Session 4

Python Classes/Objects

Python is an **object oriented programming** language.

Almost everything in Python is an object, with its *properties* and *methods*.

A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the keyword `class`:

PYTHON

```python
# Create a class named MyClass, with a property named x:
class MyClass:
  x = 5
```

Create Object

Now we can use the class named *MyClass* to create objects:

PYTHON

```python
# Create an object named p1, and print the value of x:
p1 = MyClass()
print(p1.x)

p2 = MyClass()
p3 = MyClass()
print(p2.x)
print(p3.x)

print(type(p1))
print(type(p2))
print(type(p3))
```

The `__init__()` Function

The examples above are classes and objects in their simplest form, and are not really useful in real life applications.

To understand the meaning of classes we have to understand the built-in `__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:

PYTHON

```python
# Create a class named Person, use the `__init__()` function to assign values for name and age:
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

> **Note:** The `__init__()` function is called automatically every time the class is being used to create a new object.

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:

PYTHON

```python
# The string representation of an object WITHOUT the `__str__()` function:

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

p1 = Person("John", 36)

print(p1)
```

PYTHON

```python
# The string representation of an object WITH the `__str__()` function:

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)
```

Object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object.

Let us create a method in the Person class:

```python
# Insert a function that prints a greeting, and execute it on the
p1 object:

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()
```

The self Parameter

The `self` parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

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:

```python
# Use the words 'mysillyobject' and 'abc' instead of 'self':

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

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

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

Modify Object Properties

You can modify properties on objects like this:

PYTHON

```python
# Set the age of p1 to 40:
p1.age = 40
```

Delete Object Properties

You can delete properties on objects by using the `del` keyword:

PYTHON

```python
# Delete the age property from the p1 object:
del p1.age
```

Delete Objects

You can delete objects by using the `del` keyword:

PYTHON

```python
# Delete the p1 object:
del p1
```

Example

```python
class Student:

    def __init__(self, name, student_id):

        self.name = name

        self.student_id = student_id

        self.lunch_status = False

    def reservation(self):

        if self.lunch_status:

            print(f"{self.name}, you have lunch today.")

        else:

            print(f"{self.name}, you don't have lunch today.")

    def reserve(self):

        if self.lunch_status:

            print(f"{self.name}, you have reserved before.")

        else:

            self.lunch_status = True

            print(f"{self.name}, your lunch is reserved.")

    def reserve_cancel(self):

        if self.lunch_status:

            self.lunch_status = False

            print(f"{self.name}, your reservation canceld.")
        else:
            print(f"{self.name}, you don't have reservation
today.")
```

Python Inheritance

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.

Create a Parent Class

Any class can be a parent class, so the syntax is the same as creating any other class:

```python
# Create a class named `Person`, with `firstname` and `lastname`
properties, and a `printname` method:

class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

# Use the Person class to create an object, and then execute the
printname method:

x = Person("John", "Doe")
x.printname()
```

Create a Child Class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

```python
# Create a class named `Student`, which will inherit the properties and methods from the `Person` class:

class Student(Person):
  pass
```

> **Note:** Use the `pass` keyword when you do not want to add any other properties or methods to the class.

Now the Student class has the same properties and methods as the Person class.

```python
# Use the `Student` class to create an object, and then execute the `printname` method:

x = Student("Mike", "Olsen")
x.printname()
```

Add the `__init__()` Function

So far we have created a child class that inherits the properties and methods from its parent.
We want to add the `__init__()` function to the child class (instead of the `pass` keyword).

> **Note:** The `__init__()` function is called automatically every time the class is being used to create a new object.

```python
# Add the `__init__()` function to the `Student` class:

class Student(Person):
  def __init__(self, fname, lname, age):
    self.age = age
```

When you add the `__init__()` function, the child class will no longer inherit the parent's `__init__()` function.

> **Note:** The child's `__init__()` function **overrides** the inheritance of the parent's `__init__()` function.

To keep the inheritance of the parent's `__init__()` function, add a call to the parent's `__init__()` function:

```python
class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)
```

Now we have successfully added the `__init__()` function, and kept the inheritance of the parent class, and we are ready to add functionality in the `__init__()` function.

Use the `super()` Function

Python also has a `super()` function that will make the child class inherit all the methods and properties from its parent:

```python
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
```

By using the `super()` function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

Add Methods

Add a method called `welcome` to the `Student` class:

PYTHON

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def welcome(self):
    print("Welcome", self.firstname, self.lastname, "to the class
of", self.graduationyear)
```