionaries:

- ✓ Dictionaries are not sequences at all.
- ✓ Dictionaries are known as mappings: Collection of other objects.
- ✓ Dictionaries store objects (values) by key instead of relative position.
- ✓ Dictionaries are also mutable.

Python’s core data types

4. Dictionaries:

```
>>> D = {'food': 'Spam', 'quantity': 4, 'color': 'pink'}
>>> D['food']                    # Fetch value of key 'food'
'Spam'
>>> D['quantity'] += 1           # Add 1 to 'quantity' value
>>> D
{'color': 'pink', 'food': 'Spam', 'quantity': 5}
```

Python’s core data types

4. Dictionaries:

```
>>> rec = {
        'name': {'first': ‘Hamdi', 'last': ‘Nehdi'},
        'jobs': ['dev', ‘ops'],
        'age': 28
     }
```

Python’s core data types

4. Dictionaries:

```
>>> rec['name']                    # 'name' is a nested dictionary
{'last': ‘Nehdi', 'first': ‘Hamdi'}
>>> rec['name']['last']            # Index the nested dictionary
‘Nehdi'
>>> rec['jobs']                    # 'jobs' is a nested list
['dev', ‘ops']
>>> rec['jobs'][-1]                # Index the nested list
‘ops'
>>> rec['jobs'].append(‘tutor') # Expand Bob's job description in place
>>> rec
{'age': 28, 'jobs': ['dev', ‘ops', ‘tutor'], 'name': {'last': ‘Nehdi',
        'first': ‘Hamdi'}}
```

Python’s core data types

5. Tuples:

✓ Tuples are sequences like lists.

✓ Tuples are immutables (cannot be changed like strings).

✓ Tuples are used to represent fixed collections of items

(ex: components of a specific calendar date).

Python's core data types

5. Tuples:

```
>>> T = (1, 2, 3, 4)          # A 4-item tuple
>>> len(T)                    # Length
4
>> T + (5, 6)                 # Concatenation
(1, 2, 3, 4, 5, 6)
>>> T[0]                      # Indexing, slicing, and more
1
```

Python’s core data types

5. Tuples:

- ✓ Tuples are not generally used as often as lists in practice but their immutability is the whole point ! If you pass a list in your program, it can be easily changed anywhere, but as a tuple, well you can see the difference.

Python’s other core data types

6. Files:

✓ Binary bytes files (media), Unicode text files (memo, CSV, JSON, XML).

7. Sets:

✓ Sets are neither mappings nor sequences.

✓ Sets are unordered collections of UNIQUE and IMMUTABLE objects.

Python’s other core data types

7. Sets:

```
>>> X = set('spam')          # Make a set out of a sequence in 2.X and 3.X
>>> Y = {'h', 'a', 'm'}      # Make a set with set literals in 3.X and 2.7
>>> X, Y                     # A tuple of two sets without parentheses
({'m', 'a', 'p', 's'}, {'m', 'a', 'h'})
>>> X & Y                    # Intersection
{'m', 'a'}
>>> X | Y                    # Union
{'m', 'h', 'a', 'p', 's'}
>>> X - Y                    # Difference
{'p', 's'}
```

Python: If tests/While & for loops

1. If tests:

* General format:

```
if test1:
    statements1
elif test2:
    statements2
else:
    statements3
```

* Inline format:

```
var = value1 if test else value2
```

Python: If tests/While & for loops

1. If tests:

```
>>> var = 5
>>> if var < 10:
...     print "var is valid."
... else:
...     print "var is invalid !"
...
var is valid.
```

```
>>> var = 5
>>> print "var is invalid !" if var > 10 else "var is valid."
var is valid.
```

Python: If tests/While & for loops

1. If tests:

- Truth values and Boolean tests:
  - ✓ All objects an inherent Boolean true or false value.
  - ✓ Any nonzero number or nonempty object is true.
  - ✓ Boolean operations are used to combine the results of other tests.

X and Y	*# Is true if both X as Y are true*
X or Y	*# is true if either X or Y is true*
not X	*# Is true if X is false*

Python: If tests/While & for loops

1. If tests:

```
>>> 2 < 3, 3 < 2              # Less than: return True or False (1 or 0)
(True, False)
>>> 2 or 3, 3 or 2            # Return left operand if true
(2, 3)                        # Else, return right operand (true or false)
>>> [] or 3
3
>>> [] or {}
{}
```

Python: If tests/While & for loops

1. If tests:

```
>>> 2 and 3, 3 and 2          # Return left operand if false
(3, 2)                        # Else, return right operand (true or false)
>>> [] and {}
[]
>>> 3 and []
[]
```

Python: If tests/While & for loops

2. While loops:

- ✓ Python's while statement is the most general iteration construct in the language.
- ✓ It repeatedly executes a block of statements as long as a test at the top keeps evaluating to a True value.

  - ❖ General format:

```
while test:
    statements
else:
    statements
```

Python: If tests/While & for loops

2. While loops:

```
>>> var = 5
>>> while var:
...     print "var is valid.“
...
var is valid.
var is valid.
var is valid.
 .
 .
 .                  # Infinite print
```

```
>>> var = 5
>>> while var < 10:
...     print "var is valid."
...     var += 1
...
var is valid.
var is valid.
var is valid.
var is valid.
var is valid.
```

Python: If tests/While & for loops

3. For loops:

- ✓ The for loop is a generic iterator in Python.
- ✓ It can step through the items in any ordered sequence or other ITERABLE object.

  - ❖ General format:

```
for target in object:
    statements
else:
    statements
```

Python: If tests/While & for loops

3. For loops:

```
>>> for x in ["spam", "eggs", "ham"]:
... print(x, end=' ')
...
spam eggs ham
>>> S = "lumberjack"
>>> T = ("and", "I'm", "okay")
>>> for x in S: print(x, end=' ')          # Iterate over a string
...
l u m b e r j a c k
>>> for x in T: print(x, end=' ')          # Iterate over a tuple
...
and I'm okay
```

Python: If tests/While & for loops

4. break, continue and pass:

- break: Jumps out of the closest enclosing loop (past the entire loop statement).
- continue: Jumps to the top of the closest enclosing loop (to the loop headline)
- pass: Does nothing at all (it’s an empty statement placeholder)

Python: If tests/While & for loops

4. break, continue and pass:
   - pass:

```
def func1():
    pass                # Add real code here later

if True:
    pass                # Do nothing
```

Python: If tests/While & for loops

4. break, continue and pass:
   - continue:

```
>>> x = 10
>>> while x:
...     x = x−1                 # Or, x -= 1
...     if x % 2 != 0:          # Odd? -- skip print
...         continue
...     print x
```

output ?

Python: If tests/While & for loops

4. break, continue and pass:
   - break:

```
>>> x = 10
>>> while x:
. . .    x = x−1                          # Or, x -= 1
. . .    if x % 2 != 0:                   # Odd? -- skip print
 . . .       continue
. . .    print x
. . .    If x == 4:
  . . .       Break
```

output ?

Python: Practice

4. Exercice:

   - Write a python program that checks if a given string is a Palindrome.

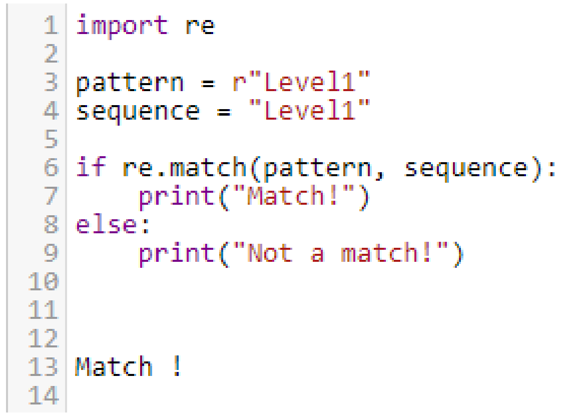
     examples: “radar”, “level”, “madam”

Python: Regular Expressions

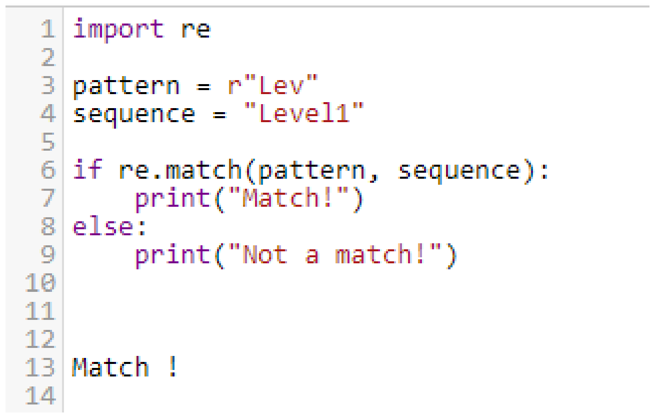
- ✓ A regular expression is a special sequence of characters that helps you MATCH or FIND other strings or sets of strings using a pattern (specialized syntax).
- ✓ Regular expressions are widely used in UNIX world.
- ✓ Regular expressions help in manipulating textual data.
- ✓ Some usage example: Validate the format of an email.
- ✓ In Python, regular expressions are supported by the “re” module.

Python: Regular Expressions

1. Basic patterns:
   - ✓ We can tackle many basic patterns in Python using ordinary characters.



```
import re

pattern = r"Level1"
sequence = "Level1"

if re.match(pattern, sequence):
    print("Match!")
else:
    print("Not a match!")



Match !
```



```
import re

pattern = r"Lev"
sequence = "Level1"

if re.match(pattern, sequence):
    print("Match!")
else:
    print("Not a match!")



Match !
```

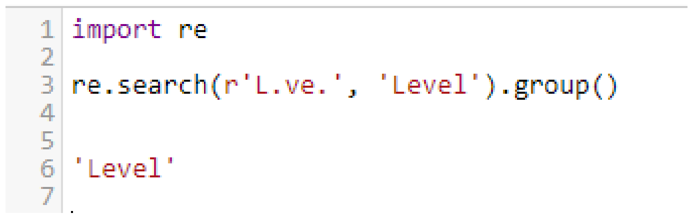
Python: Regular Expressions

1. Wild Card Characters: Special Characters

   ✓ Special characters are characters which do not match themselves as seen but actually have a special meaning when used in a regular expression.

   ❖ '.' – A period:

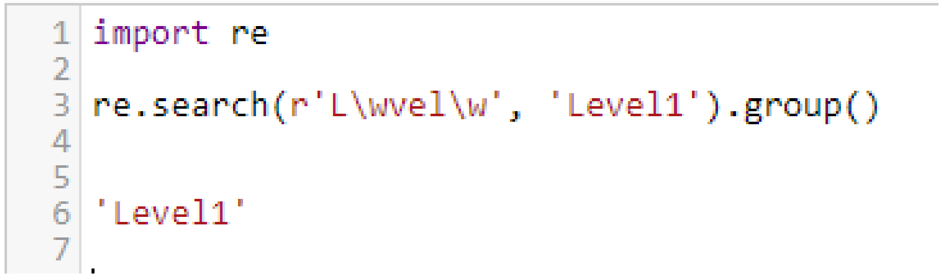
      ➢ Matches any single character except newline character.



```
import re

re.search(r'L.ve.', 'Level').group()


'Level'
```

Python: Regular Expressions

2. Wild Card Characters: Special Characters
   - ‘\w’ - Lowercase w:
     - Matches any single letter, digit or underscore.

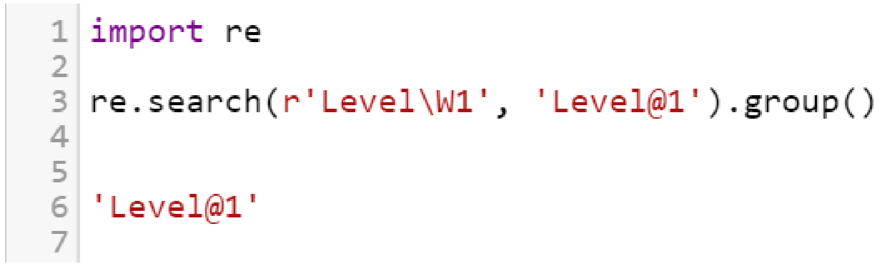


```
import re

re.search(r'L\wvel\w', 'Level1').group()


'Level1'
```

Python: Regular Expressions

2. Wild Card Characters: Special Characters

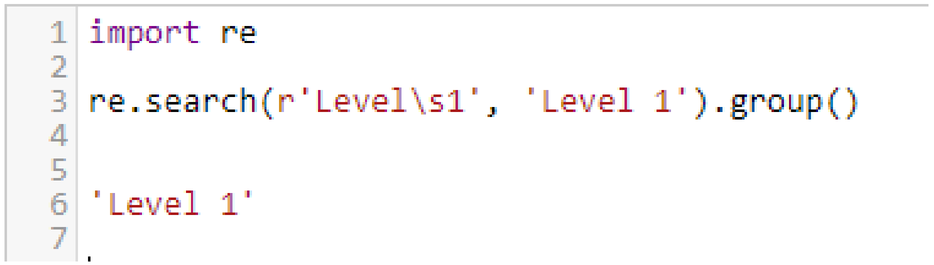
- ‘\W’ – Uppercase w:
  - Matches any character not part of \w (lowercase w).



```
import re

re.search(r'Level\W1', 'Level@1').group()


'Level@1'

```

Python: Regular Expressions

2. Wild Card Characters: Special Characters
   - ‘\s’ – Lowercase s:
     - Matches a single whitespace character like: space, newline, tab, return.



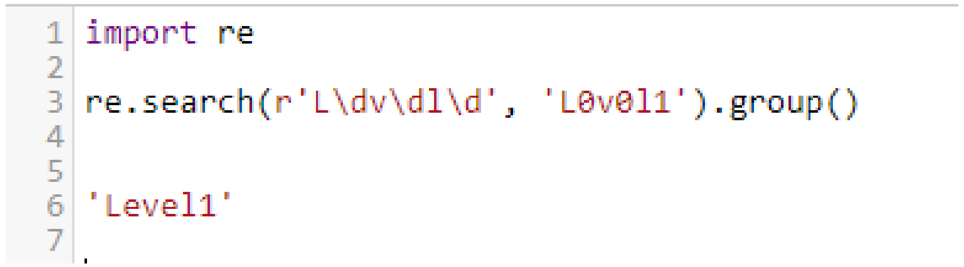
```
import re

re.search(r'Level\s1', 'Level 1').group()


'Level 1'
```

Python: Regular Expressions

2. Wild Card Characters: Special Characters

   ❖ '\d' – Lowercase d:

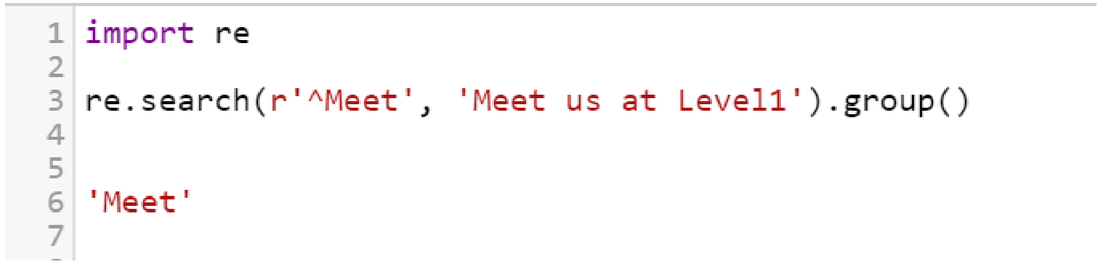
     ➢ Matches decimal digit 0-9.



```
import re

re.search(r'L\dv\dl\d', 'L0v0l1').group()


'Level1'
```

Python: Regular Expressions

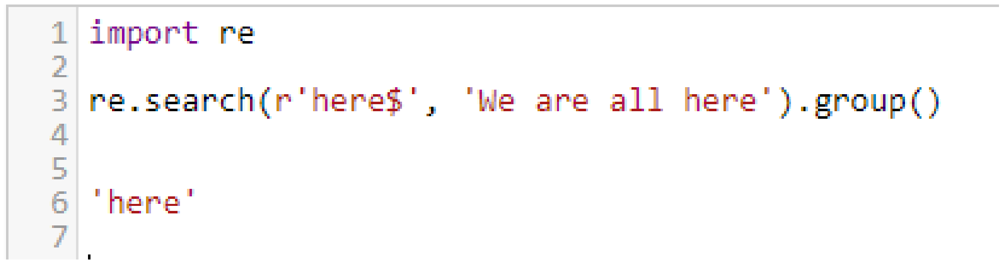
2. Wild Card Characters: Special Characters
   - ‘^’ - Caret:
     - Matches a pattern at the start of the string.



```
import re

re.search(r'^Meet', 'Meet us at Level1').group()


'Meet'

```

Python: Regular Expressions

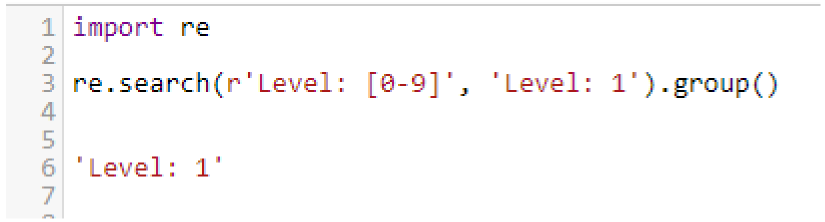
2. Wild Card Characters: Special Characters
   - '$':
     - Matches a pattern at the end of string.



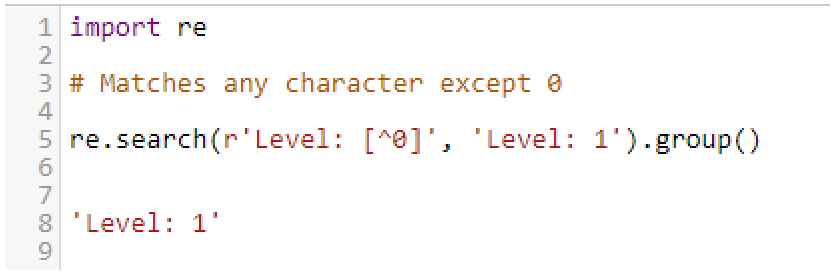
```
import re

re.search(r'here$', 'We are all here').group()


'here'
```

Python: Regular Expressions

2. Wild Card Characters: Special Characters
   - '[a-zA-Z0-9]':
     - Matches any letter from (a to z) or (A to Z) or (0 to 9).



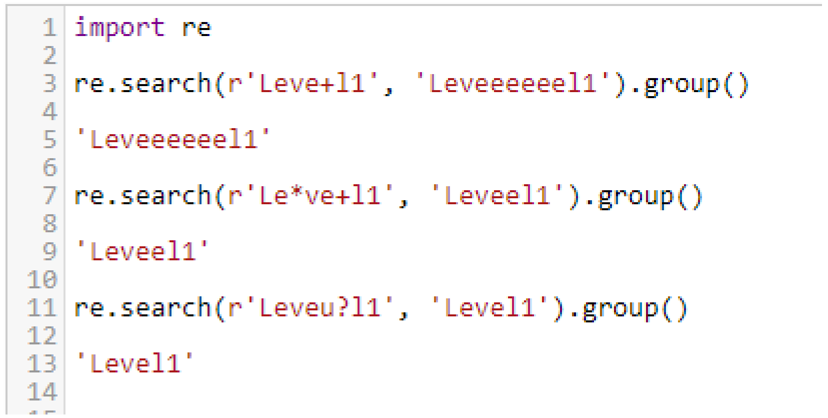
```
import re

re.search(r'Level: [0-9]', 'Level: 1').group()


'Level: 1'
```



```
import re

# Matches any character except 0

re.search(r'Level: [^0]', 'Level: 1').group()


'Level: 1'
```

Python: Regular Expressions

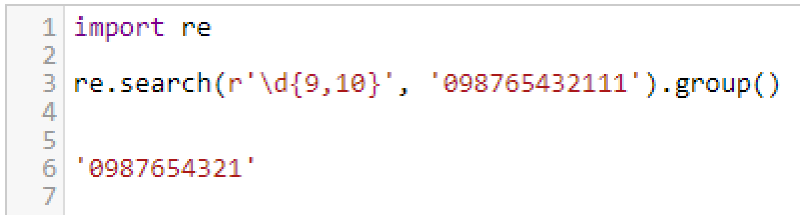
2. Wild Card Characters: Special Characters
    - '+': Checks for one or more characters to its left.
    - '*': Checks for zero or more characters to its left.
    - '?': Checks for exactly zero or one character to its left.



```
import re

re.search(r'Leve+l1', 'Leveeeeeel1').group()

'Leveeeeeel1'

re.search(r'Le*ve+l1', 'Leveel1').group()

'Leveel1'

re.search(r'Leveu?l1', 'Level1').group()

'Level1'

```

Python: Regular Expressions

2. Wild Card Characters: Special Characters
   - '{X}': Repeat exactly x number of times.
   - '{X, }': Repeat at least x times or more.
   - '{X, Y}': Repeat at least x times but no more than y times.



```
import re

re.search(r'\d{9,10}', '098765432111').group()


'0987654321'

```

Python: Regular Expressions

2. Wild Card Characters: Special Characters
   - Exercice:
     - Write a pattern that matches a valid email address in this string:

       “Please contact us at: support@datavora.com”

Python: Object-Oriented Programming

1. OOP: (Object-Oriented Programming)

- ✓ Offers a different and often more efficient way of programming, in which we factor code to minimize redundancy and write new programs by customizing existing code instead of changing it in place.

- ✓ Terminology: Class, Class variable, Data member, Instantiation, Instance, Instance variable, Object, Method, Inheritance …

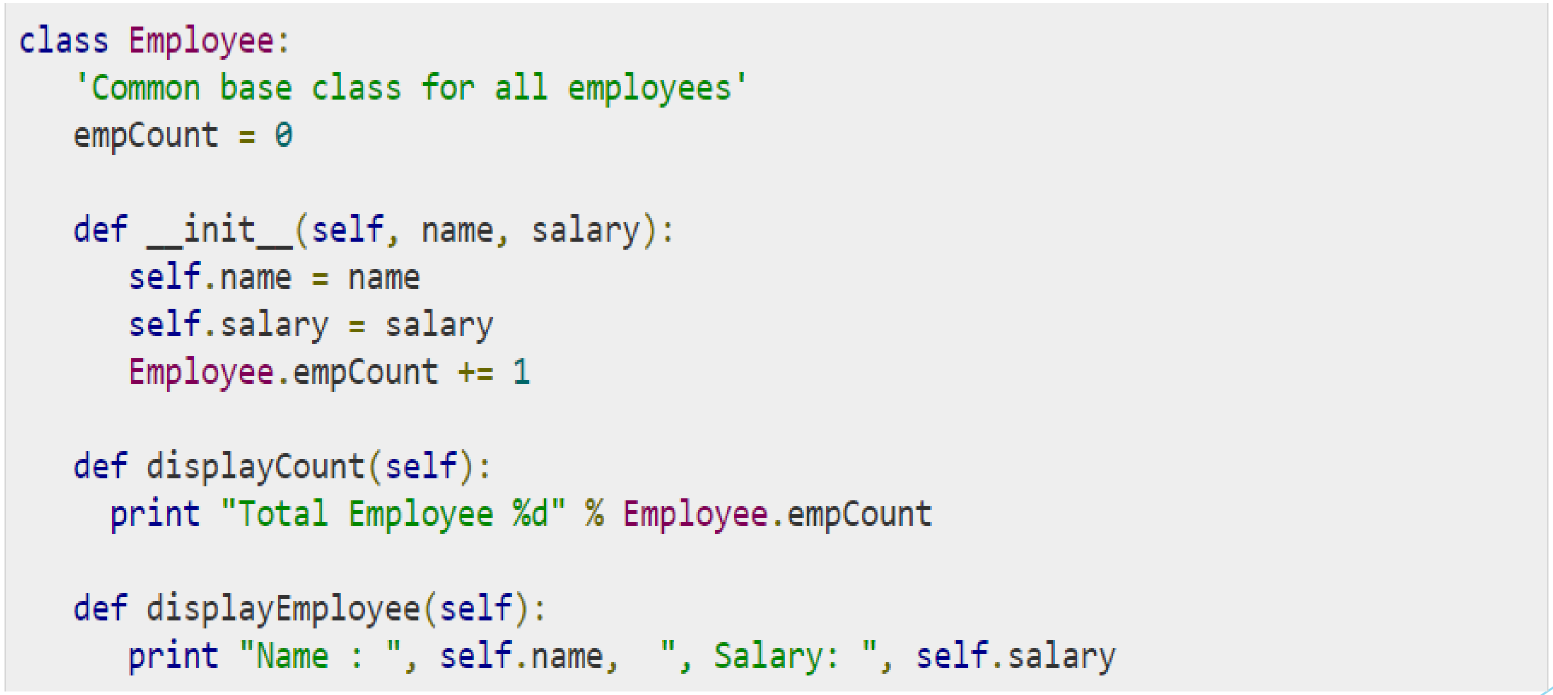
Python: Object-Oriented Programming

2. Class:

- ✓ A python class is a coding structure and device used to implement new kinds of objects that support INHERETENCE.
- ✓ Inheritance: The transfer of the characteristics of a class to other classes that are derived from it.
  - ✓ A child class can also overrides data members and methods from the parent.

Python: Object-Oriented Programming

2. Class:



```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary
```

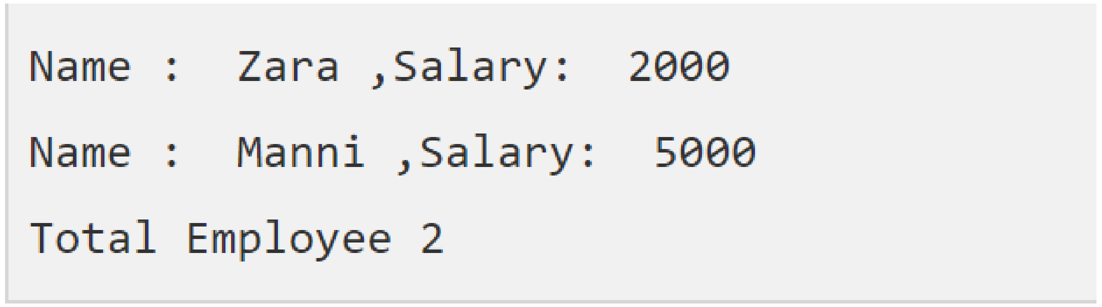
Python: Object-Oriented Programming

2. Class:

```
#!/usr/bin/python

class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary

"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

Python: Object-Oriented Programming

2. Class:



```
Name :  Zara ,Salary:  2000
Name :  Manni ,Salary:  5000
Total Employee 2
```

Python: Object-Oriented Programming

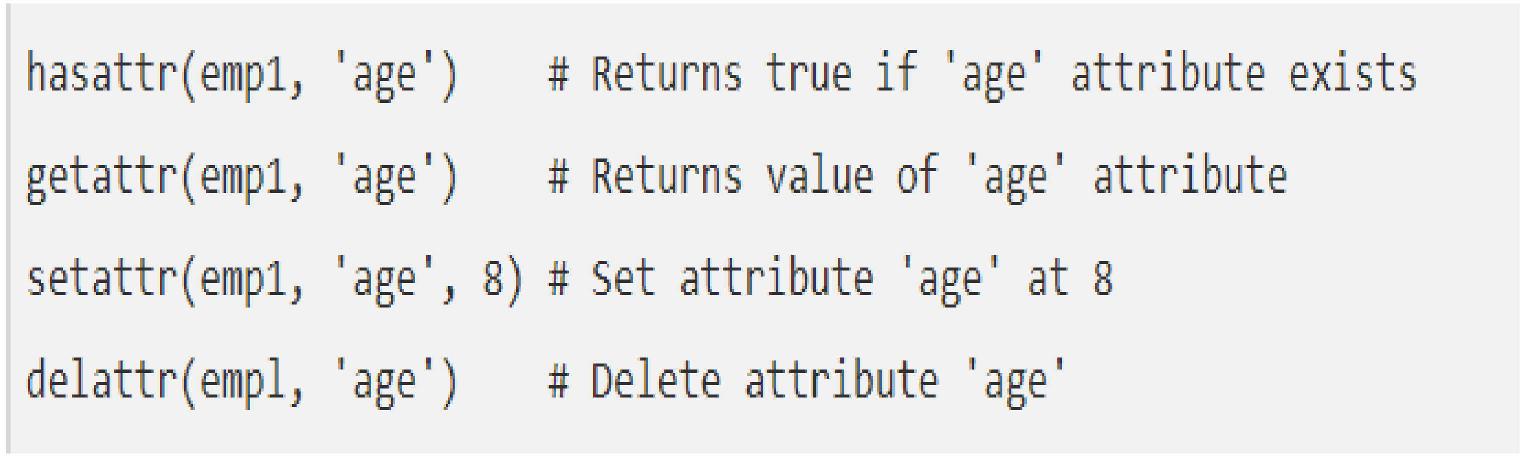
2. Class:

- ✓ We can add, remove, or modify attributes of classes and objects at any time.

```
emp1.age = 7  # Add an 'age' attribute.
emp1.age = 8  # Modify 'age' attribute.
del emp1.age  # Delete 'age' attribute.
```

Python: Object-Oriented Programming

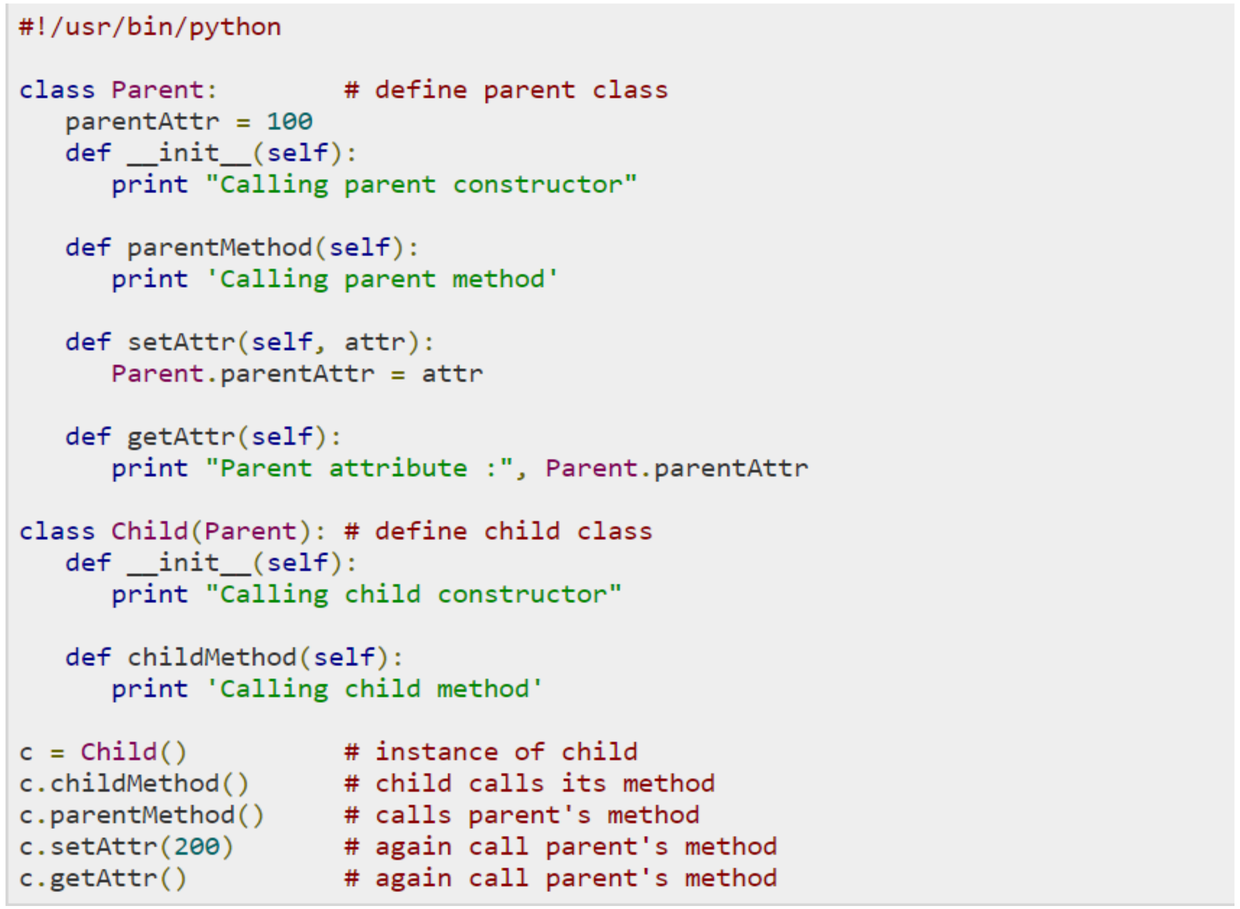
2. Class:



```
hasattr(emp1, 'age')    # Returns true if 'age' attribute exists
getattr(emp1, 'age')    # Returns value of 'age' attribute
setattr(emp1, 'age', 8) # Set attribute 'age' at 8
delattr(empl, 'age')    # Delete attribute 'age'
```

Python: Object-Oriented Programming

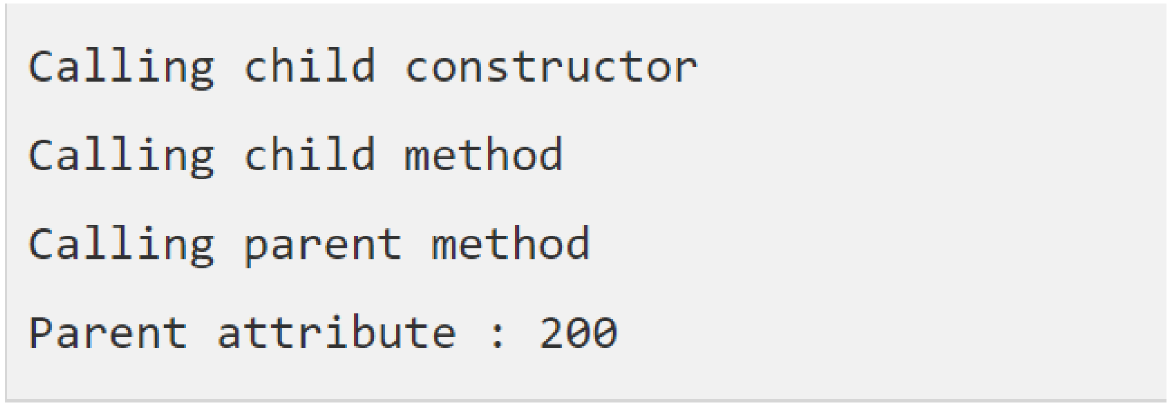
2. Class: Inheritance



```
#!/usr/bin/python

class Parent:        # define parent class
   parentAttr = 100
   def __init__(self):
      print "Calling parent constructor"

   def parentMethod(self):
      print 'Calling parent method'

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print "Parent attribute :", Parent.parentAttr

class Child(Parent): # define child class
   def __init__(self):
      print "Calling child constructor"

   def childMethod(self):
      print 'Calling child method'

c = Child()          # instance of child
c.childMethod()      # child calls its method
c.parentMethod()     # calls parent's method
c.setAttr(200)       # again call parent's method
c.getAttr()          # again call parent's method
```

Python: Object-Oriented Programming

2. Class:



```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

Python: Object-Oriented Programming

2. Class:

- ✓ Similar way, you can drive a class from multiple parent classes as follows.

```
class A:        # define your class A
.....

class B:         # define your class B
.....

class C(A, B):   # subclass of A and B
.....
```

Python: Object-Oriented Programming

2. Class: Overriding Methods

```
#!/usr/bin/python

class Parent:        # define parent class
   def myMethod(self):
      print 'Calling parent method'

class Child(Parent): # define child class
   def myMethod(self):
      print 'Calling child method'

c = Child()          # instance of child
c.myMethod()         # child calls overridden method
```

```
Calling child method
```

Python: Object-Oriented Programming

3. Polymorphism:

- ✓ Polymorphism is an important feature of class definition in Python that is utilized when you have commonly named methods across classes or subclasses. This allows functions to use objects of any of these polymorphic classes without needing to be aware of distinctions across the classes.

- ✓ Polymorphism allows for flexibility and loose coupling so that code can be extended and easily maintained over time.

Python: Object-Oriented Programming

3. **Polymorphism:**

```
class Shark():
    def swim(self):
        print("The shark is swimming.")

    def swim_backwards(self):
        print("The shark cannot swim backwards, but can sink backwards.")

    def skeleton(self):
        print("The shark's skeleton is made of cartilage.")


class Clownfish():
    def swim(self):
        print("The clownfish is swimming.")

    def swim_backwards(self):
        print("The clownfish can swim backwards.")

    def skeleton(self):
        print("The clownfish's skeleton is made of bone.")
```

Python: Object-Oriented Programming

3. Polymorphism:

```
...
sammy = Shark()
sammy.skeleton()

casey = Clownfish()
casey.skeleton()
```

```
Output
The shark's skeleton is made of cartilage.
The clownfish's skeleton is made of bone.
```

Python: Object-Oriented Programming

3. **Polymorphism:**

```python
...
sammy = Shark()

casey = Clownfish()

for fish in (sammy, casey):
    fish.swim()
    fish.swim_backwards()
    fish.skeleton()
```

```
Output
The shark is swimming.
The shark cannot swim backwards, but can sink backwards.
The shark's skeleton is made of cartilage.
The clownfish is swimming.
The clownfish can swim backwards.
The clownfish's skeleton is made of bone.
```

Thank you for your attention