CS112: Introduction to Python programming

Week 10: Class II & Numpy & Scipy I

Upcoming schedule

- Assignment 4: function

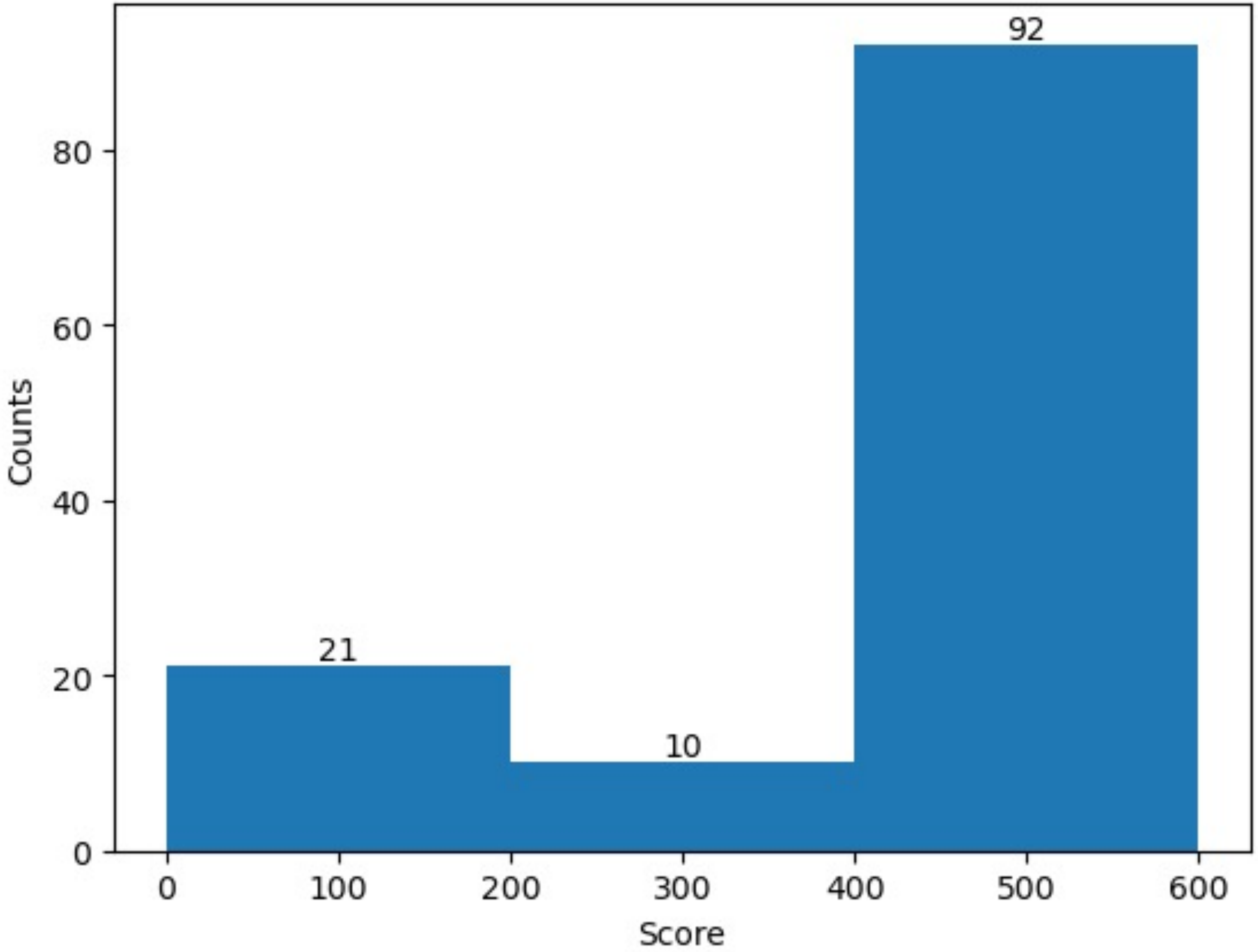
Deadline: Week 11 Friday before class

Code sharing sign-up: today

Upcoming schedule

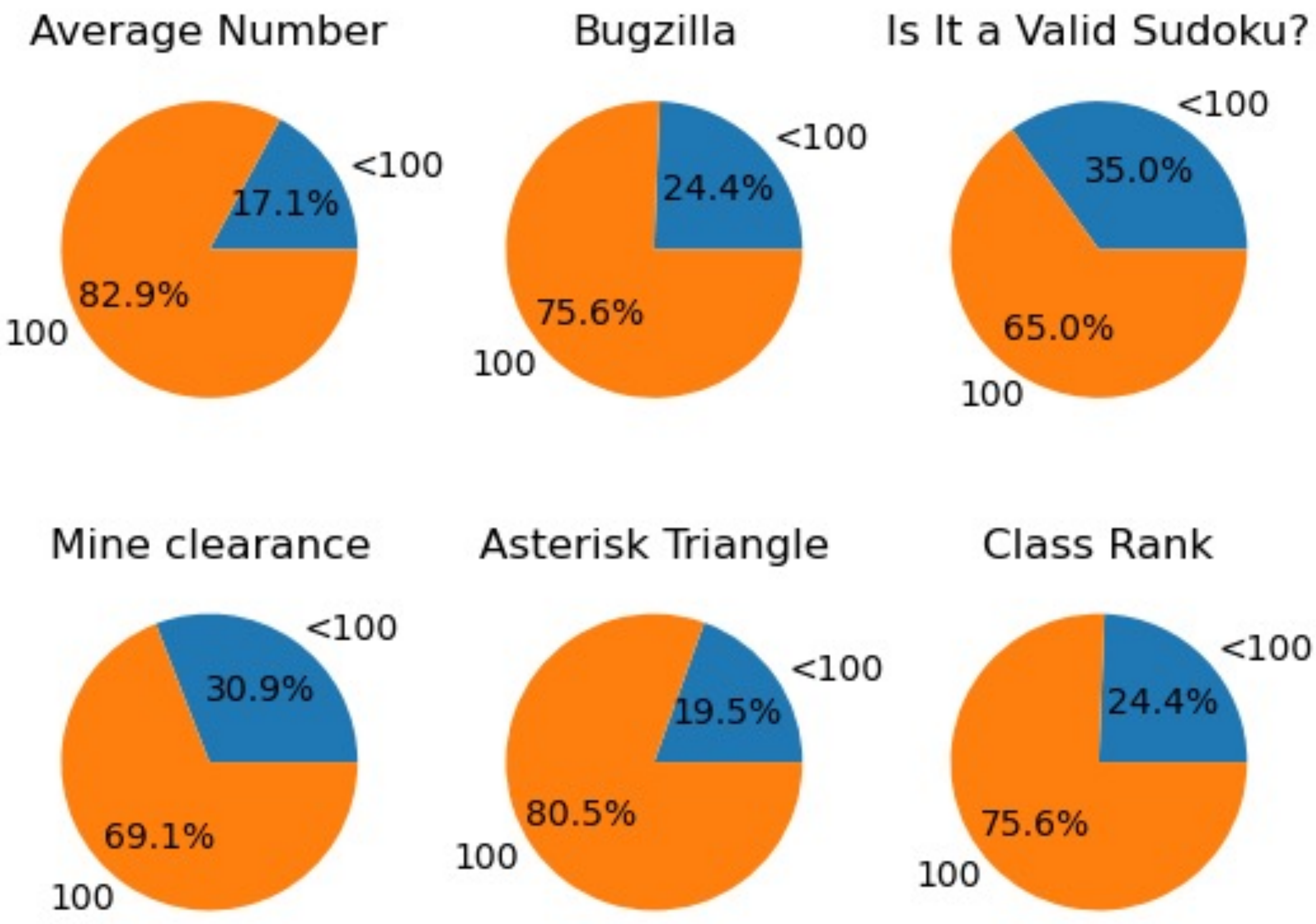
Week	Content	Assignment / Quiz
Week 9	Class I	Quiz 2, Assignment 4
Week 10	Class II & Numpy & Scipy I	
Week 11	Numpy & Scipy II	Assignment 5
Week 12	Pandas I	
Week 13	Pandas II	Assignment 6
Week 14	Data Visualization I	Quiz 3
Week 15	Data Visualization II	
Week 16	Basic statistics in Python & Clustering	

Assignment 3 statistics

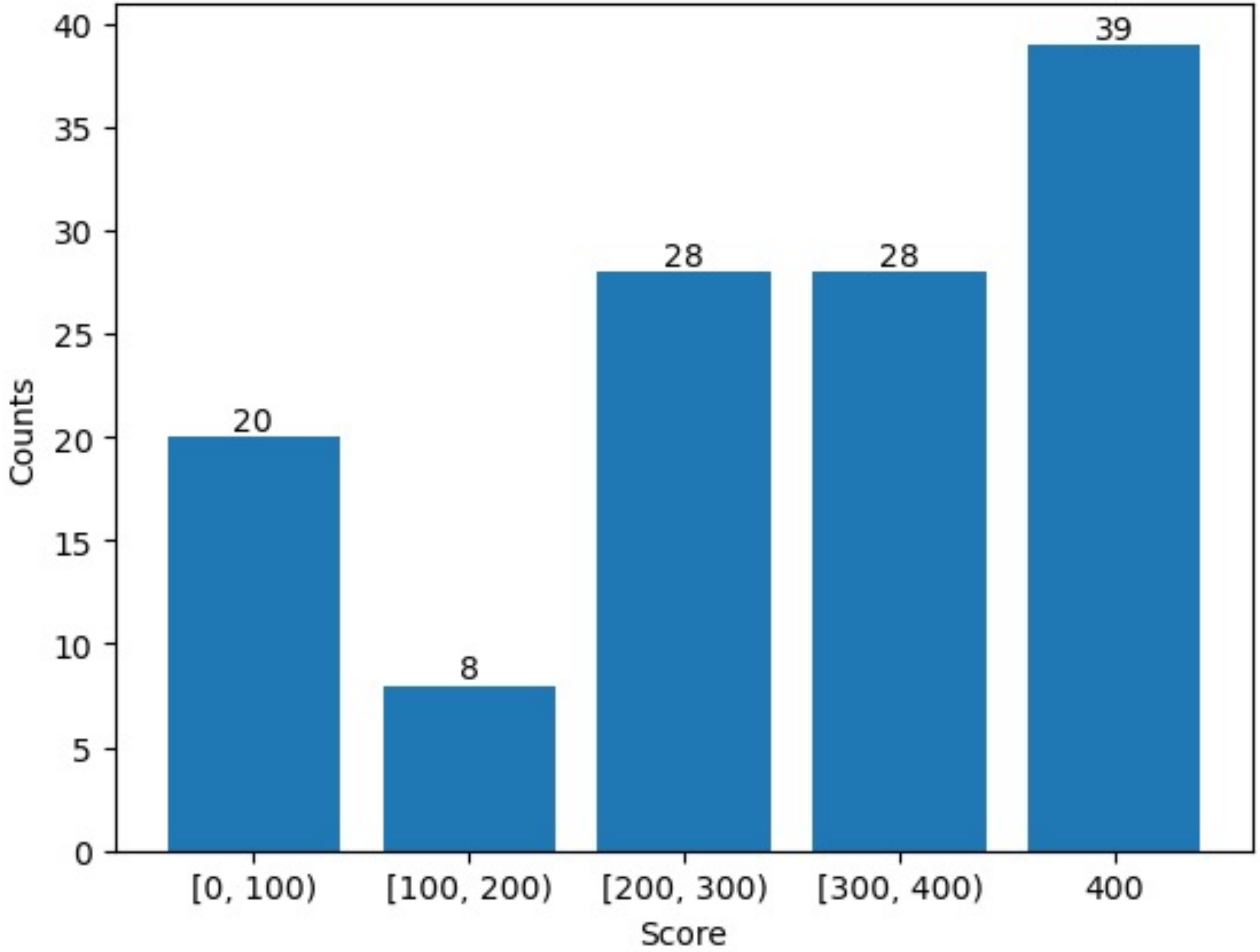


count	123.00
mean	461.62
std	215.51
min	0.00
25%	390.00
50%	600.00
75%	600.00
max	600.00

Assignment 3 statistics

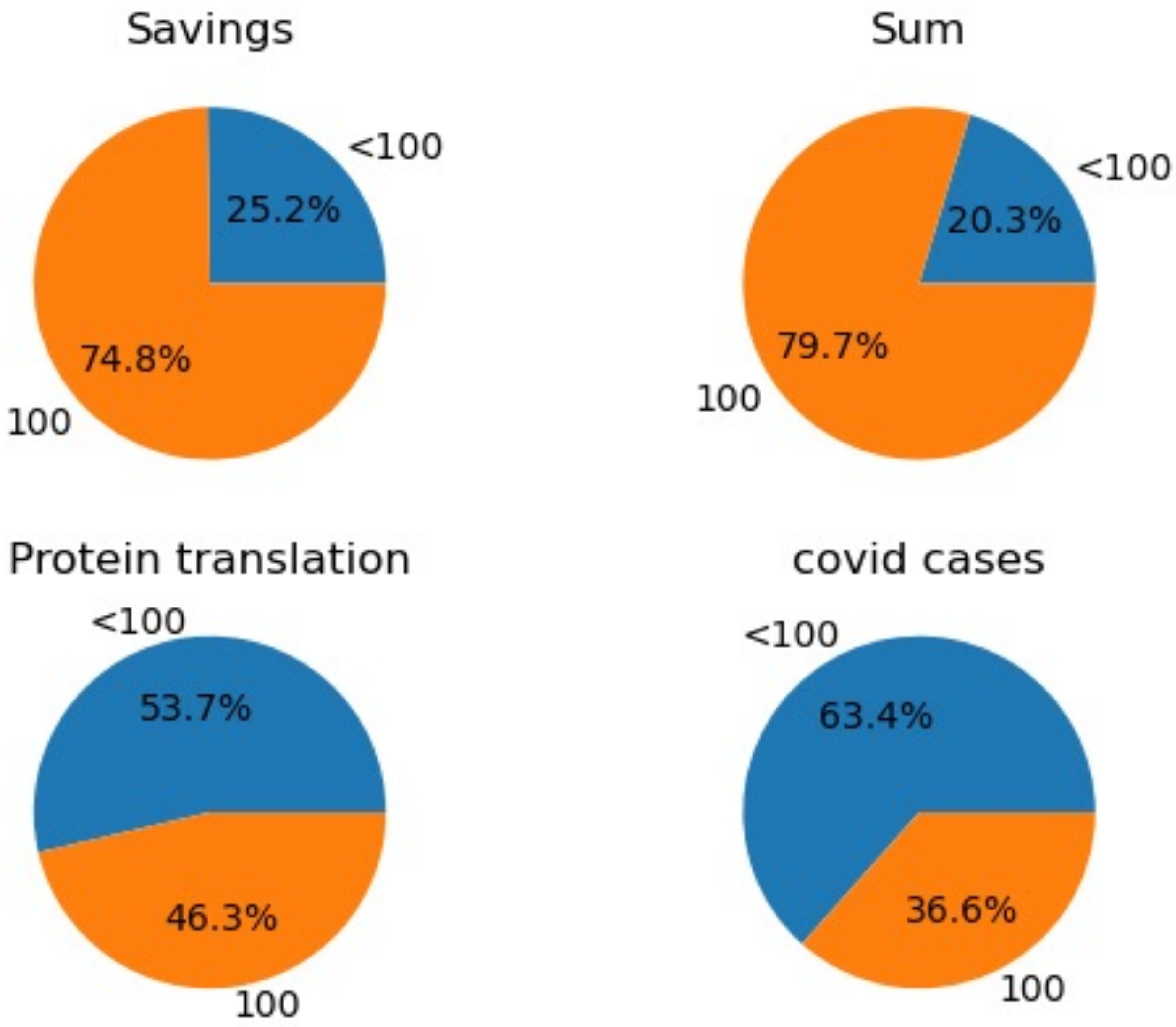


Quiz 2 statistics



count	123.00
mean	244.30
std	139.99
min	0.00
25%	175.00
50%	300.00
75%	400.00
max	400.00

Quiz 2 statistics



Object-oriented programming (OOP)

- **Object-oriented programming (OOP)（面向对象编程）** is a programming paradigm based on the concept of "objects", which can contain data and code. The data is in the form of fields (often known as **attributes** or **properties**), and the code is in the form of procedures (often known as **methods**).

- Python is an OOP language. Almost everything in Python is an object, with its properties and methods.

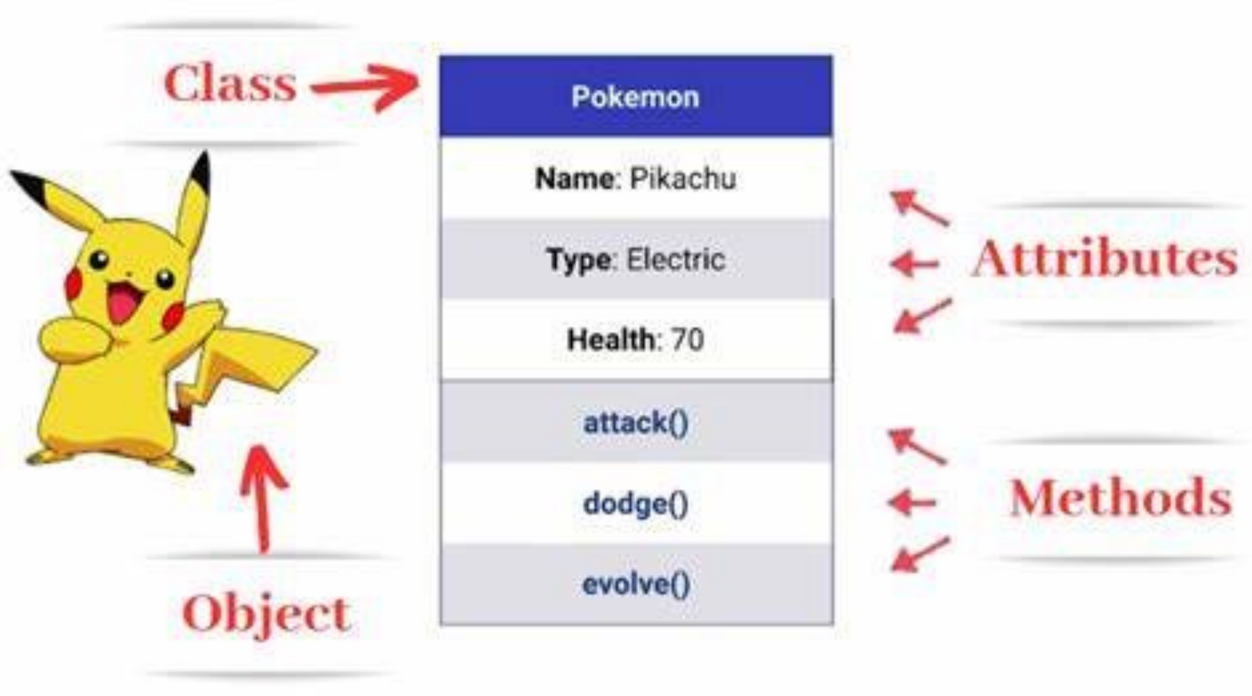
Class

Class(类): definition of a particular kind of object, including its features and how it is implemented in code; a template that is used to generate objects;

- Objects can contain data and have associated methods.
- A class can be a data type such as ***string*** or ***set***, but also something more complex like ***genome***, ***people***, ***sequences***, etc. Any object capable of being abstracted can be a class.

Object(对象): A specific *instance(实例)* of the class

Class




```python
name = str('John')
```

```python
name.lower()
```

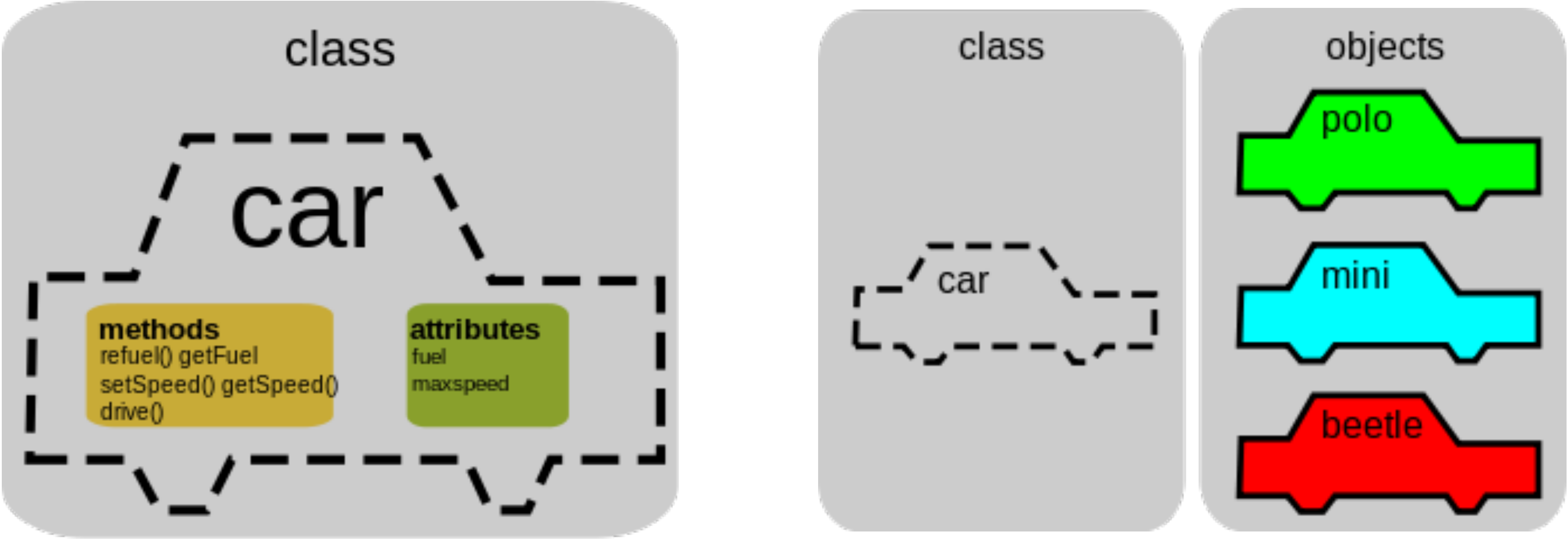
```
'john'
```

Python is an object-oriented language.

Attributes(属性) are variables associated with all the objects of a class. Whenever an object is created from a class, this object inherits the variable of the class.

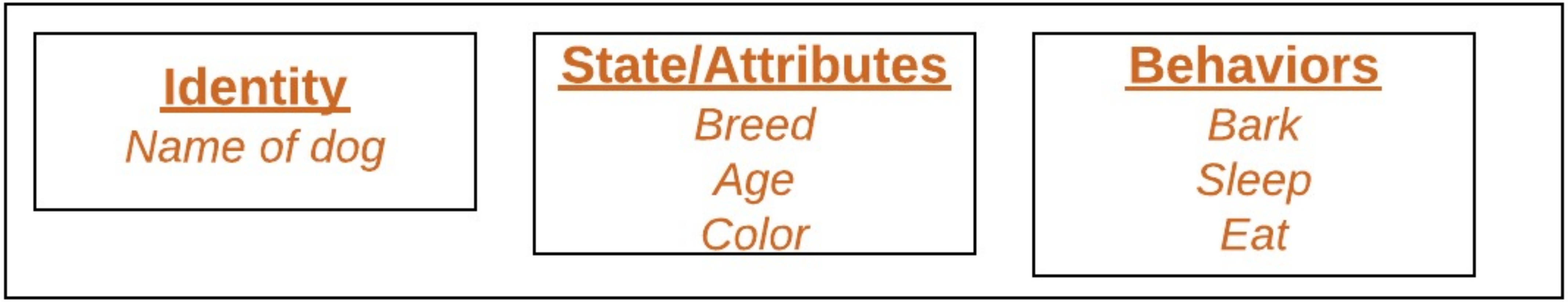
Methods(方法) are functions associated with an object.

Class



A Class is like an object constructor, or a "blueprint" for creating objects.
An Object is an instance of a Class. An Object is a copy of the class with *actual values*.

Object

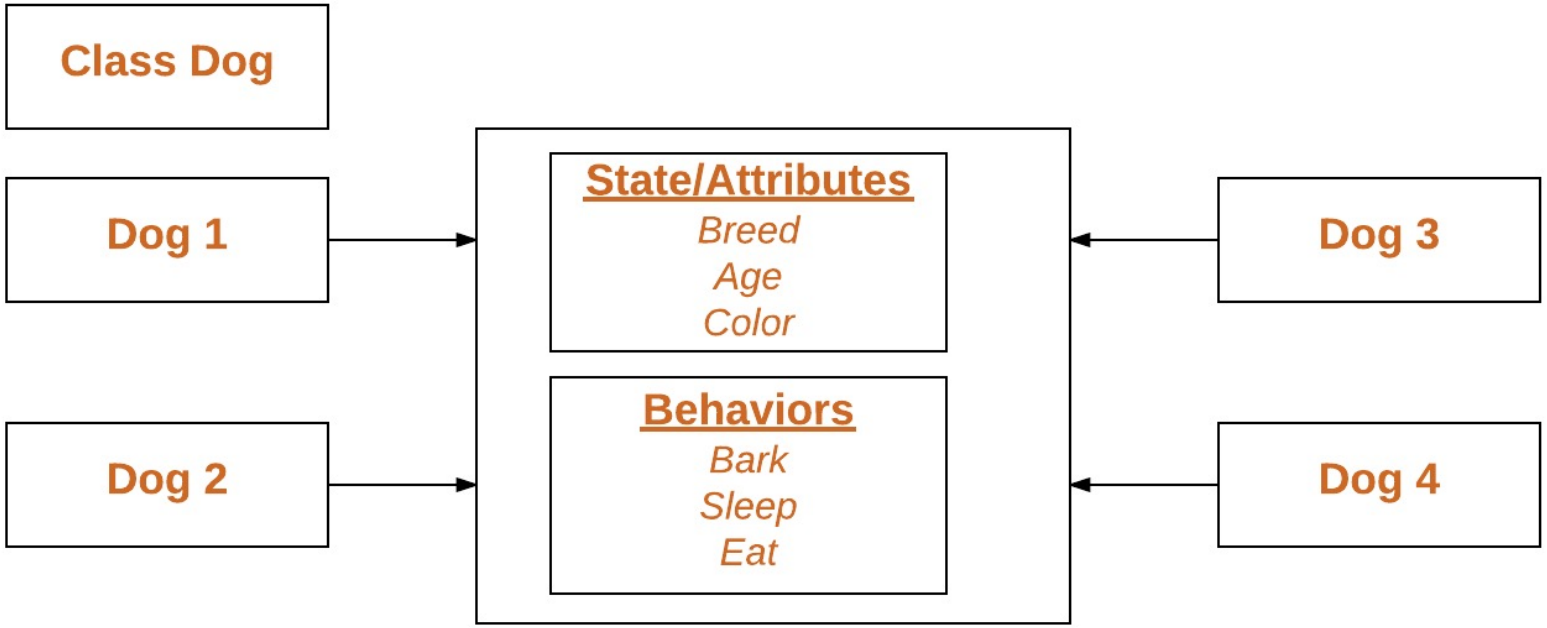


An object consists of :

- Identity: It gives a unique name to an object and enables one object to interact with other objects.
- States/Attributes: It reflects the properties of an object.
- Behaviors: It is represented by the methods of an object. It also reflects the response of an object to other objects.

https://www.geeksforgeeks.org/python-classes-and-objects/

Object



Declaring Objects (Also called instantiating （实例化） a class)
All the objects share the attributes and the behavior of the class. But the values of those attributes are unique for each object.
A single class may have any number of objects.

Creating classes

Classes are the template of the objects. The syntax to create classes in Python is very simple:

```
class NAME:
    [body]
```

Self is a variable that is used to represent the instance of the Class

```python
class Car:
    def __init__(self):
        self.wheels = 4
        self.miles = 0
```

Methods

Attributes

init is a special method that doesn't return any value. It is excuted whenever an instance of the Class is created. It is used to customize a specific initial state.

The `__init__()` function

- All classes have a function called __init__(), which is always executed when the class is being initiated.
- Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

```
class Car:
    def __init__(self):
        self.wheels = 4
        self.miles = 0
```

```
tesla = Car()
```

```
tesla.wheels
```

```
4
```

```
tesla.miles
```

```
0
```

We can access the instance attributes and methods using the object and dot (.) operator.

The `self` parameter

- Class methods must have an extra first parameter (usually named as self) in the method definition.

- We do not give a value for this parameter when we call the method, Python provides it.

- If we have a method that takes no arguments, we still have one argument.
- When we call a method of the object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2)

The **self** parameter

It does not have to be named self, you can call it whatever you like, but it has to be the first parameter of any function in the class.

```
class Car:
    def __init__(self):
        self.wheels = 4
        self.miles = 0
```

```
tesla = Car()
print(tesla.wheels)
```

```
4
```

```
class Car:
    def __init__(aa):
        aa.wheels = 4
        aa.miles = 0
```

```
tesla = Car()
print(tesla.wheels)
```

```
4
```

Classes with arguments

```
class Car:
    def __init__(self,initial_mileage):
        self.wheels = 4
        self.miles = initial_mileage
```

```
tesla = Car()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-58-be8e7b549287> in <module>
----> 1 tesla = Car()

TypeError: __init__() missing 1 required positional argument: 'initial_mileage'
```

Classes with arguments

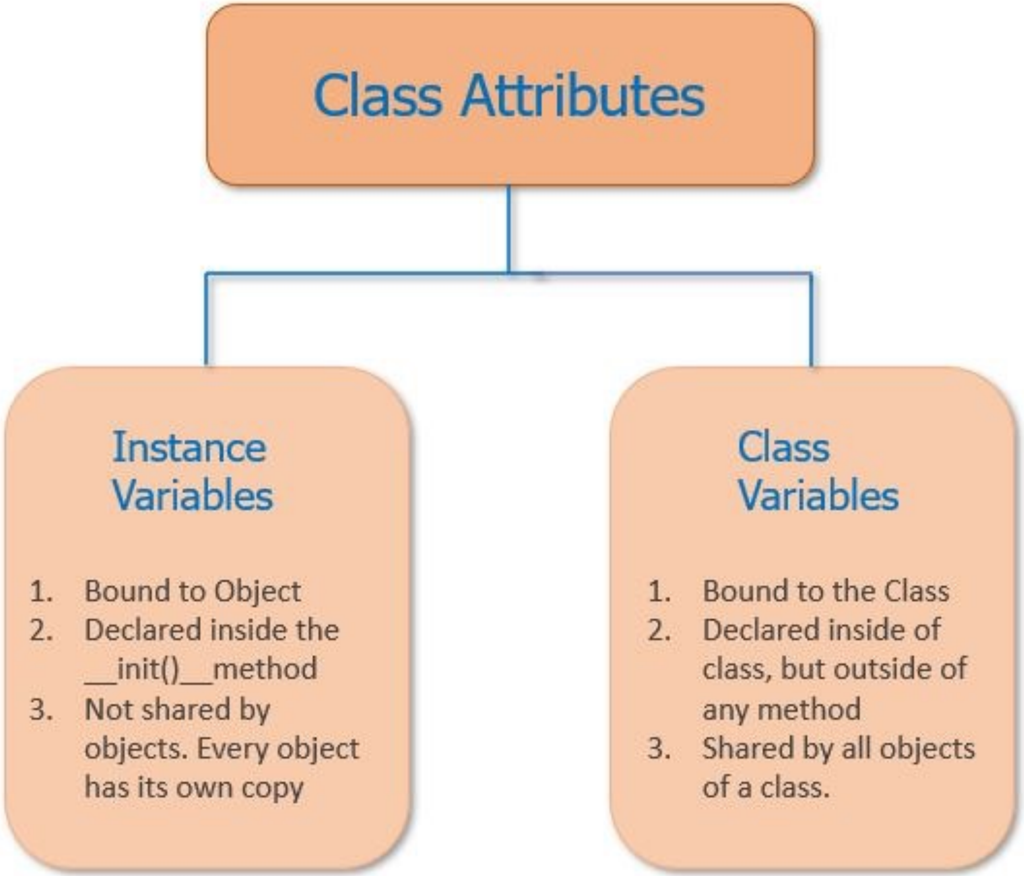
```
class Car:
    def __init__(self,initial_mileage):
        self.wheels = 4
        self.miles = initial_mileage
```

```
tesla = Car(initial_mileage = 100)
```

```
tesla.miles
```

```
100
```

Instance and Class variables



In Class, attributes can be defined into two parts:

- Instance variables: The instance variables are attributes attached to an instance of a class. We define instance variables in the constructor (the __init__() method of a class).

- Class Variables: A class variable is a variable that is declared inside of class, but outside of any instance method or __init__() method.

Instance variables

```python
class Car:
    def __init__(self, m):
        # instance variable
        self.wheels = 4
        self.miles = m
```

```python
tesla = Car(100)
print(tesla.wheels)
print(tesla.miles)
```

```
4
100
```

```python
toyota = Car(500)
print(toyota.wheels)
print(toyota.miles)
```

```
4
500
```

It is possible to change the value of the attribute of an instance

```python
tesla = Car(100)
print(tesla.wheels)
tesla.wheels = 6
print(tesla.wheels)
```

```
4
6
```

This change is specific for the instance. When new instances are created, the method __init__ is executed again to assign the attribute to the new instance

Instance variables

```python
class Car:
    def __init__(self, m):
        # instance variable
        self.wheels = 4
        self.miles = m
```

```python
tesla = Car(100)
print(tesla.wheels)
print(tesla.miles)
```

```
4
100
```

```python
toyota = Car(500)
print(toyota.wheels)
print(toyota.miles)
```

```
4
500
```

It is possible to change the value of the attribute of an instance

```python
tesla = Car(100)
print(tesla.wheels)
tesla.wheels = 6
print(tesla.wheels)
```

```
4
6
```

```python
toyota = Car(500)
print(toyota.wheels)
```

```
4
```

This change is specific for the instance. When new instances are created, the method __init__ is executed again to assign the attribute to the new instance

Access instance variables

```python
class Car:
    def __init__(self, m):
        # instance variable
        self.wheels = 4
        self.miles = m
```

```python
tesla = Car(100)
print(tesla.wheels)
print(tesla.miles)
```

```
4
100
```

```python
toyota = Car(500)
print(toyota.wheels)
print(toyota.miles)
```

```
4
500
```

getattr(object, name[, default])

Get a named attribute from an object; getattr(x, 'y') is equivalent to x.y.

```python
print(getattr(tesla, 'wheels'))
print(getattr(tesla, 'miles'))
```

```
4
100
```

Dynamically add instance variables

```python
class Car:
    def __init__(self, m):
        # instance variable
        self.wheels = 4
        self.miles = m
```

```python
tesla = Car(100)
print(tesla.wheels)
print(tesla.miles)
```

```
4
100
```

```python
toyota = Car(500)
print(toyota.wheels)
print(toyota.miles)
```

```
4
500
```

```python
tesla.value = 200
```

```python
print(tesla.value)
```

```
200
```

```python
toyota.value
```

```
---------------------------------------------------------------------------
--
AttributeError                            Traceback (most rec
t)
Cell In[19], line 1
----> 1 toyota.value

AttributeError: 'Car' object has no attribute 'value'
```

Dynamically delete instance variables

```
class Car:
    def __init__(self, m):
        # instance variable
        self.wheels = 4
        self.miles = m
```

```
tesla = Car(100)
print(tesla.wheels)
print(tesla.miles)
```

```
4
100
```

```
toyota = Car(500)
print(toyota.wheels)
print(toyota.miles)
```

```
4
500
```

```
del tesla.miles
```

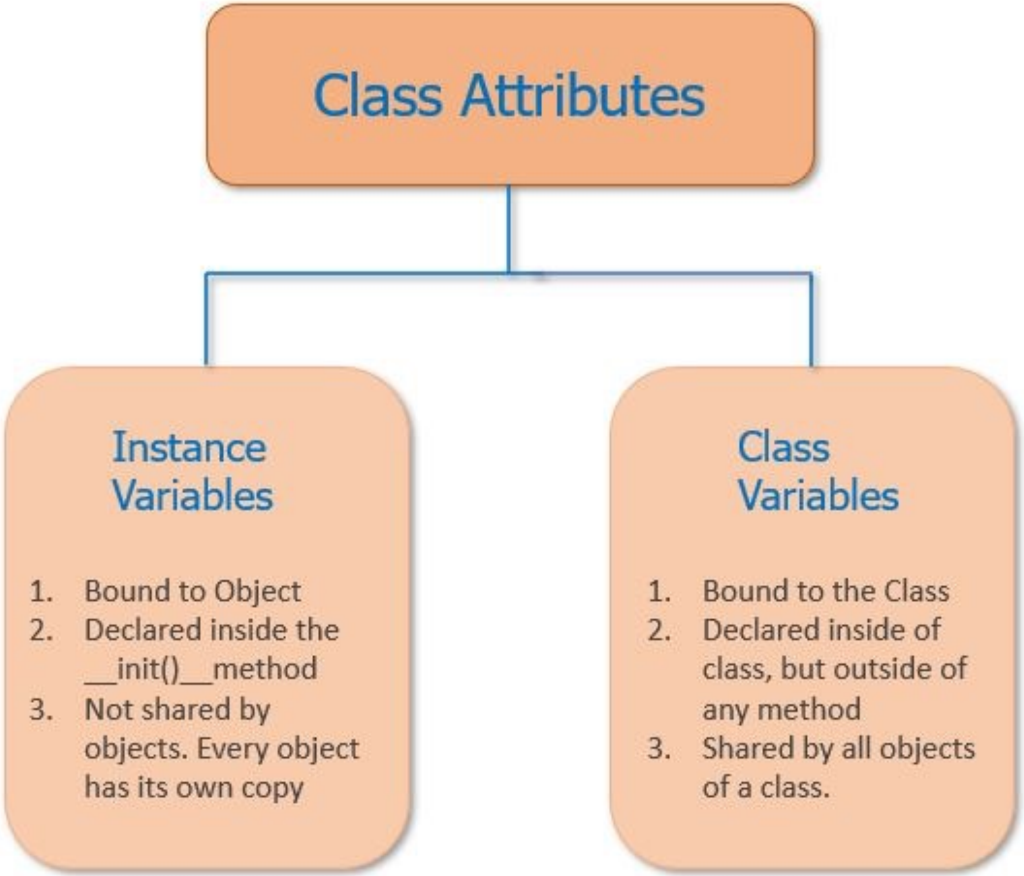
```
print(tesla.miles)
```

```
---------------------------------------------------------------------------
--
AttributeError                            Traceback (
t)
Cell In[22], line 1
----> 1 print(tesla.miles)

AttributeError: 'Car' object has no attribute 'miles'
```

```
print(toyota.miles)
```

```
500
```

Instance and Class variables



In Class, attributes can be defined into two parts:

- Instance variables: The instance variables are attributes attached to an instance of a class. We define instance variables in the constructor (the __init__() method of a class).

- Class Variables: A class variable is a variable that is declared inside of class, but outside of any instance method or __init__() method.

Class variables

```
class Car:
    # class variable
    driver = 1

    def __init__(self, m):
        # instance variable
        self.wheels = 4
        self.miles = m
```

```
tesla = Car(100)
toyota = Car(500)
print(tesla.driver)
print(toyota.driver)
```

```
1
1
```

```
Car.driver = 2
```

```
print(tesla.driver)
print(toyota.driver)
```

```
2
2
```

```
honda = Car(200)
print(honda.driver)
```

```
2
```

Class methods

Each class instance can have attributes attached to it for maintaining its state. Class instances can also have **methods** (defined by its class) for modifying its state.

```
s = 'abc'
s.upper()
```

```
'ABC'
```

```
s = [1, 2, 3]
s.append(4)
print(s)
```

```
[1, 2, 3, 4]
```

Class methods

Inside a Class, we can define the following three types of methods.

- **Instance method**: Used to access or modify the object state. If we use instance variables inside a method, such methods are called instance methods. It must have a `self` parameter to refer to the current object.
- **Class method**: Used to access or modify the class state. In method implementation, if we use only class variables, then such type of methods we should declare as a class method. The class method has a `cls` parameter which refers to the class.
- **Static method**: It is a general utility method that performs a task in isolation. Inside this method, we don't use instance or class variable because this static method doesn't take any parameters like self and cls.

Instance methods

- Instance methods are functions that belong to the object.
- Used to access or modify the object state.
- Use instance variables inside a method.
- Must have a `self` parameter to refer to the current object.

Instance methods

```
class Person:
  def __init__(self, name,
age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is
" + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

OUTPUT

```
Hello my name is
John
```

Instance methods

```python
class Student:
    def __init__(self, name, age):
        # Instance variable
        self.name = name
        self.age = age

    # inst. method to modify inst. var.
    def update_age(self, age):
        self.age = age

    # inst. method to add inst. var.
    def add_marks(self, marks):
        self.marks = marks
```

```python
# create object
stud = Student("Emma", 14)
print(stud.name, stud.age)

# call instance method
stud.update_age(18)
stud.add_marks(75)
print(stud.name, stud.age, stud.marks)
```

OUTPUT

```
Emma 14
Emma 18 75
```

Instance methods

```
class Car:
    def __init__(self,initial_mileage):
        self.wheels = 4
        self.miles = initial_mileage
    def drive(self):
        self.miles += 1
        print ('The car drived one more mile')
        print ('The current mileage of this car is %s'%self.miles)
```

```
tesla = Car(initial_mileage = 100)
```

```
tesla.miles
```

```
100
```

```
tesla.drive()
```

```
The car drived one more mile
The current mileage of this car is 101
```

```
tesla.miles
```

```
101
```

The `__str__()` function

- The __str__() function controls what should be returned when the class object is represented as a string.
- If the __str__() function is not set, the string representation of the object is returned

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)
print(p1)
```

OUTPUT

```
&lt;__main__.Person object at
0x2ae2083ab100&gt;
```

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def __str__(self):
    return f"{self.name}({self.age})"

p1 = Person("John", 36)
print(p1)
```

```
John(36)
```

The `__str__()` function

- The __str__() function controls what should be returned when the class object is represented as a string.
- If the __str__() function is not set, the string representation of the object is returned

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)
print(p1)
```

OUTPUT

```
<__main__.Person object at
0x2ae2083ab100>
```

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def __str__(self):
    return f"{self.name}({self.age})"

p1 = Person("John", 36)
print(p1)
```

```
John(36)
```

Class methods

- Used to access or modify the class state.
- To make a method as class method, add @classmethod decorator before the method definition, and add cls as the first parameter to the method.
- The @classmethod decorator is a built-in function decorator. In Python, we use the @classmethod decorator to declare a method as a class method.
- Syntax for creating a class method

```
class C(object):
     @classmethod
     def fun(cls, arg1, arg2, ...):
          ....
```

https://www.geeksforgeeks.org/python-classes-and-objects/

Class methods

The class method can be called using ClassName.method_name()

```python
class Student:
    school_name = 'ABC School' # class variable

    def __init__(self, name, age):
        self.name = name # instance variable
        self.age = age # instance variable

    # instance method
    def show(self):
        print(self.name, self.age, 'School:', Student.school_

    # class method
    @classmethod
    def change_school(cls, school_name):
        # modify class variable
        cls.school_name = school_name
```

```python
jessa = Student('Jessa', 20)
jessa.show()
```

```
Jessa 20 School: ABC School
```

```python
# change school_name
Student.change_school('XYZ School')
jessa.show()
```

```
Jessa 20 School: XYZ School
```

```python
john = Student('John', 22)
john.show()
```

```
John 22 School: XYZ School
```

Static methods

- A static method is bound to the class and not the object of the class. Therefore, we can call it using the class name.
- A static method doesn't have access to the class and instance variables because it does not receive an implicit first argument like `self` and `cls`. Therefore **it cannot modify the state of the object or class.**
- To make a method a static method, add @staticmethod decorator before the method definition.

```
class C(object):
    @staticmethod
    def fun(arg1, arg2, ...):
        ...
```

https://www.geeksforgeeks.org/python-classes-and-objects/

Static methods

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # a static method to check if a
Person is adult or not.
    @staticmethod
    def isAdult(age):
        return age > 18
```

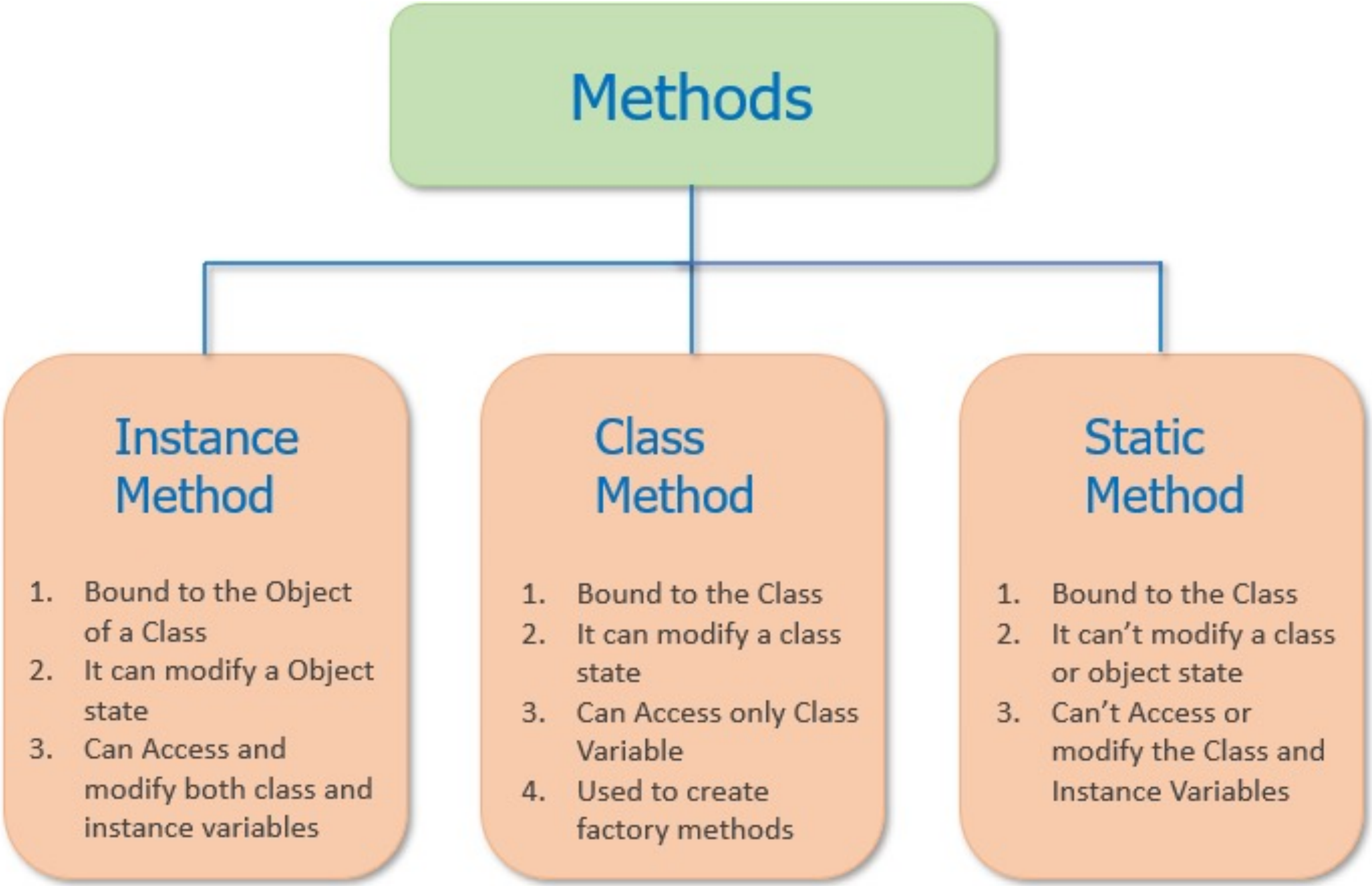
```
p1 = Person('mayank', 21)

print(p1.age)

# print the result
print(Person.isAdult(22))
```

OUTPUT

```
21
True
```

https://www.geeksforgeeks.org/python-classes-and-objects/

Class methods



Inheritance（继承）

- Inheritance allows us to define a class that inherits all the methods and properties from another class.
- Parent class is the class being inherited from, also called **base class**.
- Child class is the class that inherits from another class, also called **derived class**.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Inheritance

- The main purpose of inheritance is the reusability of code because we can use the existing class to create a new class instead of creating it from scratch.

- In inheritance, the child class acquires all the data members, properties, and functions from the parent class. Also, a child class can also provide its specific implementation to the methods of the parent class.

- For example, In the real world, Car is a sub-class of a Vehicle class. We can create a Car by inheriting the properties of a Vehicle such as Wheels, Colors, Fuel tank, engine, and add extra properties in Car as required.

Inheritance

```python
# parent class
class Person:
    # __init__ is known as the constructor
    def __init__(self, name, idnumber):
        self.name = name
        self.idnumber = idnumber
    def display(self):
        print(self.name)
        print(self.idnumber)

# child class
class Employee(Person):
    def __init__(self, name, idnumber, salary, company):
        self.salary = salary
        self.company = company

        # invoking the __init__ of the parent class
        Person.__init__(self, name, idnumber)
```

```python
a = Person('Rahul', 886012)
a.display()
```

```
Rahul
886012
```

```python
b = Employee('Rahul', 886012, 20000, 'SUSTech')
```

```python
print(b.name)
print(b.salary)
print(b.company)
```

```
Rahul
20000
SUSTech
```

```python
b.display()
```

```
Rahul
886012
```

Inheritance

```
# parent class
class Person:
    # __init__ is known as the constructor
    def __init__(self, name, idnumber):
        self.name = name
        self.idnumber = idnumber
    def display(self):
        print(self.name)
        print(self.idnumber)

# child class
class Employee(Person):
    def __init__(self, name, idnumber, salary, company):
        self.salary = salary
        self.company = company
```

Python program to demonstrate error if we forget to invoke __init__() of the parent

If you forget to invoke the __init__() of the parent class then its instance variables would not be available to the child class.

Inheritance

```
b = Employee('Rahul', 886012, 20000, 'SUSTech')
print(b.name)
print(b.salary)
print(b.company)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[3], line 2
      1 b = Employee('Rahul', 886012, 20000, 'SUSTech')
----> 2 print(b.name)
      3 print(b.salary)
      4 print(b.company)

AttributeError: 'Employee' object has no attribute 'name'
```

Inheritance

Different types of Inheritance:

- **Single inheritance**: When a child class inherits from only one parent class, it is called single inheritance. We saw an example above.

- **Multiple inheritances**: When a child class inherits from multiple parent classes, it is called multiple inheritances.

Multiple inheritances

```python
# parent class1
class Person:
    # __init__ is known as the constructor
    def __init__(self, name, idnumber):
        self.name = name
        self.idnumber = idnumber
    def display(self):
        print(self.name)
        print(self.idnumber)

# parent class2
class Company:
    # __init__ is known as the constructor
    def __init__(self, company):
        self.company = company
    def company_info(self):
        print(self.company)

# child class
class Employee(Person, Company):
    def __init__(self, name, idnumber, salary, company)
        self.salary = salary
        Person.__init__(self, name, idnumber)
        Company.__init__(self, company)
```

```python
a = Employee('John', '00100', 1000, 'Google')
```

```python
print(a.name)
print(a.company)
print(a.salary)
```

```
John
Google
1000
```

```python
a.display()
a.company_info()
```

```
John
00100
Google
```

Inheritance

In Python, based upon the number of child and parent classes involved, there are five types of inheritance. The type of inheritance are listed below:

1. Single inheritance
2. Multiple Inheritance
3. Multilevel inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

See *Types of inheritance.pdf* on BB

CS112: Introduction to Python programming

Week 10: Numpy & Scipy

Numpy

Install Documentation Learn Community About Us Contribute

https://numpy.org/

NumPy

The fundamental package for scientific computing with Python

GET STARTED

D&I Grant from CZI Including NumPy, SciPy, Matplotlib and Pandas

- NumPy is a Python package. It stands for 'Numerical Python'. It is a library consisting of multidimensional array objects and a collection of routines for processing of array.

Numpy

- NumPy is the fundamental package for scientific computing in Python.
- It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

Numpy

- Install numpy
- Using Anaconda
  - conda install numpy
- To test whether NumPy module is properly installed, try to import it from Python:

```
import numpy as np
```

Numpy—Ndarray Object

- The most important object defined in NumPy is an N-dimensional array type called **ndarray**. It describes the collection of items of the **same type**. Items in the collection can be accessed using a zero-based index.

Numpy—Ndarray Object

- Create an ndarray:
  - numpy.array(object), *dtype*)

  Object: can be a list or tuple or nested list

  dtype : Desired data type of array, optional

```
import numpy as np
a = [0, 0, 1, 4]
b = np.array(a)
print(b)
```

```
[0 0 1 4]
```

```
type(b)
```

```
numpy.ndarray
```

```
c = (0, 0, 1, 4)
d = np.array(c)
print(d)
```

```
[0 0 1 4]
```

```
type(d)
```

```
numpy.ndarray
```

```
a2d = np.array([[1, 2, 3], [4, 5, 6]])
print(a2d)
```

```
[[1 2 3]
 [4 5 6]]
```

NumPy data types

NumPy has some extra data types, and refer to data types with one character

- i - integer
- b - boolean
- u - unsigned integer
- f - float
- c - complex float
- m - timedelta
- M - datetime
- O - object
- S - string
- U - unicode string
- V - fixed chunk of memory for other type (void)

Check data type of an existing array

arr.dtype

```
import numpy as np
a = [0, 0, 1, 4]
b = np.array(a)
```

```
print(b.dtype)
```

```
int64
```

Numpy—Ndarray Object

```
import numpy as np
a = [0, 0, 1, 4]
b = np.array(a)
print(b.dtype)
```

```
int64
```

b is integer arrays, as it is created from a list of integers

```
e = np.array([1, 4.2, -2, 7])
print(e.dtype)
```

```
float64
```

- e is a floating point array even though only one of the elements of the list from which it was made was a floating point number
- The array function automatically promotes all the numbers to the type of the most general entry in the list

Numpy—Ndarray Object

```python
a = [0, 0, 1, 4]
b = np.array(a, dtype = 'f')
print(b)
print(b.dtype)
```

```
[0. 0. 1. 4.]
float32
```

numpy.array(object), *dtype*)

Object: can be a list or tuple or nested list

dtype : Desired data type of array, optional

Numpy—Ndarray Object

- ndarray to list

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])
a
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

```
a.tolist()
```

```
[[1, 2, 3], [4, 5, 6]]
```

Array placeholder content

- Using the NumPy `zeros` and `ones` function to create arrays where all the elements are either zeros or ones

- They each take one mandatory argument, the number of elements in the array, and one optional argument that specifies the data type of the array. If unspecified, the data type is a float

```
a = np.zeros((2,3))
a
```

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

```
a = np.ones((2,3))
a
```

```
array([[1., 1., 1.],
       [1., 1., 1.]])
```

```
np.ones((2, 3), dtype = int)
```

```
array([[1, 1, 1],
       [1, 1, 1]])
```

Array placeholder content

```
>>> np.full((2,2),2)
array([[2, 2],
       [2, 2]])
```

Return a new array of given shape and type, filled with \`fill_value\`.

```
>>> np.eye(2,3)
array([[1., 0., 0.],
       [0., 1., 0.]])
```

Return a 2-D array with ones on the diagonal and zeros elsewhere.

```
>>> np.identity(2)
array([[1., 0.],
       [0., 1.]])
```

Return the identity array.
The identity array is a square array with ones on the main diagonal.

```
>>> np.random.random((2,2))
array([[0.6, 0.4],
       [0.1, 0.2]])
```

Return random floats in the half-open interval [0.0, 1.0).

NumPy `linspace` and `logspace` functions

- The `linspace` function creates an array of *N* evenly spaced points between a starting point and an ending point. The form of the function is `linspace`(start, stop, N). If the third argument *N* is omitted, then *N*=50:

```
>>> np.linspace(0, 10, 5)
array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

NumPy `linspace` and `logspace` functions

- The `logspace` function produces evenly spaced points on a logarithmically spaced scale. The form of the function is `logspace(start, stop, N)`. The start and stop refer to a power of 10, i.e., the array starts at 10^{start} and ends at 10^{stop}:

```
>>> np.set_printoptions(precision=1)
>>> np.logspace(0, 3, 3)
array([    1. ,    31.6, 1000. ])
```

NumPy `arange` function

- The **`arange`** function return evenly spaced values within a given interval. numpy.arange(start, stop, *step*)
- In general, `arange` produces an integer array if the arguments are all integers; if any one of the arguments is a float, the generated array would be a float

```
>>> np.arange(0, 10, 2)
array([0, 2, 4, 6, 8])
>>> np.arange(0., 10, 2)
array([0., 2., 4., 6., 8.])
>>> np.arange(0, 10, 1.5)
array([0. , 1.5, 3. , 4.5, 6. , 7.5, 9. ])
```

NumPy `arange` function

- The **`arange`** function return evenly spaced values within a given interval. numpy.arange(start, stop, *step*)
- In general, `arange` produces an integer array if the arguments are all integers; if any one of the arguments is a float, the generated array would be a float

```
>>> np.arange(0, 10, 2)
array([0, 2, 4, 6, 8])
>>> np.arange(0., 10, 2)
array([0., 2., 4., 6., 8.])
>>> np.arange(0, 10, 1.5)
array([0. , 1.5, 3. , 4.5, 6. , 7.5, 9. ])
```

For integer arguments the function is roughly equivalent to the Python built-in **`range`**, but returns an **`ndarray`** rather than a **`range`** instance.
When using a non-integer step, such as 0.1, it is often better to use **`numpy.linspace`**.

Array attributes

```
a2d = np.array([[1, 2, 3], [4, 5, 6]])
print(a2d.ndim)
```

```
2
```

```
print(a2d.shape)
```

```
(2, 3)
```

```
print(a2d.size)
```

```
6
```

```
print(a2d.dtype)
```

```
int64
```

ndarray.ndim

returns the number of array dimensions

ndarray.shape

returns a tuple consisting of array dimensions. It can also be used to resize the array.

For more introduction on array attributes and methods:

https://numpy.org/doc/stable/reference/arrays.ndarray.html

Array Indexing & Slicing

- Similar to Python lists, numpy arrays can be sliced.
- One-dimensional arrays can be indexed and sliced the same way as strings and lists, i.e., array indexes are 0-based:

```
import numpy as np
a = np.array([1, 2, 3, 4])
```

```
a[2]
```

```
3
```

```
a[1:4]
```

```
array([2, 3, 4])
```

```
a[::-1]
```

```
array([4, 3, 2, 1])
```

```
a[:4:2]
```

```
array([1, 3])
```

```
a[[1, 3]]
```

```
array([2, 4])
```

Array Indexing & Slicing

- Multi-dimensional arrays can be indexed and sliced per axis:

```
import numpy as np
a = np.array([
        [ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 10, 11, 12]])
```

```
a[0, 1]
```

```
2
```

```
a[0]
```

```
array([1, 2, 3, 4])
```

```
a[0][1]
```

```
2
```

```
a[:2, 1:3]
```

```
array([[2, 3],
       [6, 7]])
```

```
a[1, :]
```

```
array([5, 6, 7, 8])
```

```
a[:, -1]
```

```
array([ 4,  8, 12])
```

```
a[1:3, :]
```

```
array([[ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

Array Indexing & Slicing

- Multi-dimensional arrays can be indexed and sliced per axis:

```
import numpy as np
a = np.array([
        [ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 10, 11, 12]])
```

```
a[0, 1]
```

```
2
```

```
a[0]
```

```
array([1, 2, 3, 4])
```

```
a[0][1]
```

```
2
```

```
a[[0, 1, 2], [0, 1, 0]]
```

```
array([1, 6, 9])
```

```
for row in a:
    print(row)
```

```
[1 2 3 4]
[5 6 7 8]
[ 9 10 11 12]
```

```
for element in a.flat:
    print(element)
```

```
1
2
3
4
5
6
7
8
9
10
11
12
```

Array Indexing & Slicing

- Boolean indexing

```
import numpy as np
a = np.arange(-2, 5)
a
```

```
array([-2, -1,  0,  1,  2,  3,  4])
```

```
a > 0
```

```
array([False, False, False,  True,  True,  True,  True])
```

```
a[a>0]
```

```
array([1, 2, 3, 4])
```

Array Indexing & Slicing

- Boolean indexing

```
import numpy as np
a = np.arange(-2, 5)
a
```

```
array([-2, -1,  0,  1,  2,  3,  4])
```

```
a[a>0] = 100
```

```
a
```

```
array([ -2,  -1,   0, 100, 100, 100, 100])
```

Array Indexing & Slicing

- Boolean indexing

```
import numpy as np
a = np.arange(-2, 5)
a
```

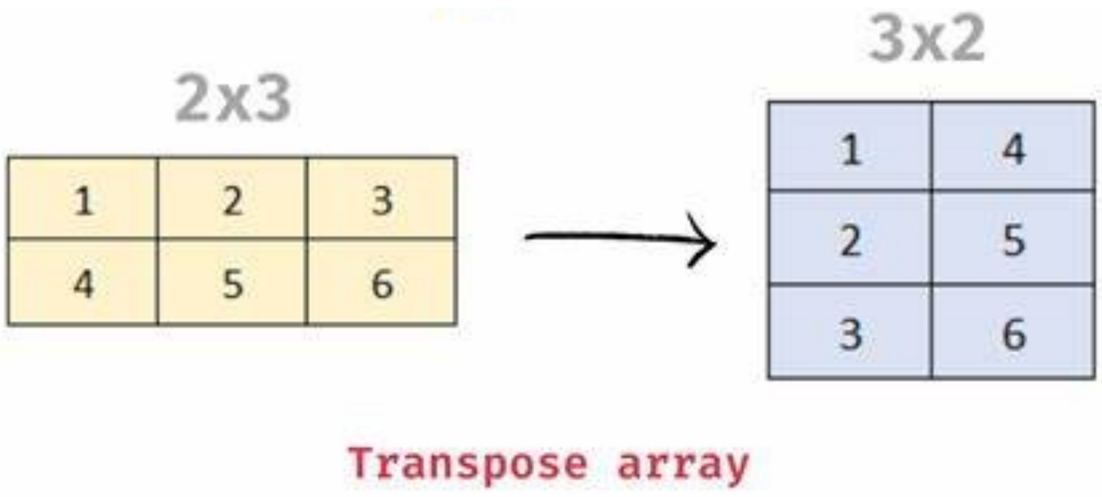
```
array([-2, -1,  0,  1,  2,  3,  4])
```

```
a[a==100] = -100
```

```
a
```

```
array([  -2,   -1,    0, -100, -100, -100, -100])
```

Array transpose（转置）




```
a2d = np.array([[1, 2], [4, 5], [6, 7]])
print(a2d)
```

```
[[1 2]
 [4 5]
 [6 7]]
```

array.T

```
print(a2d.T)
```

```
[[1 4 6]
 [2 5 7]]
```

numpy.transpose()

```
print(np.transpose(a2d))
```

```
[[1 4 6]
 [2 5 7]]
```

Array adding elements

- `numpy.append(arr, values, axis=None)`

Append values to the end of an array.

arr: *array_like*. Values are appended to a copy of this array.
values: *array_like*. These values are appended to a copy of *arr*. It must be of the correct shape (the same shape as *arr*, excluding *axis*).
axis: *int, optional*. The axis along which *values* are appended. If *axis* is not given, both *arr* and *values* are flattened before use.

Returns:
A copy of *arr* with *values* appended to *axis*. Note that append does not occur in-place: a new array is allocated and filled. If *axis* is None, *out* is a flattened array.

Array adding elements

- `numpy.append(arr, values, axis=None)`

Append values to the end of an array.

```
a2d = np.array([[1, 2], [4, 5], [6, 7]])
print(a2d)
print(a2d.shape)
```

```
[[1 2]
 [4 5]
 [6 7]]
(3, 2)
```

```
new = np.array([[0, 0]])
print(new)
print(new.shape)
```

```
[[0 0]]
(1, 2)
```

```
np.append(a2d, new, axis = 0)
```

```
array([[1, 2],
       [4, 5],
       [6, 7],
       [0, 0]])
```

```
np.append(a2d, new, axis = 1)
```

```
---------------------------------------------------------------------------
ValueError
t)
```

Array adding elements

- `numpy.append(arr, values, axis=None)`

Append values to the end of an array.

```
a2d = np.array([[1, 2], [4, 5], [6, 7]])
print(a2d)
print(a2d.shape)
```

```
[[1 2]
 [4 5]
 [6 7]]
(3, 2)
```

```
new = np.array([[0, 0]])
print(new)
print(new.shape)
```

```
[[0 0]]
(1, 2)
```

```
np.append(a2d, new)
```

```
array([1, 2, 4, 5, 6, 7, 0, 0])
```

If *axis* is not given, both *arr* and *values* are flattened before use.

Array adding elements

- `numpy.append(arr, values, axis=None)`

Append values to the end of an array.

```
a2d = np.array([[1, 2], [4, 5], [6, 7]])
print(a2d)
print(a2d.shape)
```

```
[[1 2]
 [4 5]
 [6 7]]
(3, 2)
```

```
new = np.array([[0, 0]])
print(new)
print(new.shape)
```

```
[[0 0]]
(1, 2)
```

```
new2d = np.array([[0, 0], [-1, -1]])
print(new2d)
print(new2d.shape)
```

```
[[ 0  0]
 [-1 -1]]
(2, 2)
```

```
np.append(a2d, new2d, axis = 0)
```

```
array([[ 1,  2],
       [ 4,  5],
       [ 6,  7],
       [ 0,  0],
       [-1, -1]])
```

Array adding elements

- `numpy.concatenate((a1, a2, ...), axis=0)`
- Join a sequence of arrays along an existing axis.

```
a2d = np.array([[1, 2], [4, 5], [6, 7]])
print(a2d)
print(a2d.shape)
```

```
[[1 2]
 [4 5]
 [6 7]]
(3, 2)
```

```
new2d = np.array([[0, 0], [-1, -1]])
print(new2d)
print(new2d.shape)
```

```
[[ 0  0]
 [-1 -1]]
(2, 2)
```

```
np.concatenate((a2d, new2d), axis = 0)
```

```
array([[ 1,  2],
       [ 4,  5],
       [ 6,  7],
       [ 0,  0],
       [-1, -1]])
```

Array adding elements

- `numpy.concatenate((a1, a2, ...), axis=0)`
- Join a sequence of arrays along an existing axis.

```
a1 = np.zeros((2, 4))
a2 = np.ones((3, 4))
a3 = np.full((4, 4), 2)
print(a1)
print(a2)
print(a3)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]]
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
[[2 2 2 2]
 [2 2 2 2]
 [2 2 2 2]
 [2 2 2 2]]
```

```
np.concatenate((a1, a2, a3))
```

```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [2., 2., 2., 2.],
       [2., 2., 2., 2.],
       [2., 2., 2., 2.],
       [2., 2., 2., 2.]])
```

Array deleting elements

- `numpy.delete(arr, obj, axis=None)`
- Return a new array with sub-arrays along an axis deleted.

```
import numpy as np
arr = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
arr
```

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

```
np.delete(arr, 1, 0)
```

```
array([[ 1,  2,  3,  4],
       [ 9, 10, 11, 12]])
```

```
np.delete(arr, [1, 3], 1)
```

```
array([[ 1,  3],
       [ 5,  7],
       [ 9, 11]])
```

If *axis* is None, *obj* is applied to the flattened array.

```
np.delete(arr, [1, 3])
```

```
array([ 1,  3,  5,  6,  7,  8,  9, 10, 11, 12])
```

Array Stacking & splitting

```
a = np.array([[3, 1, 2], [8, 7, 9]])
b = np.array([[2, 4, 6], [5, 4, 8]])
np.vstack((a, b))
```

```
array([[3, 1, 2],
       [8, 7, 9],
       [2, 4, 6],
       [5, 4, 8]])
```

nunpy.vstack(tup):
Stack arrays in sequence vertically (row wise).
tup: *sequence of ndarrays*

```
np.hstack((a, b))
```

```
array([[3, 1, 2, 2, 4, 6],
       [8, 7, 9, 5, 4, 8]])
```

nunpy.hstack(tup):
Stack arrays in sequence horizontally (column wise).
tup: *sequence of ndarrays*

Array Stacking & splitting

```
c = np.hstack((a, b))
```

```
np.hsplit(c, 3)
```

```
[array([[3, 1],
        [8, 7]]),
 array([[2, 2],
        [9, 5]]),
 array([[4, 6],
        [4, 8]])]
```

```
np.vsplit(c, 2)
```

```
[array([[3, 1, 2, 2, 4, 6]]), array([[8, 7, 9, 5, 4, 8]])]
```

numpy.vsplit(ary, indices_or_sections)

Split an array into multiple sub-arrays vertically (row-wise).

numpy.hsplit(ary, indices_or_sections)

Split an array into multiple sub-arrays horizontally (column-wise).

Shape & reshape

- ndarray.shape

  This array attribute returns a tuple consisting of array dimensions. It can also be used to resize the array.

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])
print (a.shape)
```

```
(2, 3)
```

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])
print (a)
```

```
[[1 2 3]
 [4 5 6]]
```

```
# this resizes the ndarray
a.shape = (3,2)
print (a)
```

```
[[1 2]
 [3 4]
 [5 6]]
```

Shape & reshape

- NumPy also provides a reshape() function to resize an array.

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])
b = a.reshape(3,2)
print (b)
```

```
[[1 2]
 [3 4]
 [5 6]]
```

Shape & reshape

`numpy.ravel()`

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])
b = a.reshape(3,2)
print (b)
```

```
[[1 2]
 [3 4]
 [5 6]]
```

```
a.ravel()
```

```
array([1, 2, 3, 4, 5, 6])
```

Return a flattened array.

Array Math

- Basic mathematical functions operate **elementwise** on arrays, and are available both as operator overloads and as functions in the numpy module:

```
import numpy as np
x = np.array([[1, 2], [3, 4]])
y = np.array([[5, 6], [7, 8]])
```

```
print(x)
```

```
[[1 2]
 [3 4]]
```

```
print(x + 1)
```

```
[[2 3]
 [4 5]]
```

```
print(x * 2)
```

```
[[2 4]
 [6 8]]
```

```
print(x - y)
```

```
[[-4 -4]
 [-4 -4]]
```

```
print(x + y)
```

```
[[ 6  8]
 [10 12]]
```

Array Math

```
import numpy as np
x = np.array([[1, 2], [3, 4]])
print(x)
```

```
[[1 2]
 [3 4]]
```

```
print(np.log(x))
```

```
[[0.         0.69314718]
 [1.09861229 1.38629436]]
```

```
print(np.sqrt(x))
```

```
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

```
print(np.sin(x))
```

```
[[ 0.84147098  0.90929743]
 [ 0.14112001 -0.7568025 ]]
```

```
print(np.mean(x))
```

```
2.5
```

```
print(np.max(x))
```

```
4
```

```
print(np.mean(x, axis = 0))
```

```
[2. 3.]
```

```
print(np.mean(x, axis = 1))
```

```
[1.5 3.5]
```

```
print(np.max(x, axis = 0))
```

```
[3 4]
```

```
print(np.max(x, axis = 1))
```

```
[2 4]
```

Array Math

- Basic mathematical functions operate **elementwise** on arrays, and are available both as operator overloads and as functions in the numpy module:

- These operations with arrays are called ***vectorized*** operations because the entire array, or “vector,” is processed as a unit.
- Vectorized operations are **much faster** than processing each element of an array one by one.
- Writing code that takes advantage of these kinds of vectorized operations is almost always preferred to other means of accomplishing the same task

Array & List

- Lists are part of the core Python programming language; arrays are a part of the numerical computing package NumPy
- The elements of a NumPy array must all be of the same type, whereas the elements of a Python list can be of completely different types
- Arrays allow Boolean indexing; lists do not
- NumPy arrays support "vectorized" operations like element-by-element addition and multiplication
- Adding one or more additional elements to a NumPy array creates a new array and destroys the old one. Therefore, it can be very inefficient to build up large arrays by appending elements one by one. By contrast, elements can be added to a list without creating a whole new list