Introduction to Python

T5 Bootcamp by SDAIA

SDAIA
الهيئة السعودية للبيانات
والذكاء الاصطناعي
Saudi Data & AI Authority

Object Oriented Programming

SDAIA
الهيئة السعودية للبيانات
والذكاء الاصطناعي
Saudi Data & AI Authority

Outline

- Immutability
- Classes and Objects
- Instantiation
- Methods
- Encapsulation
- Abstraction
- Polymorphism
  - Interfaces
  - Inheritance

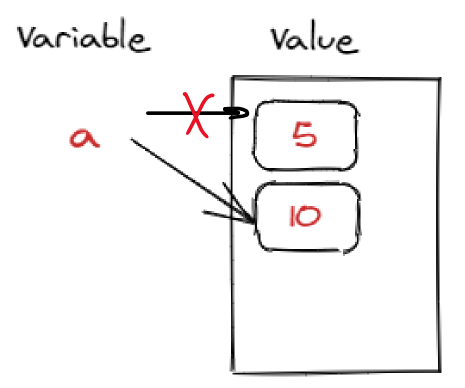
Immutability

Immutability

- A **Variable** is a named place in the memory where a programmer can store data and later retrieve the data using the variable name
- In Python, everything is an ***Object***; that is, it has address, properties, and methods.

- A variable stores the pointer to the object holding the value; and not literally the value.
- When **reassigning**, we change the pointer and create another object.

```
a = 5
a = 10
```



Mutable and Immutable Objects

- Lists are mutable!
- Reassigning an element changes only that element, and doesn't return a new list.
  - How do we know? The address of the list **L** doesn't change.
- Strings are immutable!
  - The address for s before and after the addition of "C" is different.
- **Mutable**: list, dict, set
- **Immutable**: str, tuple

```
L = [1, 2, 3]
print(hex(id(L)))
>> 0x22216057500

L[1] = 50
print(hex(id(L)))
>> 0x22216057500
```

```
s = "AB"
print(hex(id(s)))
>> 0x222174d1970

s += "C"
print(hex(id(s)))
>> 0x22216030c30
```

Why do we care: mutable and immutable?

- **Mutability** in lists allow sorting in-place to reduce memory footprint
- **Immutability** in strings eliminates the chance that it might be changed unexpectedly throughout the code, reducing the chance of errors.
- Remember: objects and functions pass around values. If values being passed are immutable, you are guaranteed they won't change.

Classes and Objects

Objects in Python behave differently

Q1. What is: [1, 2, 3] + [4, 5, 6]?

A) addition

B) concatenation

C) error

Objects in Python behave differently

Q1. What is: [1, 2, 3] + [4, 5, 6]?

A) addition

B) concatenation

C) error

Q2. What is: [1, 2, 3] * [4, 5, 6]?

A) element-wise addition

B) dot product

C) error

Objects in Python behave differently

Q1. What is: [1, 2, 3] + [4, 5, 6]?

A) addition

B) concatenation

C) error

Q2. What is: [1, 2, 3] * [4, 5, 6]?

A) element-wise addition

B) dot product

C) error

Q3. What is: [1, 2, 3] * 5?

A) element-wise multiplication

B) the list repeated 5 times

C) error

Objects in Python behave differently

Q1. What is: [1, 2, 3] + [4, 5, 6]?

- A) addition
- B) concatenation
- C) error

Q3. What is: [1, 2, 3] * 5?

- A) element-wise multiplication
- B) the list repeated 5 times
- C) error

Q2. What is: [1, 2, 3] * [4, 5, 6]?

- A) element-wise addition
- B) dot product
- C) error

Q4. What is: [1, 2, 3] - 3?

- A) element-wise subtraction
- B) remove item 3
- C) error

Python Classes and Objects

- Python is an object oriented programming (OOP) language.
- int, str, list are some of Python's built-in data types / objects.
- We can define our own class of objects, then create instances of them.

- A **Class** is a "blueprint" for creating objects
- An **Object** is two-folds:
  - **Properties** (a.k.a., data, state) - variables that belong to an object
  - **Methods** (a.k.a, operations, behavior) - functions that belong to an object

Point: Class Definition

- Let's say we want to define what a **Point** is to Python
- What properties would a point have?
- What functions would a point have?

- Note: **self** refers to an **Instance/object** of this class; i.e., to any Point created out of it.

```
class Point:

  def __init__(self, x, y):
    self.x = x
    self.y = y

  def move(self, dx, dy):
    self.x += dx
    self.y += dy
```

Point: Object Instantiation

- We can now create an instance / object of the **Point** class

```python
class Point:

  def __init__(self, x, y):
    self.x = x
    self.y = y

  def move(self, dx, dy):
    self.x += dx
    self.y += dy
```

```python
p1 = Point(2, 4)
p2 = Point(5, 7)

print(p1.x)
>> 2
print(p2.y)
>> 7

p2.move(2, 2)
print(p2.x, p2.y)
>> 7 9
```

Point: more methods (and representation)

- Let's define the Euclidian distance function between points
- Let's also controls how this object is **repr**esented (defaults to address)
  - Note: methods that start with two underscores (“__”) are called dunder methods. Such methods serve special purposes in Python.

```
  def distance(self, other):
    return ((self.x - other.x) **2 + (self.y - other.y) **2) **0.5

  def __repr__(self):
    return f"({self.x}, {self.y})"
```

Point: Full Class Definition

```python
class Point:
  def __init__(self, x, y):
    self.x = x
    self.y = y

  def move(self, dx, dy):
    self.x += dx
    self.y += dy

  def distance(self, other):
    return ((self.x - other.x) **2 + (self.y - other.y) **2) **0.5

  def __repr__(self):
    return f"({self.x}, {self.y})"
```

Encapsulation

- ***Encapsulation*** prevents direct access (read) and modification (write) of a variable (property)
- Logic can be inserted before and after the intended operation
- Here the **rank** of **Person** is encapsulated

```python
class Person:
  def __init__(self, name, rank):
    self.name = name   # public variable
    self.__rank = rank # private

  def promote(self, steps):
    if steps > 0:
      self.__rank += steps
    else:
      raise ValueError("must be positive")

  @property
  def rank(self):
    return self.__rank
```

Encapsulation

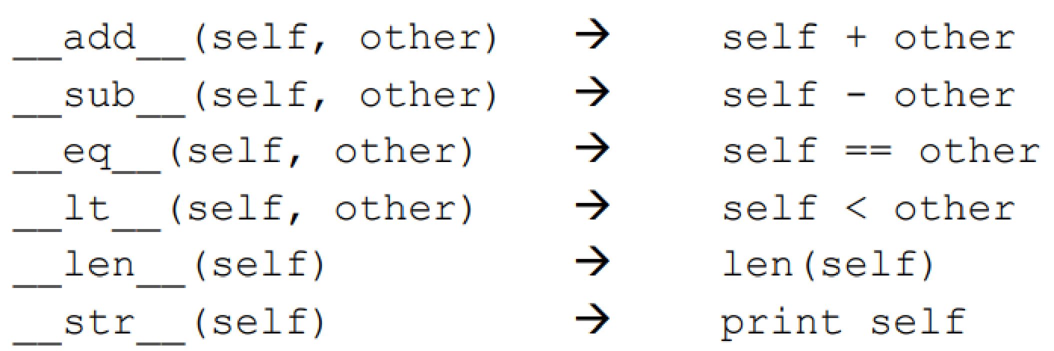
```python
class Person:
  def __init__(self, name, rank):
    self.name = name   # public variable
    self.__rank = rank # private

  def promote(self, steps):
    if steps > 0:
      self.__rank += steps
    else:
      raise ValueError("must be positive")

  @property
  def rank(self):
    return self.__rank
```

```
p = Person("Ahmad", 10)
p.promote(5)
p.rank
>> 15

p.promote(-5)
>> ValueError: must be positive
```

Note: because of **@property** ***decorator*** the **rank** method is accessed like a property; i.e., without parenthesis ().

Abstraction

- **Abstraction** makes a distinction between: how operations are implemented vs. how to the object is used
- +, -, ==, <, >, len(), print, and many others have special dunder methods to implement them:



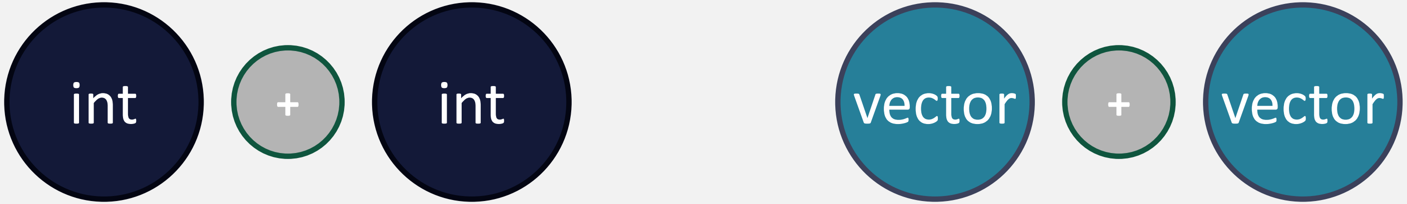
`__add__(self, other)`	→	`self + other`
`__sub__(self, other)`	→	`self - other`
`__eq__(self, other)`	→	`self == other`
`__lt__(self, other)`	→	`self < other`
`__len__(self)`	→	`len(self)`
`__str__(self)`	→	`print self`

See: https://docs.python.org/3/reference/datamodel.html#basic-customization

Vector Objects

- Let‘s define what a **Vector** is to Python
- What properties should a vector have?
- What functions should a vector have?

- Abstraction makes it possible to use Vectors as we use ints without worrying about how addition, multiplication, and other operations are implemented.



int + int

vector + vector

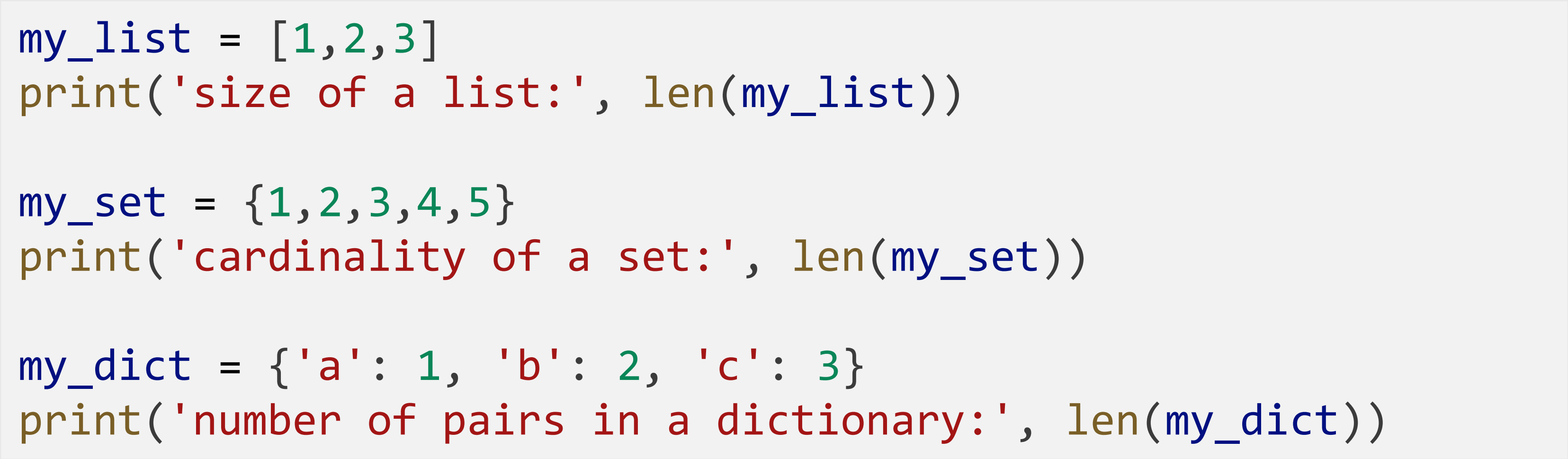
Vector: + Operator Overloading

```
class Vector:
  def __init__(self, x, y, z):
    self.x = x
    self.y = y
    self.z = z

  def __repr__(self):
    return f"<{self.x}, {self.y}, {self.z}>"

  # + operator overloading
  def __add__(self, other):
    return Vector(self.x + other.x, self.y + other.y, self.z + other.z)
```

We define + between **Vector** objects as the element-wise addition of its (x, y, z) that results a new Vector object. Hence, the "return Vector" statement.

Polymorphism: usage from built-in library

- What makes len() work on many different types in Python? Polymorphism!



```
my_list = [1,2,3]
print('size of a list:', len(my_list))

my_set = {1,2,3,4,5}
print('cardinality of a set:', len(my_set))

my_dict = {'a': 1, 'b': 2, 'c': 3}
print('number of pairs in a dictionary:', len(my_dict))
```

Polymorphism through **Interfaces**

```
def len(__obj: Sized) -> int: ...
```

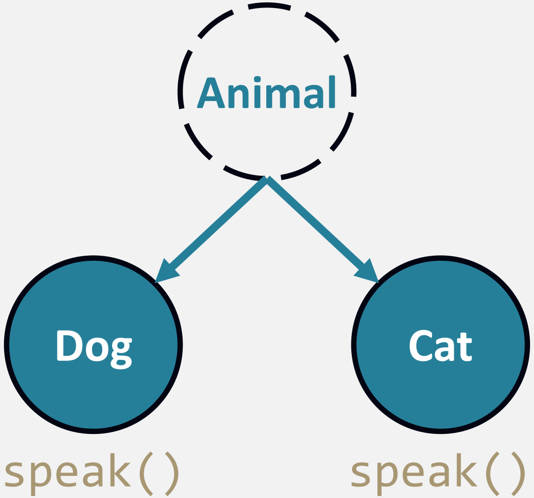
- **Sized** interface is something that has the method __len__

- Let's implement the ***interface*** that the function len() assumes about the object passed to it as an argument.

```
class Vector:
    def __len__(self):
        return 3

v = Vector()
len(v)
```

Polymorphism through **Inheritance**

```
class Animal:
  def __init__(self, name, level):
    self.name = name
    self.level = level
  def speak(self):
    pass

class Dog(Animal):
  def speak(self):
    return f"Woof! " * self.level

class Cat(Animal):
  def speak(self):
    return f"Meow! " * self.level
```



Thank you

SDAIA
الهيئة السعودية للبيانات
والذكاء الاصطناعي
Saudi Data & AI Authority