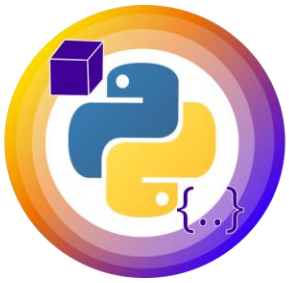


Python OOP: Glossary

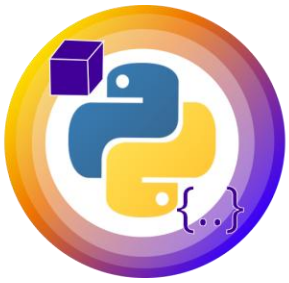
Glossary

Key Takeaways

- **Important Terminology:**
  - **OOP**: Object-Oriented Programming. A programming paradigm or approach used to analyze and solve problems that is based on the representation of real-world objects in the system.
  - "**Pythonic**": adjective commonly used to describe an approach to programming that follows the standards, rules, and best practice of the Python programming language.
  - **Class**: one of the building-blocks of Object-Oriented Programming that acts as a "blueprint" where the data and the actions of the objects are defined.
  - **Instance**: a concrete object that is created from the class "blueprint".
  - **Method**: an "action" defined in the class that the instances of the class can perform. It is very similar to a function, but closely related to instances such that instances can call methods and methods can act on the individual data of the instances.

Key Takeaways

- **Important Terminology:**
  - **Object:** a concrete instance of a class stored in memory.
  - **Docstrings**: (also Documentation Strings) they are strings located at the top of functions, classes, methods, or modules that describe their purpose, arguments, attributes, and relevant characteristics. They are "linked" to the element they describe by the __doc__ attribute.
  - **Mutation**: the result of changing the original object in memory instead of making a separate copy of the object.
  - **Aliasing**: when two or more names (variables) point to the same memory address.
  - **Cloning**: process of creating a separate copy of an object. This copy is initially equal to the original object, but it could change and the original object would not be affected.



Python OOP: Classes

Introduction to Classes

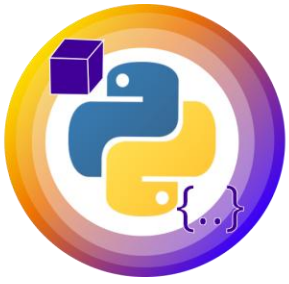
Key Takeaways

- **Classes**
  - They act like “blueprints” that describe the state and behavior of a type of real-world object or concept.
  - They are used to represent real-world objects or entities relevant to the context of a program or system. For example, houses, bank accounts, employees, clients, cars, products.
  - Main Elements:
    - ✓ Class Attributes
    - ✓ __init__()
    - ✓ Methods
  - Guidelines:
    - ✓ Class names are typically nouns. They should start with an uppercase letter. For example: **H**ouse, **H**uman, **D**og, **A**ccount.
    - ✓ If the name has more than one word, each word should be capitalized following the PascalCase naming convention. For example: **S**avings**A**ccount
    - ✓ The body of the class must be indented.

```
class <ClassName>(object):
```

Keyword

Optional parameter in Python 3



Python OOP: Instances and Instance Attributes

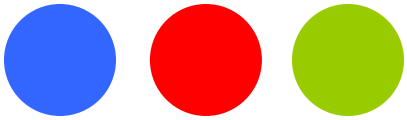
Instances

Key Takeaways

- **Instances**
  - They are **concrete** representations or example of the abstract objects that classes describe.
  - They are **created from a class** that acts like a "blueprint". Classes determine the attributes and functionality of their instances.
  - You can assign custom or predefined values for their attributes. These values are assigned in the constructor __init__(), a method that runs when an object is created.
  - They share the same "categories" of attributes, but the attributes can have different values. Changing the value of an attribute for one instance doesn't affect the other instances.

Classes act like "blueprints" to create instances

✓ For example: a class could have a "color" attribute. All the instances of that class would have this attribute, but the values can be different for each instance. One instance could have the value "blue" and another one the value "red".



Instances

Key Takeaways

- **General Syntax to Create an Instance**

```
<variable> = <ClassName>(<arguments>)
```

- **Example**

Class Name | Arguments

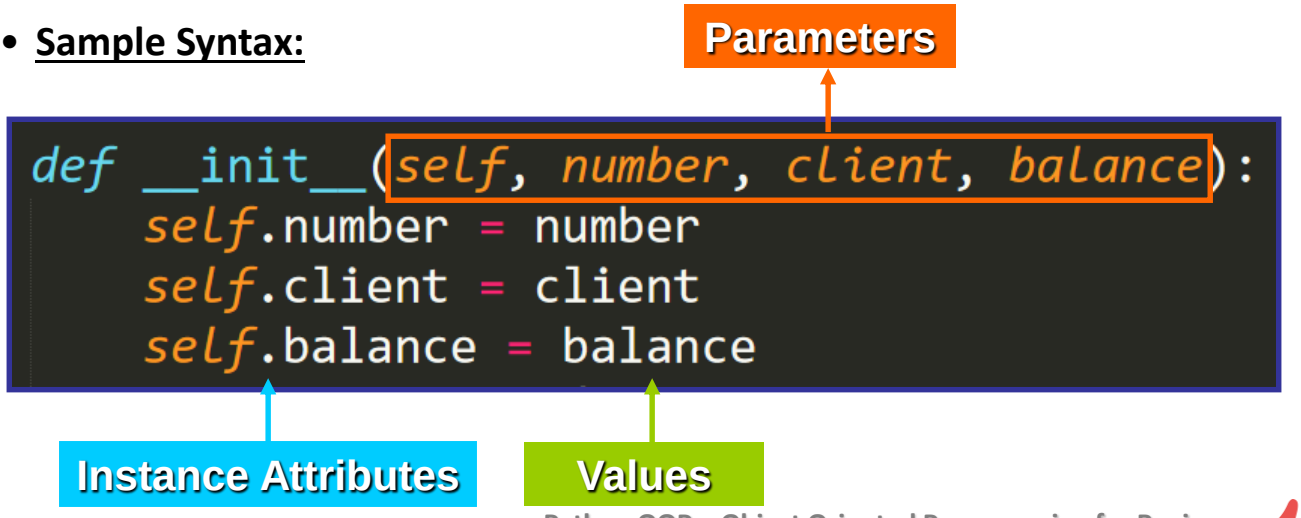
```
my_account = BankAccount("5621", "Gino Navone", 33424.4)
```

```
class BankAccount:

    accounts_created = 0

    def __init__(self, number, client, balance):
        self.number = number
        self.client = client
        self.balance = balance
        BankAccount.accounts_created += 1

    def display_balance(self):
        print(self.balance)
```

self is not included in the list of arguments

Key Takeaways

- **Constructor __init__():**
  - This is a reserved method,
  - Also called the **"constructor"** of the class.
  - It's called when an object (instance) is created.
  - Syntax:
    - ✓ Keyword "def".
    - ✓ __init__() with double underscore.
    - ✓ Within parentheses:
      - Parameter "self".
      - The formal parameters separated by a comma and a space after the comma.
    - ✓ A colon (:) at the end of the line.
- **Sample Syntax:**



Instance Attributes

Key Takeaways

- **Instance Attributes**
  - They **belong to the instances**. These attributes are relevant to the context of the program.
    - ✓ For example: bank accounts have an owner, a balance, and a number. These could be instance attributes in a BankAccount class.
  - Their **values are independent**. They are **not shared** across instances. Each instance has its own individual copy of the attribute.
  - To pass custom values for these attributes, you need to add them as parameters in the constructor __init__() and assign them using self.<attribute> = <value>
  - You can access and modify the values of these attributes after the instance has been created.
  - Changing the value of an attribute for one instance doesn't affect the value of the other instances of the class.
  - You can also set fixed values for these attributes when the object is created by assigning a value in __init__().

Instance Attributes

Key Takeaways

- **General Syntax to Assign a Value to an Instance Attribute**

```
self.<instance_attribute> = <value>
```

- **Example**

```
class BankAccount:

    accounts_created = 0

    def __init__(self, number, client, balance):
        self.number = number
        self.client = client
        self.balance = balance
        BankAccount.accounts_created

    def display_balance(self):
        print(self.balance)
```

Could be fixed values

```
self.balance = 10000
```

Instance Attributes

Key Takeaways

- **General Syntax to Get the Value of an Instance Attribute**

```
self.<instance_attribute>
```

- **Example**

```
class BankAccount:

    accounts_created = 0

    def __init__(self, number, client, balance):
        self.number = number
        self.client = client
        self.balance = balance
        BankAccount.accounts_created += 1

    def display_balance(self):
        print(self.balance)
```

We can get and use the current value

Instance Attributes

Key Takeaways

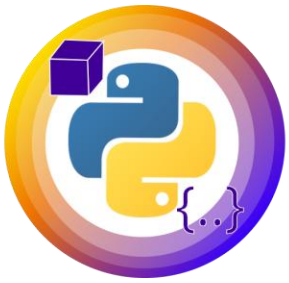
- **Instance Attributes – Default Values**
  - You can set default values for the parameters in the class constructor __init__() using <parameter>=<value>
  - If the arguments are omitted, the default arguments are used and they can be assigned to the instance attributes.
  - Warning: The parameters that have default arguments have to be located at the end of the list of parameters. Otherwise, an error will be thrown.

```
class BankAccount:

    accounts_created = 0

    def __init__(self, number, client, balance=1034.4):
        self.number = number
        self.client = client
        self.balance = balance
        BankAccount.accounts_created += 1

    def display_balance(self):
        print(self.balance)
```

Default value for balance

No argument = default value

```
myAccount = BankAccount("5621", "Gino Navone")
```

Python OOP: Class Attributes

Class Attributes

Key Takeaways

- **Class Attributes**
  - They **belong to the class** and all instances share the same class attribute. There is only one copy of the attribute.
    - ✓ For example: if we want our class to keep track of how many accounts have been created, the BankAccount class could have a accounts_created class attribute and all the instances of this class would access that same value.
  - The value of a class attribute is **shared across instances**. They all access the value from the same source, the class.
  - **Changing the value** of a class attribute **affects all instances**, since they take the value from the same source.
  - You can access and modify the values of class attributes.
  - The value of a class attribute can be accessed using the name of the class. No instance is required to access class attributes.

Class Attributes

Key Takeaways

- **General Syntax to Assign a Value to a Class Attribute Within the Class**

```
<class_attribute> = <value>
```

- **Example**

Shared across instances

```
class BankAccount:

    accounts_created = 0

    def __init__(self, number, client):
        self.number = number
        self.client = client
        self.balance = balance
        BankAccount.accounts_created += 1

    def display_balance(self):
        print(self.balance)
```

Class Attributes

Key Takeaways

- **General Syntax to Access the Value of a Class Attribute**

```
<ClassName>.<class_attribute>
```

- **Example**


```
class BankAccount:

    accounts_created = 0

    def __init__(self, number, client):
        self.number = number
        self.client = client
        self.balance = balance
        BankAccount.accounts_created += 1

    def display_balance(self):
        print(self.balance)
```

Class Attributes

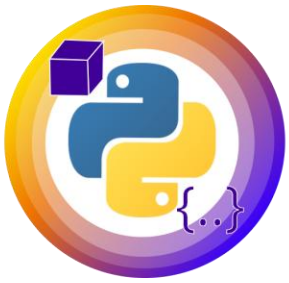
Key Takeaways

- **General Syntax to Modify the Value of a Class Attribute**

```
<ClassName>.<class_attribute> = <value>
```

- **Example**

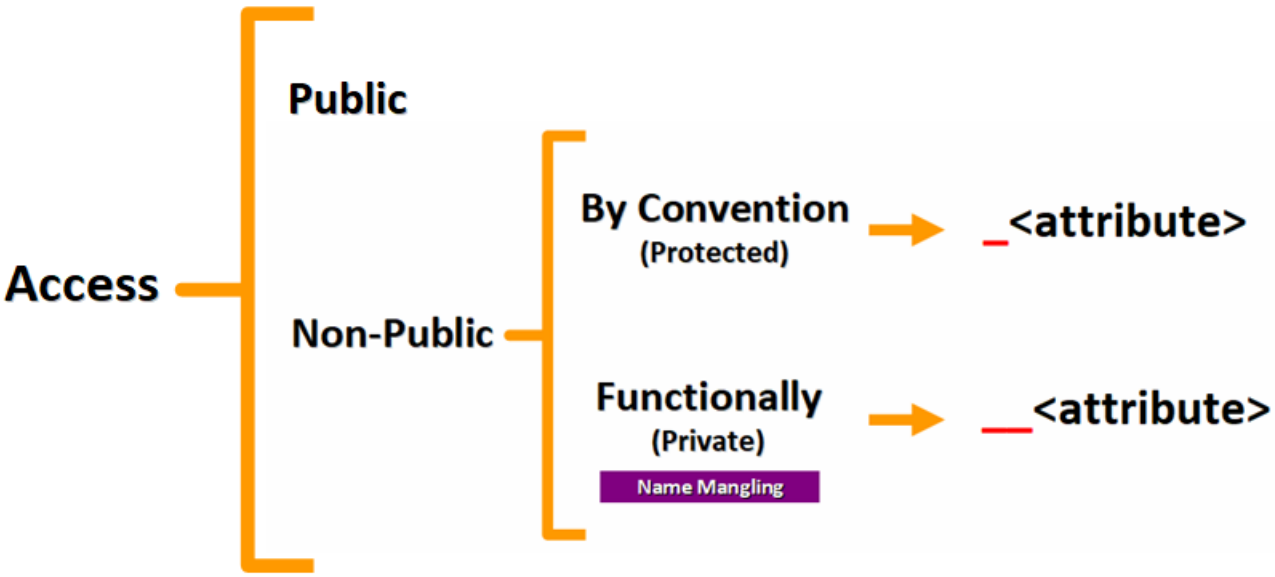

```
class BankAccount:

    accounts_created = 0

    def __init__(self, number, client):
        self.number = number
        self.client = client
        self.balance = balance
        BankAccount.accounts_created += 1

    def display_balance(self):
        print(self.balance)

BankAccount.accounts_created = 5
```



Python OOP: Encapsulation & Abstraction

Key Takeaways

- **Encapsulation**
  - "Bundling" of data and actions into a single unit (class).
  - Applied through the principle of information hiding.
  - You should restrict direct access to your data unless there is an important reasons not to do so.
  - To do this, you can define non-public attributes in Python.



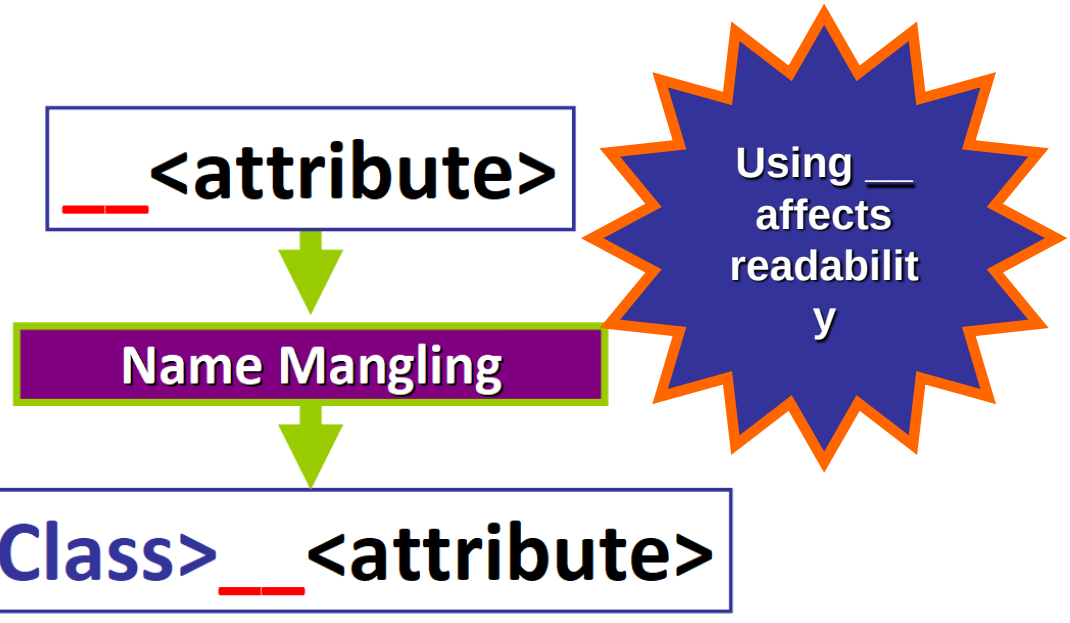
Note: The terms "private" and "protected" are symbolic in Python. No attribute is completely private.

Public vs. Non-Public

Key Takeaways

- **Public and Non-Public Attributes. Example:**

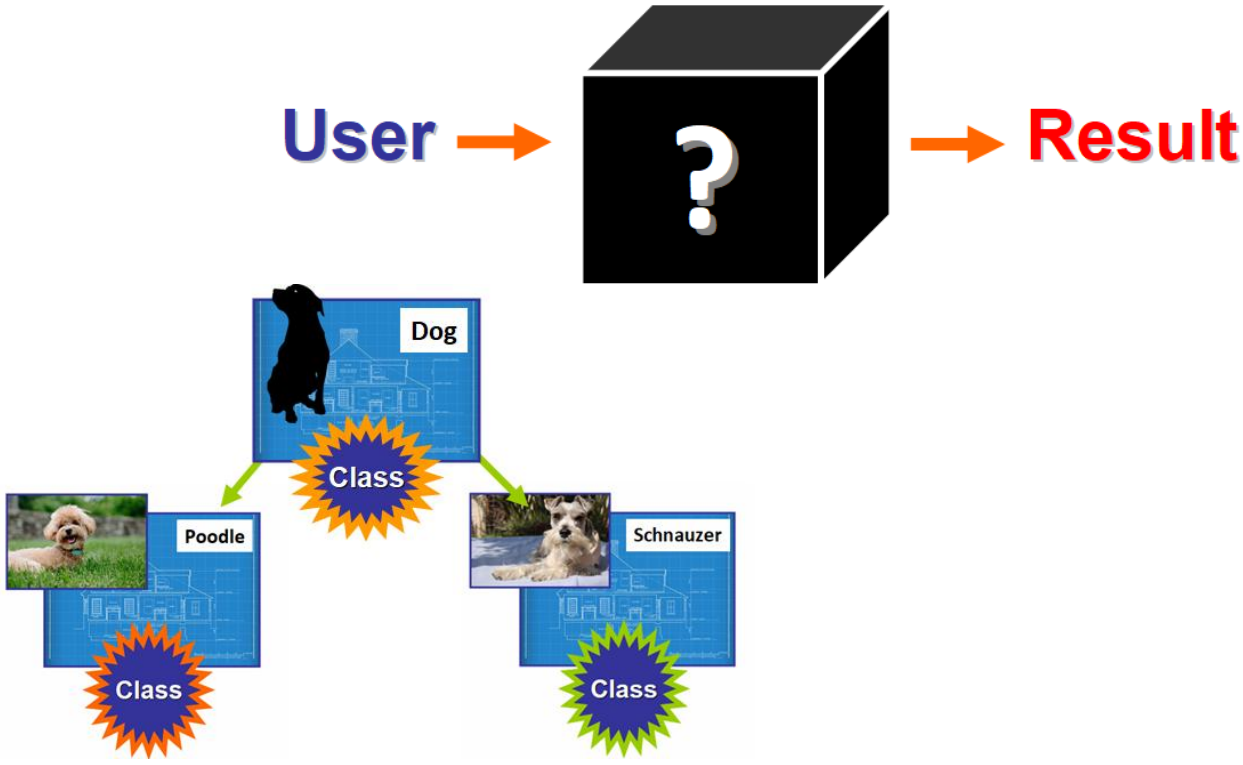
```
class Player:
    def __init__(self, username, rank, time_played):
        self.username = username
        self._rank = rank
        self.__time_played = time_played
```

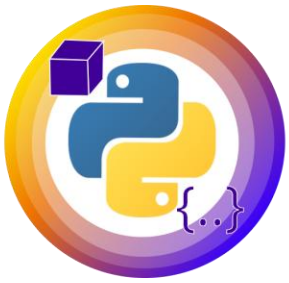


The **recommended way** to indicate that an attribute is "protected" and should not be accessed outside of the class, is to use a **single leading underscore**.

Key Takeaways

- **Abstraction**
  - Different facets and expressions:
    - The interface of a component should be independent of the implementation.
    - Relying on more general or “abstract” types of objects to avoid code repetition with the use of inheritance.

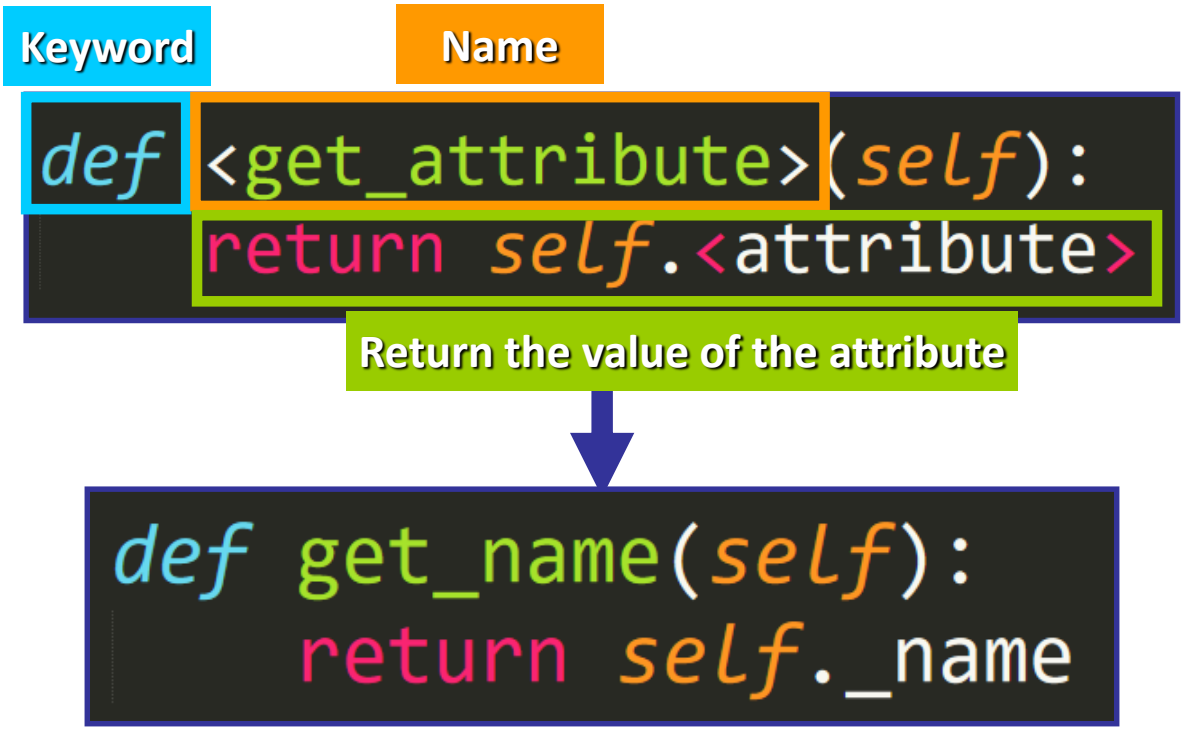




Python OOP: Getters, Setters, & Properties

Key Takeaways

- **Getters:**
  - Methods that instances can call to "**get**" the value of a protected instance attribute.
  - They serve as intermediaries to avoid accessing the data directly.
  - Naming Rules:
    - **get + _ + <attribute>**
    - Examples: get_age, get_name, get_code



Setters

Key Takeaways

- **Setters:**
  - Methods that instances can call to "**set**" the value of a protected instance attribute.
  - They serve as intermediaries to avoid accessing the data directly.
  - You can check if the value is valid before assigning it and you can react appropriately if the value is not valid.
  - They take one argument: the new value for the attribute.
  - Naming Rules:
    - **set + _ + <attribute>**
    - Examples: set_age, set_name, set_code

Keyword Name Parameter

```
def set_name(self, name):
    if isinstance(name, str):
        self._name = name
    else:
        print("Please enter a valid name")
```

Update the value

Key Takeaways

- **Properties:**
  - They are the “pythonic” way of working with getters and setters.
  - The property can be accessed with the same syntax used to access public instance attributes.
  - No need to call getters and setters explicitly, but they do act as intermediaries “behind the scenes”.
  - Two alternatives:
    - Using the built-in function **property()**.
    - Using the **@property** decorator.
  - @property is the recommended syntax to work with properties in Python.
    - Advantages:
      - ✓ More compact.
      - ✓ Improved readability.
      - ✓ No namespace pollution.

Properties

Key Takeaways

- **Using property():**

```
class Patient:

    def __init__(self, name, age, id_num, num_children=0):
        self.name = name
        self.age = age
        self._id_num = id_num
        self._num_children = num_children

    def get_id_num(self):
        print("Getter")
        return self._id_num

    def set_id_num(self, new_id):
        print("Setter")
        if isinstance(new_id, str):
            self._id_num = new_id
        else:
            print("Please enter a valid id")

    id_num = property(get_id_num, set_id_num)
```

Getter **Setter**

```
patient = Patient("Gino", 15, "4535")

patient.id_num # Calls getter

patient.id_num = "545435" # Calls setter
```

Properties

Key Takeaways

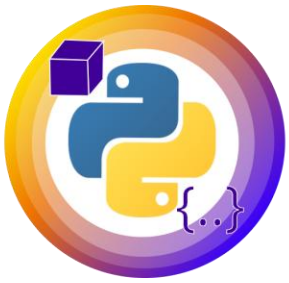
- **Using @property:**

```
class House:

    def __init__(self, price):
        self._price = price

    @property
    def price(self):
        return self._price

    @price.setter
    def price(self, new_price):
        if price > 0:
            self._price = _price
        else:
            print("Please enter a valid price")
```

Getter

Setter

```
house = House(50000)

house.price # Calls getter

house.price = 60000 # Call setter
```

Python OOP: Methods

Key Takeaways

- **Methods**
  - They represent the **actions** that the instances of a class can perform. These actions are relevant to the context of the program.
    - ✓ For example: a BankAccount class could have a display_balance method. It could also have a deposit method to update the current balance.
  - You can think of methods as functions that belong to classes and that instances have special access to.
  - Methods are called by instances.
  - The first parameter of a method is *self*, which refers to the instance that calls the method.
  - You can use methods to modify or create instance attributes and to modify the value of class attributes.
  - Conventions:
    - ✓ Method names contain a verb because they represent actions.
      - ♦ For example: **display**_name, **make**_transfer, **move**_forward.

Methods

Key Takeaways

- **General Syntax**

Keyword

Separated by commas

```
def <method_name>(self, <params>):
    # Code
```

- **Example**

```
class BankAccount:

    accounts_created = 0

    def __init__(self, number, client):
        self.number = number
        self.client = client
        self.balance = balance
        BankAccount.accounts_creat

    def display_balance(self):
        print(self.balance)
```

Shared by all instance s

```
def display_balance(self):
    print(self.balance)
```

Methods

Key Takeaways

- **General Syntax to Call a Method**

"Skipping" self

```
<instance>.<method>(<arguments>)
```

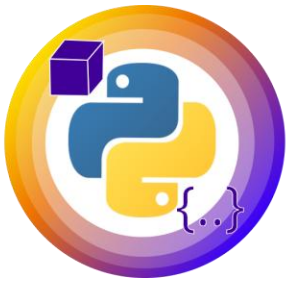
Separated by Commas

- **Example**

```
class BankAccount:

    accounts_created = 0

    def __init__(self, number, client):
        self.number = number
        self.client = client
        self.balance = balance
        BankAccount.accounts_created += 1

    def display_balance(self):
        print(self.balance)

my_account = BankAccount("5621", "Gino Navone", 2343.32)

my_account.display_balance()
```

'self' refers to the instance that calls the method

If the method has only one parameter (self) use an empty set of parentheses to call it

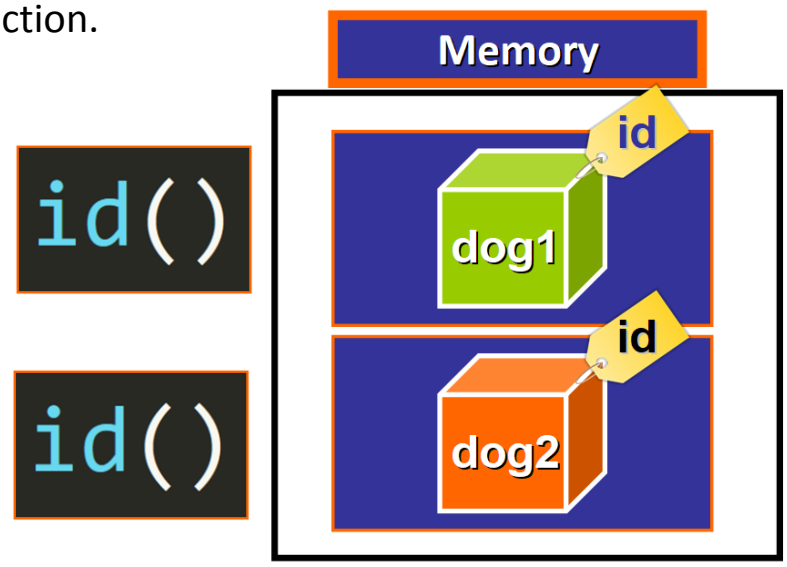


Python OOP: Objects in Memory

Objects in Memory

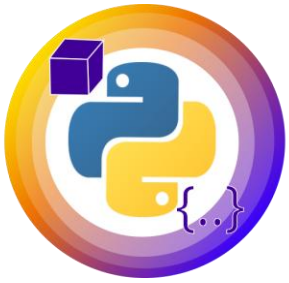
Key Takeaways

- **Objects in Memory:**
  - "Everything in Python is an Object."
  - Objects are stored in memory.
  - Each object has a **unique number** assigned that represents the memory address where it is currently stored. This number is called the "id" of the object.
  - You can get this value using the **id()** function.
    - Syntax: id(<object_var>)
  - Objects are passed **by reference** to avoid making copies of the objects every time that you pass them as arguments to a function.



Key Takeaways

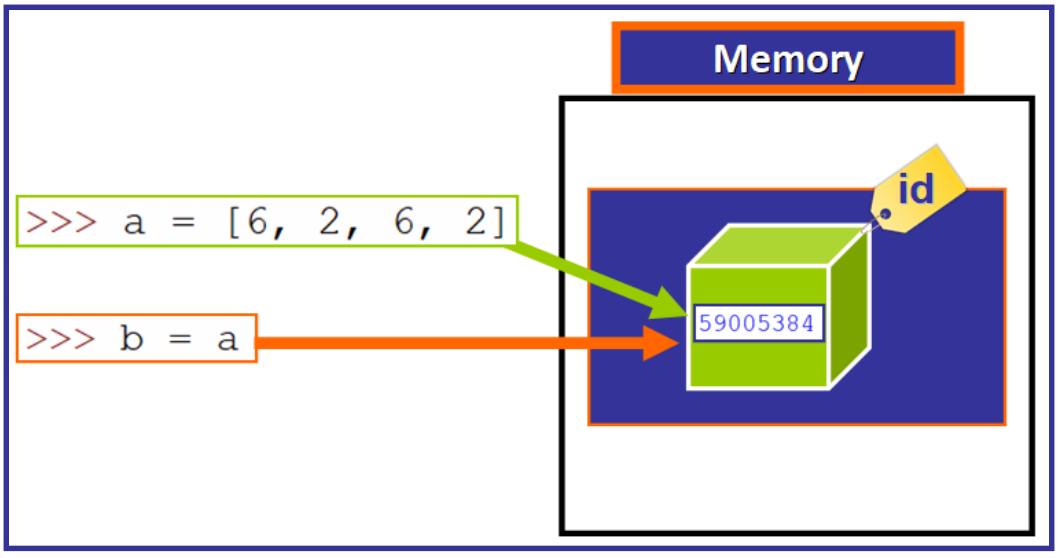
- **The "is" operator:**
  - Returns **True** when the two operands **point to the same object** in memory.
  - Returns **False** if the operands **point to different objects** in memory.
  - There are **a few exceptions** to optimize memory usage:
    - Small integers in a range from [-5, 256] are retrieved from the same existing object to avoid creating new objects whenever you use an integer.
    - Certain strings are represented with the same object in memory to avoid creating several different objects to represent the same string.



Python OOP:
Aliasing, Mutation, & Cloning

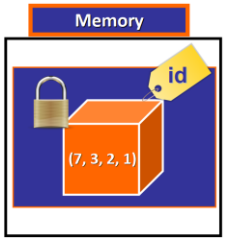
Key Takeaways

- **Aliasing:**
  - Occurs when **two or more variables point to the same object** in memory.
  - These variables that point to the same object are called “aliases.”
  - Aliases have the **same id** value when you call the id() function.
  - **Warning:** Aliases can be a source of bugs because you can modify an object using one the aliases and forget that other names point to the same object, so they will be modified too.

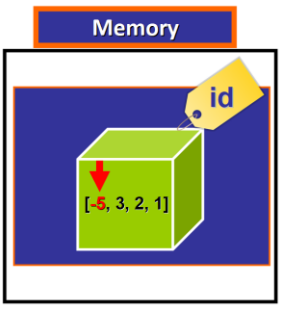


Key Takeaways

- **Mutation:**
  - There are two types of objects:
    - **Mutable**: **can** change after they've been defined.
    - **Immutable**: **can't** change after they've been defined.
  - Built-in Mutable Objects:
    - Lists, Sets, Dictionaries, more…

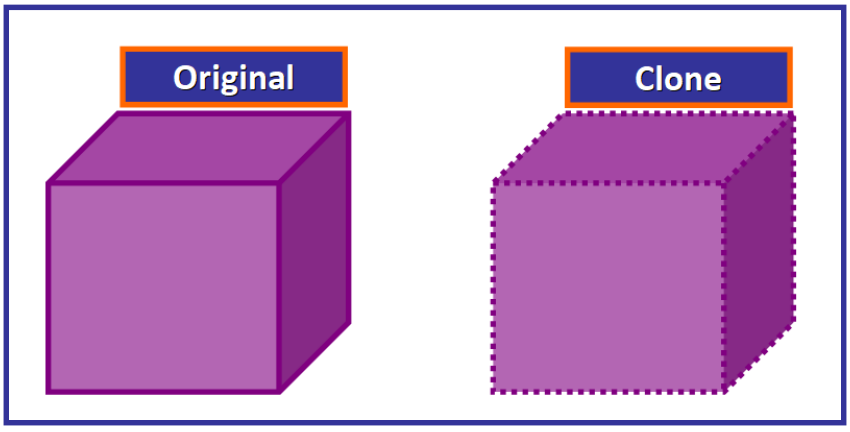


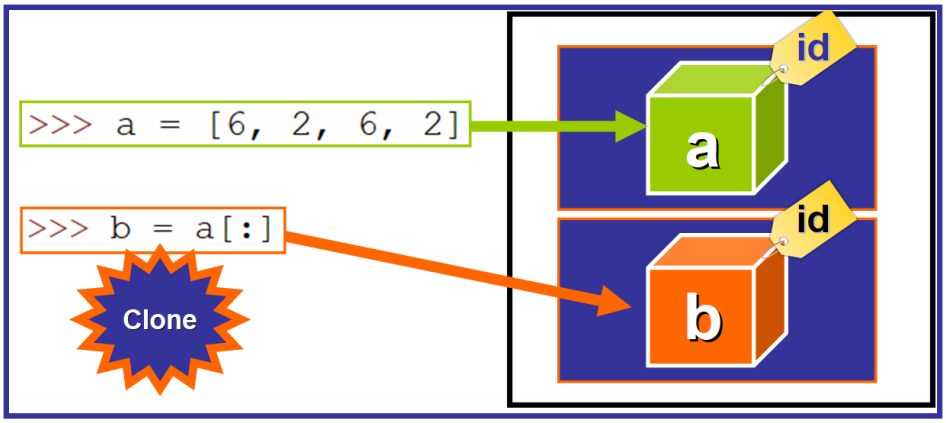
  - Built-in Immutable Objects:
    - Tuples, Strings, Floats, Integers, Booleans, more…
  - **Advantages:**
    - Mutable: can be modified easily without additional memory usage.
    - Immutable: safer from bugs and easier to understand.
  - **Disadvantages:**
    - Mutable: more prone to bugs due to aliasing.
    - Immutable: additional memory usage because a new object has to be created when you need a modified copy of the original.

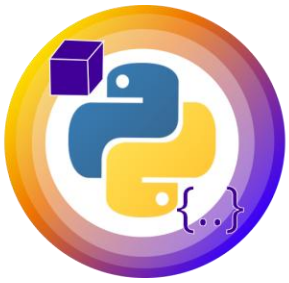


Key Takeaways

- **Cloning:**
  - It is making a copy of the object in memory.
  - This copy has the same value of the original object, but it is a **different object in memory** (not an alias).
  - This way, you can modify the clone without affecting the original object.







Python OOP: Inheritance
(Attributes)

Inheritance (Attributes)

Key Takeaways

- **Inheritance (Attributes)**
  - Take advantage of natural hierarchies between objects and concepts by creating classes that "inherit" attributes and behaviors from other classes.
    - ✓ For example: a SavingsAccount is a type of account, so it could inherit the attributes and functionality of the Account class.
  - Advantages:
    - ✓ Reuse existing code.
    - ✓ Write more maintainable and scalable code.
    - ✓ Improve project structure and design.
  - Key Concepts:
    - Parent class (superclass): the class from which another class inherits attributes and behaviors.
    - Child class (subclass): the class that inherits attributes and behaviors from another class.
  - You can create multilevel hierarchies. Classes can inherit from classes that inherit from classes that inherit from classes and so on…

Key Takeaways

- **General Syntax (First Step – Parent & Child)**

```
class <Superclass>:
    # Body

class <Subclass>(<Superclass>):
    # Body
```

- **Example**

```
class Account:
    # Body

class SavingsAccount(Account):
    # Body
```

Inheritance (Attributes)

Key Takeaways

- **General Syntax (Second Step - Attributes)**

```
class <Superclass>:
    # Body

class <Subclass>(<Superclass>):

    def __init__(self, <parameters>):
        <Superclass>.__init__(self, <arguments_for_parent_class>)
        # Other instance attributes

    # Methods
```

Attributes are automatically inherited if you don't define __init__() in the subclass

- **Example**

```
class Account:

    accounts_created = 0

    def __init__(self, number, client, balance):
        self.number = number
        self.client = client
        self.balance = balance
        accounts_created += 1

    def display_balance(self):
        print(self.balance)

class SavingsAccount(Account):

        def __init__(self, number, client, balance, interest_rate):
                Account.__init__(self, number, client, balance)
                self.interest_rate = interest_rate

        def display_interest_rate(self):
                print(self.interest_rate)
```

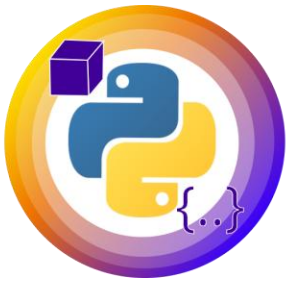
Inheritance (Attributes)

Key Takeaways

- **Example in More Detail**

```python
class Account:

    accounts_created = 0

    def __init__(self, number, client, balance):
        self.number = number
        self.client = client
        self.balance = balance
        accounts_created += 1

    def display_balance(self):
        print(self.balance)

class SavingsAccount(Account):

        def __init__(self, number, client, balance, interest_rate):
                Account.__init__(self, number, client, balance)
                self.interest_rate = interest_rate

        def display_interest_rate(self):
                print(self.interest_rate)
```

SavingsAccount inherits from Account



Python OOP: Inheritance
(Methods)

Inheritance (Methods)

Key Takeaways

- **Inheritance (Methods)**
  - Take advantage of natural hierarchies between objects and concepts by creating classes that "inherit" functionality and behaviors from other classes.
    - ✓ For example: a SavingsAccount is a type of account, so it could inherit the functionality of the Account class.
  - All instances of a subclass have access to the methods of the superclass. They can call them, passing the corresponding arguments.
  - *self* will still refer to the instance that calls the method, even if the method belongs to the parent class.
  - When an instance calls a method, the first method that is found in the hierarchy with that name is executed.
  - Method Overriding:
    - Occurs when there are two methods with the same name, one in the superclass and another one in the subclass.
    - The method in the subclass is executed.
    - You can explicitly call the method from the parent class using <parent_class>.<method>(<arguments>)

Key Takeaways

- **General Syntax (First Step – Parent & Child)**

```
class <Superclass>:
    # Body

class <Subclass>(<Superclass>):
    # Body
```

- **Example**

```
class Account:
    # Body

class SavingsAccount(Account):
    # Body
```

Inheritance (Methods)

Key Takeaways

- **General Syntax (Second Step - Methods)**

```
class Account:

    accounts_created = 0

    def __init__(self, number, client, balance):
        self.number = number
        self.client = client
        self.balance = balance
        Account.accounts_created += 1

    def display_balance(self):
        print(self.balance)

class SavingsAccount(Account):

        def __init__(self, number, clie
                Account.__init__(self, numb
                self.interest_rate = interest_

        def display_interest_rate(self):
                print(self.interest_rate)

my_savings_account = SavingsAccount("5621", "G  Navone", 452.34, 0.02)

my_savings_account.display_balance()
```

An instance of the subclass calls a method of the superclass

```
>>> my_savings_account = SavingsAccount("5621", "Gino Navone", 452.34, 0.02)
>>> my_savings_account.display_balance()
452.34
```

Method Call