Object Oriented Programming

#11

Serdar ARITAN

Biomechanics Research Group,
Faculty of Sports Sciences, and
Department of Computer Graphics
Hacettepe University, Ankara, Turkey

Python String format() Method

Customizing String Formatting

The format() method formats the specified value(s) and insert them inside the string's placeholder.
The placeholder is defined using curly brackets: **{}**.
The **format()** method returns the formatted string.

```
txt = "For only {price:.2f} dollars!"
print(txt.format(price = 49))

txt1 = "My name is {fname}, I'm {age}".format(fname = "Serdar", age = 55)
txt2 = "My name is {0}, I'm {1}".format("Serdar", 55)
txt3 = "My name is {}, I'm {}".format("Serdar", 55)
```

Python String format() Method

Customizing String Formatting

[:<] Left aligns [:>] Right aligns [:^] Center aligns

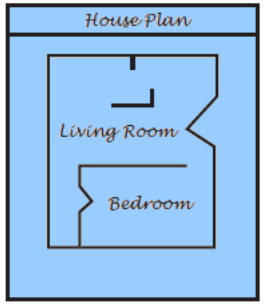
```
txt = "We have {:<8} students."
print(txt.format(38))
We have 38       students.
txt = "We have {:^8} students."
print(txt.format(38))
We have    38    students.
```

[:b] Binary [:c] Unicode [:d] Decimal [:f] Fixed point

```
txt = "We have {:b} students."
print(txt.format(38))
We have 100110 students.
txt = "We have {:c} students."
print(txt.format(38))
We have & students.
txt = "We have {:.2f} students."
print(txt.format(38))
We have 38.00 students.
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Blueprint that describes a house



Instances of the house described by the blueprint

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Changing the String Representation of Instances

To change the string representation of an instance, define the `__str__()` and `__repr__()` methods. For example:

```
class Pair:
        def __init__(self, x, y):
                self.x = x
                self.y = y
        def __repr__(self):
                return 'Pair({0.x!r}, {0.y!r})'.format(self)
        def __str__(self):
                return '({0.x!s}, {0.y!s})'.format(self)
```

The `__repr__()` method returns the code representation of an instance, and is usually the text you would type to re-create the instance. The `__str__()` method converts the instance to a string, and is the output produced by the `str()` and `print()` functions.

CLASSES AND OBJECT-ORIENTED PROGRAMMING

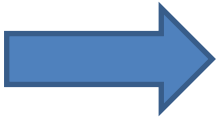
Changing the String Representation of Instances

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

        def __repr__(self):
                #return 'Pair({0.x!r}, {0.y!r})'.format(self)
                #return 'Pair(%r, %r)' % (self.x, self.y)
                return  f'Pair({self.x}, {self.y})'

        def __str__(self):
                #return '({0.x!s}, {0.y!s})'.format(self)
                #return '(%s, %s)' % (self.x, self.y)
                return  f'({self.x}, {self.y})'
```

!r (repr), !s (str) and !a (ascii) were kept around just to ease compatibility with the str.format alternative, **you don't need to use them with f-strings**.

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Customizing String Formatting



```
_formats = {
            'row' : '{d.x}-{d.y}',
            'col' : '\n{d.x}\n{d.y}'
            }

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

        def __repr__(self):
                return 'Pair({0.x!r}, {0.y!r})'.format(self)
        def __str__(self):
                return '({0.x!s}, {0.y!s})'.format(self)
        def __format__(self, code):
                if code == '':
                        code = 'row'
                fmt = _formats[code]
                return fmt.format(d=self)
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Customizing String Formatting

The __format__() method provides a hook into Python's string formatting function-ality. It's important to emphasize that the interpretation of format codes is entirely up to the class itself.

```
>>> p = Pair(3,4)
>>> p
Pair(3, 4)
>>> format(p, 'col')
'\n3\n4'
>>> print(p)
(3, 4)
>>> print('The pair is {:col}'.format(p))
The pair is
3
4
>>> print('The pair is {:row}'.format(p))
The pair is 3-4
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
import random
# The Coin class simulates a coin that can be flipped.
class Coin:

    def __init__(self):
        self.sideup = 'Heads'

    def toss(self):
        if random.randint(0, 1) == 0:
            self.sideup = 'Heads'
        else:
            self.sideup = 'Tails'

    def get_sideup(self):
        return self.sideup
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
# The main function.
def main():
# Create an object from the Coin class
    my_coin = Coin()

    # Display the side of the coin that is facing up
    print('This side is up:', my_coin.get_sideup())

    # Toss the coin
    print('I am tossing the coin...')
    my_coin.toss()

    # Display the side of the coin that is facing up
    print('This side is up:', my_coin.get_sideup())

# Call the main function
main()
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
===== RESTART: CoinToss.py =====
This side is up: Heads
I am tossing the coin...
This side is up: Tails
>>>
===== RESTART: CoinToss.py =====
This side is up: Heads
I am tossing the coin...
This side is up: Tails
>>>
===== RESTART: CoinToss.py =====
This side is up: Heads
I am tossing the coin...
This side is up: Heads
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
# The main function.
def main():
# Create an object from the Coin class
    my_coin = Coin()

    # Display the side of the coin that is facing up
    print('This side is up:', my_coin.get_sideup())
    # Toss the coin
    print('I am tossing the coin...')
    my_coin.toss()
    # But now We are going to cheat!
    my_coin.sideup = 'Tails'
    # Display the side of the coin that is facing up
    print('This side is up:', my_coin.get_sideup())

# Call the main function
main()
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
===== RESTART: CoinToss.py =====
This side is up: Heads
I am tossing the coin...
This side is up: Tails
>>>
===== RESTART: CoinToss.py =====
This side is up: Heads
I am tossing the coin...
This side is up: Tails
>>>
===== RESTART: CoinToss.py =====
This side is up: Heads
I am tossing the coin...
This side is up: Tails
```



CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
# The main function.
def main():
# Create an object from the Coin class
    my_coin = Coin()

    # Display the side of the coin that is facing up
    print('This side is up:', my_coin.get_sideup())
    # Toss the coin
    print('I am tossing the coin...')
    my_coin.toss()
    # Delete the proporties!
    del my_coin.sideup
    # Display the side of the coin that is facing up
    print('This side is up:', my_coin.get_sideup())

# Call the main function
main()
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
===== RESTART: CoinToss.py =====
This side is up: Heads
I am tossing the coin...
Traceback (most recent call last):
  File "CoinToss.py", line 32, in <module>
    main()
  File "CoinToss.py", line 29, in main
    print('This side is up:', my_coin.get_sideup())
  File "CoinToss.py", line 14, in get_sideup
    return self.sideup
AttributeError: 'Coin' object has no attribute 'sideup'
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Rather than relying on language features to encapsulate data, Python programmers are expected to observe certain naming conventions concerning the intended usage of data and methods. The first convention is that any name that starts with a single leading underscore (_) should always be assumed to be internal implementation. For example:

```
class A:
    def __init__(self):
        self._internal = 0  # An internal attribute
        self.public = 1     # A public attribute
    def public_method(self):
        '''          A public method          '''
    def _internal_method(self):
```

Python doesn't actually prevent someone from accessing internal names. However, doing so is considered impolite, and may result in fragile code.

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Indicating Privacy Using Double Underscores (__)

Python does not enforce this separation between the designer and programmer. All methods and instance variables are public, so both designers and programmers have access. Python does provide support for the designer to indicate attributes that the programmer should not modify directly. Whenever a class designer names an attribute with two leading underscores, this is a message to anyone using the class that the designer considers this a private variable. No one should change or modify its value. To prevent this change from accidentally happening. They can be accessed outside the class, but we must add **`_ClassName`** to the start.

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
class NewClass (object):
    def __init__ (self, attribute='default', name='Instance'):
        self.name = name                    # public attribute
        self. __attribute = attribute  # a " private " attribute
    def __str__ (self):
        return '{} has attribute {}'.format(self.name, self.__attribute)

>>> inst1 = NewClass(name='Monty', attribute='Python')
>>> print(inst1)
Monty has attribute Python
>>> print(inst1.__attribute)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    print(inst1.__attribute)
AttributeError: 'NewClass' object has no attribute '__attribute'
>>> dir(inst1)
['_NewClass__attribute', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'name']
>>> print(inst1._NewClass__attribute)
Python
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

A property is a special member of a class.

```
_pi = 3.14159
class Circle:
        def __init__(self, radius):
        self.radius = radius

        @property
        def area(self):
                return self.__class__._pi * self.radius ** 2

        @property
        def perimeter(self):
                return 2 * self.__class__._pi * self.radius
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

A property is a special member of a class.

```
>>> c = Circle(4.0)
>>> c.radius
4.0
>>> c.area              # Notice lack of ()
50.26548245743669
>>> c.perimeter         # Notice lack of ()
25.132741228718345
>>> c.perimeter = 5
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    c.perimeter = 5
AttributeError: can't set attribute
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Creating an Instance Without Invoking `__init__`.

```
>>> ========= RESTART: =========
>>> c = Circle.__new__(Circle)
>>> c
return 'Circle(r = {0.radius!r})'.format(self)
AttributeError: 'Circle' object has no attribute 'radius'
>>> setattr(c, 'radius', 5)
>>> c
Circle(r = 5)
>>> c.perimeter
31.4159
p = getattr(c, 'perimeter')
>>> p
31.4159
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

A property is a special member of a class. It gets and sets a value.

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

p = Person('Serdar')

print(p.first_name)
p.first_name = 48
print(p.first_name)
del p.first_name
print(p.first_name)
```

name = 48 !

del !!!

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
class Person:
    def __init__(self, first_name):
        self.first_name = first_name
    # Getter function
    @property
    def first_name(self):
        return self._first_name

    # Setter function
    @first_name.setter
    def first_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._first_name = value

    # Deleter function (optional)
    @first_name.deleter
    def first_name(self):
        raise AttributeError('Can't delete attribute')
```

def first_name

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
A>>> a = Person('Serdar')
>>> a.first_name = 48
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    a.first_name = 48
  File "ManagedAttributes.py", line 15, in first_name
    raise TypeError('Expected a string')
TypeError: Expected a string
>>> del a.first_name
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    del a.first_name
  File "ManagedAttributes.py", line 21, in first_name
    raise AttributeError("Can't delete attribute")
AttributeError: Can't delete attribute
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
class Person:
    def __init__(self, first_name):
        self.set_first_name(first_name)

    # Getter function
    def get_first_name(self):
        return self._first_name
    # Setter function
    def set_first_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._first_name = value
    # Deleter function (optional)
    def del_first_name(self):
        raise AttributeError("Can't delete attribute")
    # Make a property from existing get/set methods
    name = property(get_first_name, set_first_name, del_first_name)
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
>>> a = Person('Serdar')
>>> a.name = 48
File "ManagedAttributes.py", line 15, in first_name
    raise TypeError('Expected a string')
TypeError: Expected a string
>>> del a.name
File "ManagedAttributes.py", line 21, in first_name
    raise AttributeError("Can't delete attribute")
AttributeError: Can't delete attribute
>>> a.set_first_name('Canan')
>>> a.get_first_name()
'Canan'
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Inheritance and the "Is a" Relationship

When one object is a specialized version of another object, there is an "is a" relationship between them. For example, a grasshopper is an insect. Here are a few other examples of the "is a" relationship:

- A car is a vehicle.
- A flower is a plant.
- A rectangle is a shape.
- A football player is an athlete.
- A destroyer is a ship

When an "is a" relationship exists between objects, it means that the specialized object has all of the characteristics of the general object, plus additional characteristics that make it special. In object-oriented programming, inheritance is used to create an "is a" relationship among classes.

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Inheritance involves a superclass and a subclass. The superclass is the general class and the subclass is the specialized class. You can think of the subclass as an extended version of the superclass. The subclass inherits attributes and methods from the superclass without any of them having to be rewritten. Furthermore, new attributes and methods may be added to the subclass, and that is what makes it a specialized version of the superclass.

Superclasses are also called **base classes**, and **subclasses** are also called **derived classes**. Either set of terms is correct. For consistency, this course we will use the terms superclass and subclass.

CLASSES AND OBJECT-ORIENTED PROGRAMMING

A class can inherit from one or more other classes in Python. **The class we want to derive from must first be defined.** The derived class is specified in the parentheses after the class name.

```
class A:
    def width(self):
        print("a, width called")

class B(A):
    def size(self):
        print("b, size called")
# Create new class instance.
b = B()
# Call method on B.
b.size()
# Call method from derived class.
b.width()
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Correct use of the super() function is actually one of the most poorly understood aspects of Python. Occasionally, you will see code written that directly calls a method in a parent like this.

```
class Base:
    def __init__(self):
        print('Base.__init__')
class A(Base):
    def __init__(self):
        Base.__init__(self)
        print('A.__init__')
a = A()
```

```
Base.__init__
A.__init__
```

Although this “works” for most code, it can lead to bizarre trouble in advanced code involving multiple inheritance.

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
class Base:
    def __init__(self):
        print('Base.__init__')
class A(Base):
    def __init__(self):
        Base.__init__(self)
        print('A.__init__')
class B(Base):
    def __init__(self):
        Base.__init__(self)
        print('B.__init__')
class C(A,B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)
        print('C.__init__')
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

You'll see that the `Base.__init__()` method gets invoked **twice**

```
>>> c = C()
Base.__init__
A.__init__
Base.__init__
B.__init__
C.__init__
>>>
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
class Base:
    def __init__(self):
        print('Base.__init__')
class A(Base):
    def __init__(self):
        super().__init__()
        print('A.__init__')
class B(Base):
    def __init__(self):
        super().__init__()
        print('B.__init__')
class C(A,B):
    def __init__(self):
        super().__init__()
        print('C.__init__')
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

each `__init__()` method only gets called once

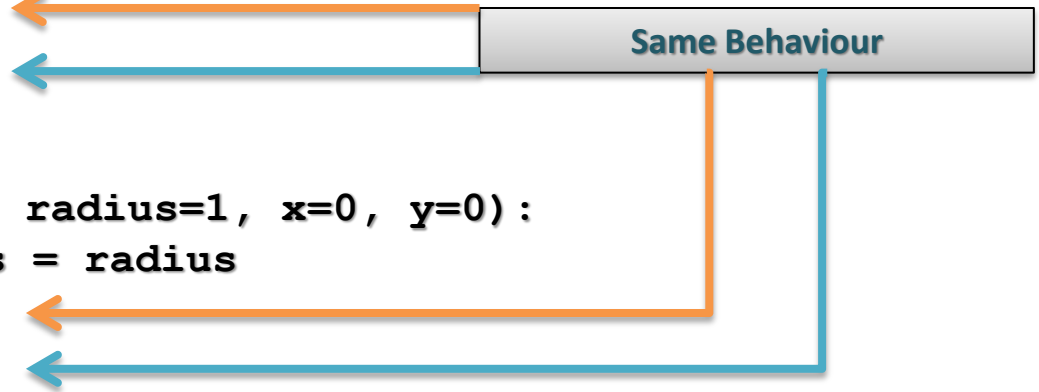
```
>>> c = C()
Base.__init__
B.__init__
A.__init__
C.__init__
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.A'>, <class
'__main__.B'>, <class '__main__.Base'>, <class 'object'>)
```

For every class that we define, Python computes what's known as a method resolution order (MRO) list. The MRO list is simply a linear ordering of all the base classes.

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Inheritance in Python is easier and more flexible than inheritance in compiled languages such as Java and C++ because the dynamic nature of Python doesn't force as many restrictions on the language.

```
class Square:
        def __init__(self, side=1, x=0, y=0):
                self.side = side
                self.x = x
                self.y = y

class Circle:
        def __init__(self, radius=1, x=0, y=0):
                self.radius = radius
                self.x = x
                self.y = y
```



CLASSES AND OBJECT-ORIENTED PROGRAMMING

Instead of defining the x and y variables in each shape class, abstract them out into a general **Shape** class, and have each class defining an actual shape inherit from that general class.

```
class Shape:
        def __init__(self, x, y):
            self.x = x
            self.y = y
class Square(Shape):
        def __init__(self, side=1, x=0, y=0):
            super().__init__(x, y)
            self.side = side
class Circle(Shape):
        def __init__(self, r=1, x=0, y=0):
            super().__init__(x, y)
            self.radius = r
```

Says Square inherits from Shape

Must call __init__ method of Shape

CLASSES AND OBJECT-ORIENTED PROGRAMMING

There are two requirements in using an inherited class in Python. The first requirement is defining the inheritance hierarchy, which you do by giving the classes inherited from, in parentheses, immediately after the name of the class being defined with the class keyword. In the previous code, Circle and Square both inherit from Shape. The second and more subtle element is the necessity to explicitly call the __init__ method of inherited classes. Python doesn't automatically do this for you, but you can use the super function to have Python figure out which inherited class to use. This is accomplished in the example code by the `super().__init__(x,y)` lines. Instead of using super, we could call Shape's `__init__` by explicitly naming the inherited class using `Shape.__init__(self, x, y)` which would also call the Shape initialization function with the instance being initialized.

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Inheritance comes into effect when you attempt to use a method that isn't defined in the base classes but is defined in the superclass. To see this, let's define another method in the Shape class called move, which will move a shape by a given displacement.

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

    def move(self, delta_x, delta_y):
        self.x = self.x + delta_x
        self.y = self.y + delta_y
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
>>> ========================== RESTART ==========================
>>> class Shape:
        def __init__(self, x, y):
            self.x = x
            self.y = y

        def move(self, delta_x, delta_y):
            self.x = self.x + delta_x
            self.y = self.y + delta_y

>>> class Circle(Shape):
        def __init__(self, r=1, x=0, y=0):
            super().__init__(x, y)
            self.radius = r

>>> c = Circle(1)
>>> c.move(3, 4)
>>> c.x
3
>>> c.y
4
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Inheritance allows an instance to inherit attributes of the class. Instance variables are associated with object instances, and only one instance variable of a given name exists.

```
class P:
    z = “hello”
    def set_p(self):
        self.x = “Class P”

    def print_p(self):
        print(self.x)

class C(P):
    def set_c(self):
        self.x = "Class C“

    def print_c(self):
        print(self.x)
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
>>> c = C()
>>> c.set_p()
>>> c.print_p()
Class P
>>> c.print_c()
Class P
>>> c.set_c()
>>> c.print_c()
Class C
>>> c.print_p()
Class C
>>>
```

The object `c` in this example is an instance of class `C`. `C` inherits from `P` but `c` doesn't inherit from some invisible instance of class `P`.

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
>>> c.z; C.z; P.z
'hello'
'hello'
'hello'
>>> C.z = "Bonjour"
>>> c.z; C.z; P.z
'Bonjour'
'Bonjour'
'hello'
>>> c.z = "Ciao"
>>> c.z; C.z; P.z
'Ciao'
'Bonjour'
'hello'
```

Similarly, if you try setting `z` through the instance `c`, a new instance variable is created, and you end up with three different variables.

CLASSES AND OBJECT-ORIENTED PROGRAMMING

When comparing objects, a **hash** code can be used for **better speed**. A dictionary uses hashes. With **__hash__** we can implement custom hash computations on a class. If one value is unique on the class, it makes an excellent hash.

```
class Snake:
    def __init__(self, name, color, unique_id):
        self.name = name
        self.color = color
        self.unique_id = unique_id
    def __hash__(self):
        # Hash on a unique value of the class.
        return int(self.unique_id)

# Hash now is equal to the unique ID values used.
p = Snake("Python", "green", 55)
print(hash(p))
p = Snake("Python", "green", 105)
print(hash(p))
```

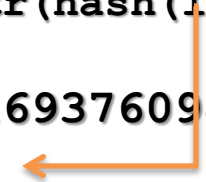
CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
int_val = 42
str_val = 'bco601'
flt_val = 45.356

# Printing the hash values.
# Notice Integer value doesn't change
print("The integer hash value is : " + str(hash(int_val)))
print("The string hash value is : " + str(hash(str_val)))
print("The float hash value is : " + str(hash(flt_val)))

The integer hash value is : 42
The string hash value is : 5061135356237158578
The float hash value is : 8208801112800788893
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
# tuple are immutable
tuple_val = (1, 2, 3, 4, 5)

# list are mutable
list_val = [1, 2, 3, 4, 5]

# Printing the hash values.
print("The tuple hash value is : " + str(hash(tuple_val)))
print("The list hash value is : " + str(hash(list_val)))
```

```
The tuple hash value is : -5659871693760987716
TypeError: unhashable type: 'list'
```



CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __eq__(self, other):
        if isinstance(other, Student):
            return self.name == other.name and self.age == other.age
        return False
    def __hash__(self):
        return hash((self.name, self.age))

# Creating instances
person1 = Student("Nihat", 30)
person2 = Student("Duru", 25)
person3 = Student("Nihat", 30)
# Using the custom objects in a set
students = {person1, person2, person3}
print(len(students))  #2 person1 and person3 are equal and have the same hash
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
# Checking hashes
print(hash(person1)) # Output: hash value of person1
print(hash(person2)) # Output: hash value of person2
print(hash(person3)) # Output: same hash value as person1


-6256235285033312628
49114157741098794 72
-6256235285033312628
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
# Creating instances
person1 = Student("Nihat", 30)
person2 = Student("Duru", 25)
person3 = Student("Nihat", 30.1)

-6256235285033312628
4911415774109879472
157056365813361925
```

- Usually comparing objects (which may involve several levels of recursion) is expensive.
- Preferably, the **`hash()`** function is an order of magnitude less expensive.
- Comparing two hashes is easier than comparing two objects.

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Method Overloading is a feature that allows a class to have two or more methods having same name, if their argument lists are different.
Argument lists could differ in –

1. Number of parameters.
2. Data type of parameters.
3. Sequence of Data type of parameters.

Method overloading is also known as Static Polymorphism.

```
def add_bullet(sprite, start, direction, speed):
def add_bullet(sprite, start, to, speed, acceleration):
# For bullets that are controlled by a script
def add_bullet(sprite, script):
# for bullets with curved paths
def add_bullet(sprite, curve, speed):
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
class Character(object):
    # your character __init__ and other methods go here

    def add_bullet(self, sprite=default, start= 0,
                   direction=None):

        if direction is None:
            # just use sprite and start
        else:
            # use sprite, start and direction

# direction will be none, so enter the 'if' clause
add_bullet('hollowPoint', 100 )
# direction isn't None, it's 90, so enter the 'else' clause
add_bullet('hollowPoint', 100, 90 )
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
class Character(object):
    # your character __init__ and other methods go here

    def add_bullet(self, *args, **kwargs):
        # args -- tuple of anonymous arguments
        # kwargs -- dictionary of named arguments
```

*args, **kwargs ???

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
>>> def f(*args, **kwargs):
        print ('args: ', args, ' kwargs: ', kwargs)

>>> f('a')
args:  ('a',)  kwargs:  {}
>>> f('bullet = hollow')
args:  ('bullet = hollow',)  kwargs:  {}
>>> f( bullet = 'hollow')
args:  ()  kwargs:  {'bullet': 'hollow'}
>>> f('hollowPoint', 100 )
args:  ('hollowPoint', 100)  kwargs:  {}
>>> f('hollowPoint', 100, 90)
args:  ('hollowPoint', 100, 90)  kwargs:  {}
>>> f('hollowPoint', 100, 90, speed = 120)
args:  ('hollowPoint', 100, 90)  kwargs:  {'speed': 120}
>>>
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Serializing a object is the process of converting the object to a stream of bytes that can be saved to a file for later retrieval. In Python, object serialization is called **pickling**. The Python standard library provides a module named **`pickle`** that has various functions for serializing, or pickling, objects. Once you import the pickle module, you perform the following steps to pickle an object:

- You open a file for binary writing.
- You call the pickle module's dump method to pickle the object and write it to the specified file.
- After you have pickled all the objects that you want to save to the file, you close the file.

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
>>> import pickle

>>> telefonrehberi = {'Hacettepe' : '312-2970000',
                      'SBF' : '312-2976890',
                      'SA'  : '312-2976893'}
>>> output_file = open('phonebook.dat', 'wb')
>>> pickle.dump(telefonrehberi, output_file)
>>> output_file.close()
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
>>> import pickle

>>> input_file = open('phonebook.dat', 'rb')
>>> pb = pickle.load(input_file)
>>> pb
{'SA': '312-2976893', 'Hacettepe': '312-2970000', 'SBF': '312-
2976890'}
>>>
>>> input_file.close()
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
import pickle

class SimpleObject:

    def __init__(self, name):
        self.name = name
        l = list(name)
        l.reverse()
        self.name_backwards = ''.join(l)
        return

data = []
data.append(SimpleObject('pickle'))
data.append(SimpleObject('cPickle'))
data.append(SimpleObject('last'))

output_file = open('mySimpleObjects.dat', 'wb')
pickle.dump(data, output_file)
output_file.close()
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
import pickle

input_file = open('mySimpleObjects.dat', 'rb')
mydata = pickle.load(input_file)

for i, o in enumerate(mydata):
    print(f'myData {i} : {o.name} ({o.name_backwards})')

input_file.close()
```

```
myData 0 : pickle (elkcip)
myData 1 : cPickle (elkciPc)
myData 2 : last (tsal)
```