

# Classes, Objects & Methods





## Learning Outcome

---

- Explain the OO concepts of classes, objects, methods and messages
- Implement a class with instance variables, instance methods and constructors
- Explain the concept of abstraction and encapsulation
- Construct a program using classes, objects, methods and messages



## Procedural vs Object Oriented

### Procedural Programming

- Centered on the procedures or action that take place in a program
- Procedures (or functions) and data are separated

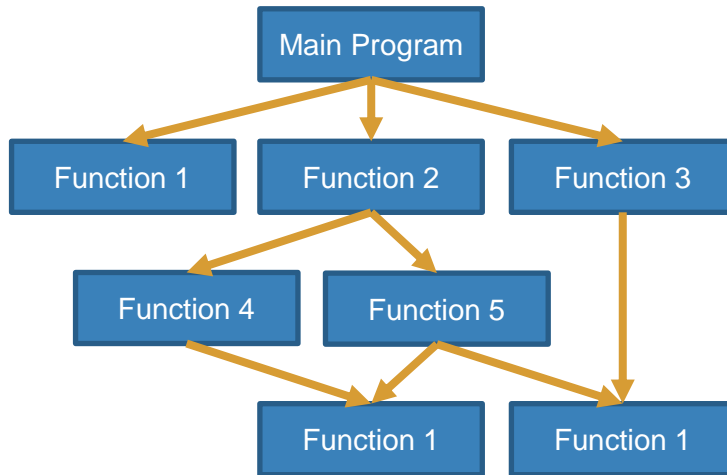
### Object-Oriented Programming

- Centered on objects that are created from abstract data types that encapsulate data and functions together

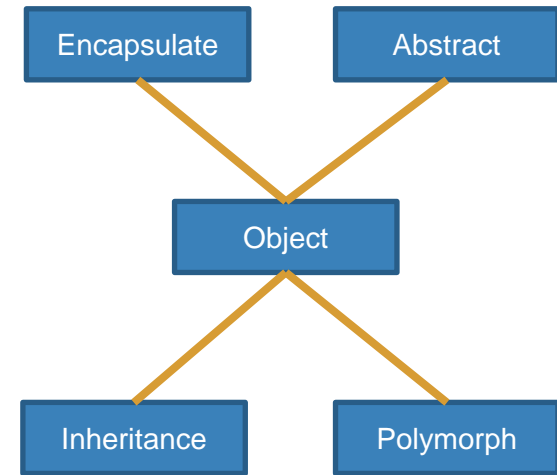


## Procedural vs Object Oriented

### Procedural Programming



### Object-Oriented Programming





## Procedural Programming

```
weight = input('What is your weight?')  
height = input('What is your height?')  
bmi = float(weight) / float(height) ** 2  
print(bmi)
```

Focus of procedural programming is on the creation of procedures that operate on the program's data.

### functions

input()

input()

float()

float()

print()

### data

'What is your weight?'

'What is your height?'

weight

height

bmi

As the procedural  
program becomes  
larger, your program  
becomes more  
complex and harder  
to change







## Object-Oriented Programming

```
class Person:
    def __init__(self, weight, height):
        self.__weight = weight
        self.__height = height
    def get_bmi(self):
        return self.__weight / self.__height ** 2

p = Person(71, 1.76)
print(p.get_bmi())
```

Centered on creating objects that contains both data (attributes) and procedures (methods).

### methods

`__init__()`

`get_bmi()`

### parameter variable

`self, weight, height`

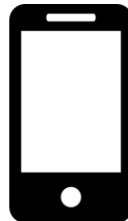
`self`

### attributes

`__weight`

`__height`

Object Oriented Programming addresses the problem of code and data separation through encapsulation and data hiding.







## Why OOP?

### Object Reusability

Objects are abstracted, and can be reused in other projects, speed up development process



### Maintainability

Codes are modular and bugs can be discovered and fixed more easily



### Extensibility

New objects can be added with minimum impact to existing objects

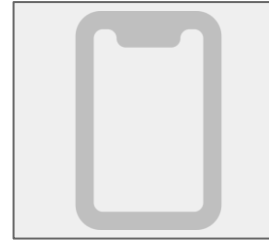


# Classes & Objects



A **class** is a code that specifies the data attributes and methods for a particular type of **object**.

A class is a blueprint, that objects may be created from.





## Class Definitions

```
class Customer:  
    def __init__(self, name, email):  
        self.__name = name  
        self.__email = email
```

A **class** definition is a set of statements that define a class's methods and data attributes.

methods

`__init__()`

attributes

`__name`

`__email`

The `__init__` method is usually the first method inside a class definition.

It executes automatically when an instance of the class is created in memory.



## What is self?

```
class Customer:  
    def __init__(self, name, email):  
        self.__name = name  
        self.__email = email
```

All methods, including the initializer must have the required **self** parameter variable.

Immediately after an object is created in memory, the **\_\_init\_\_** method executes, and the **self** parameter is automatically assigned the object that was just created.



## Things to note

### **self**

When defining your class method, you must explicitly list self as the first argument

When you call the method from outside the class, python automatically adds the self instance reference for you.

### **\_\_init\_\_**

The initializer is optional, but if defined, it will be called automatically after an instance is created.

You can **define** multiple initializer with different parameters but the **last one will override the earlier definitions**

# Lifecycle of Classes and Objects







## Working with instances

Each instance has its own set of data attributes

Use the self parameter to create an instance attribute

Can create many instances of the same class in a program

```
class Customer:
    def set_name(self, name):
        self.__name = name
    def set_email(self, email):
        self.__email = email
    def get_customer_info(self):
        return "Name: " + self.__name + ", Email: " + self.__email
```

```
# create c1 instance from Customer class
c1 = Customer()
c1.set_name("Ah Kaw")
c1.set_email("ahkaw@gmail.com")
```

```
# create c2 instance from Customer class
c2 = Customer()
c2.set_name("Ah Hua")
c2.set_email("ahhua@gmail.com")
```



## Life cycle of an instance

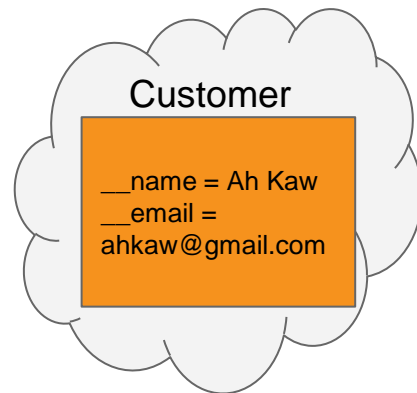
```
class Customer:
    def __init__(self, name, email):
        self.__name = name
        self.__email = email
    def get_customer_info(self):
        return "Name: " + self.__name + ", Email: " +
            self.__email
```

An object is created in memory from the Customer class

```
c = Customer("Ah  
Kaw",  
"ahkaw@gmail.com")
```

The `__init__` initializer is called, and the `self` parameter is set to the newly created object

A Customer object will exist with its `__name` and `__email` attributes set to Ah Kaw and ahkaw@gmail.com





## Life cycle of an instance

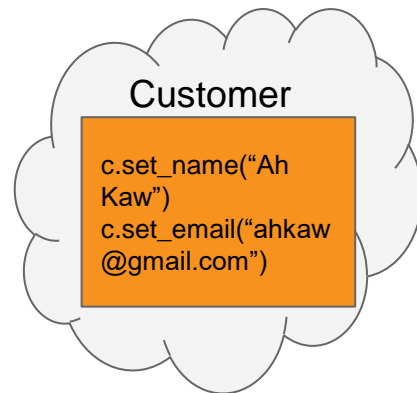
```
class Customer:
    def set_name(self, name):
        self.__name = name
    def set_email(self, email):
        self.__email = email
    def get_customer_info(self):
        return "Name: " + self.__name + ", Email: " + self.__email
```

An object is created in memory from the Customer class

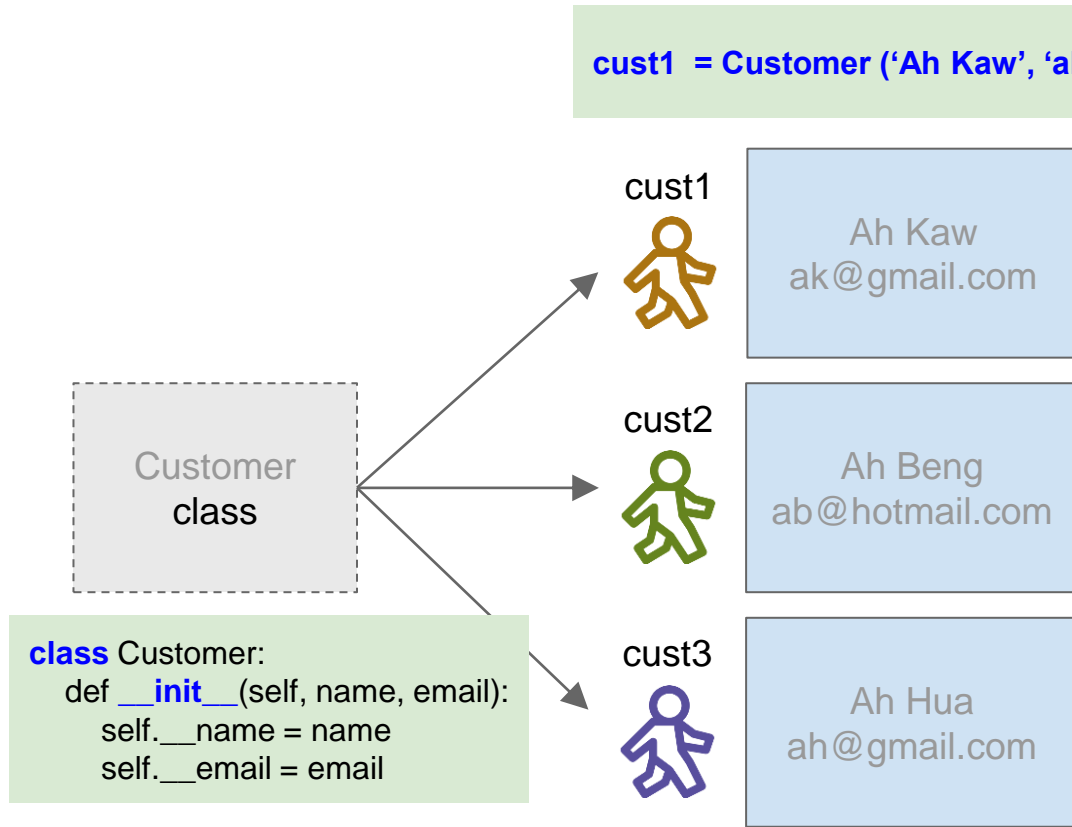
There is no `__init__` initializer called since it is not provided

A Customer object will exist however no attributes are created at this stage until the `set_name` or `set_email` method is called.

```
c = Customer()
```



The **Customer** class describes the data attributes and methods of a particular type of object may have.

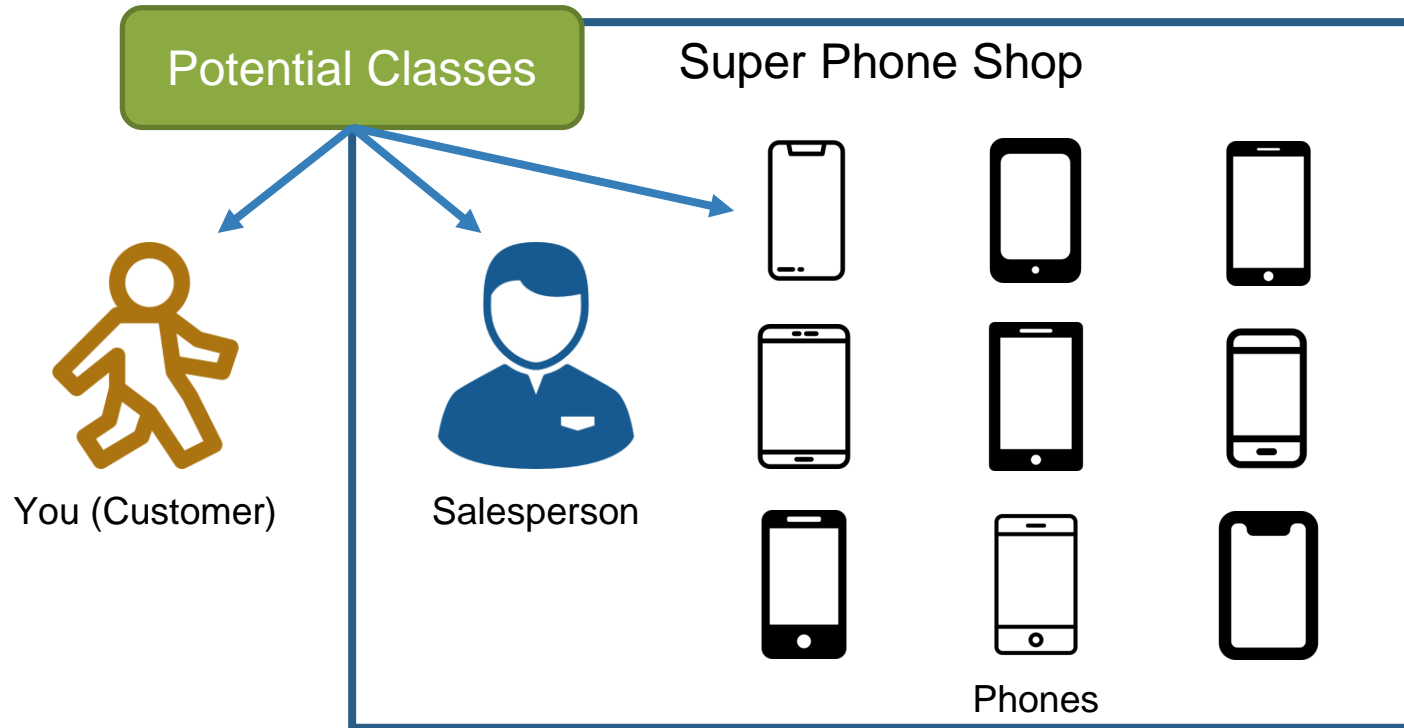


```
cust1 = Customer('Ah Kaw', 'ak@gmail.com')
```

The **cust1**, **cust2**, **cust3** objects are instances of the **Customer** class. It has the data attributes and methods described by the **Customer** class.



## Scenario – Buying a Phone



# Attributes of a Class





# Unified Modelling Language

## UML

Provides a standard diagrams for graphically depicting object-oriented system.

A class is represented with a box that is divided into three sections

*Class name goes here*

*Data attributes listed here*

*Methods listed here*





## Designing UML Diagram



Data attributes are values that define the state of the phone

Each method manipulates one or more of the data attributes

### attributes

make

model

camera

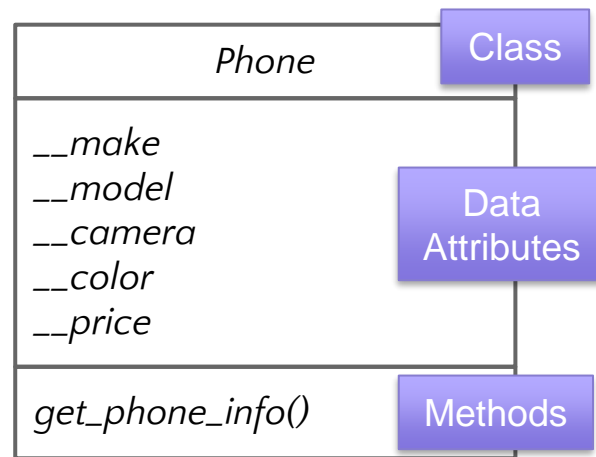
color

price

### methods

get\_phone\_info()

### UML diagram for Phone class?





## Data Attributes

### Identify Attributes

Every class has their own attributes, lets identify them!

Customer
<code>--name</code> <code>--email</code> <code>--mobile_number</code>

Salesperson
<code>--name</code>

Phone
<code>--make</code> <code>--model</code> <code>--price</code>

# Data vs Class Attributes





## Data vs Class Attributes

### Data attributes

Data attributes are pieces of data held by a specific instance of a class (object). To reference this attribute from code outside the class, you qualify it with the instance name

*e.g. c.count2*

### Class attributes

Class attributes are variables owned by the class itself. To reference this attribute from code outside the class, you qualify it with the class name

*e.g. Counter.count1*



## Data vs Class Attributes

```
class Counter:
    count1 = 0
    def __init__(self):
        self.count2 = 0

c = Counter()
c.count2 += 1
Counter.count1 += 5
```

Which one of these is a **class** attribute? **count1**

Which one of these is a **data** attribute? **count2**

To reference a **data** attribute from code outside the class, you qualify it with the **instance** name  
To reference a **class** attribute from code outside the class, you qualify it with the **class** name



## Data vs Class Attributes

```
class Counter:
    count1 = 0
    def __init__(self):
        self.count2 = 0
    def increase_count2(self):
        self.count2 += 1
    def increase_count1(self):
        self.__class__.count1 += 1
```

To reference a **data attribute** from code inside the class, you qualify it with **self**

To reference a **class attribute** from code inside the class, you qualify it with **self.\_\_class\_\_**

**\_\_class\_\_** is a built-in attribute of every class instance (of every class). It is a reference to the class that self is an instance of (in this case, the Counter class).



## Data vs Class Attributes

### Data attributes

Each instance of a class has its own set of data attributes.

### Class attributes

Class attributes are shared by all instances of a class.

```
class Counter:
    count1 = 0
    def __init__(self):
        self.count2 = 0
        self.count2 += 1
        self.__class__.count1 += 1
c1 = Counter()
c2 = Counter()
c3 = Counter()
print('Class variable %d, Data variable %d' % (Counter.count1, c1.count2))
```

```
count1 = 3 # class variable shared by all instances
count2 = 1 # data variable has its own set of data attributes
```



# Encapsulation





# Encapsulation

## Encapsulation

- hides internal representation of an object from the outside
- allows the access of attribute of an object to be controlled via methods



## Accessor methods

Also known as **getter**  
Provide a safe way for external code outside the class to retrieve the values of attributes

## Mutator methods

Also known as **setter**  
Control the way that an instance attribute value is modified



# Encapsulation



Customer

## attributes

name

email

mobile\_number

## methods

get\_name()

get\_email()

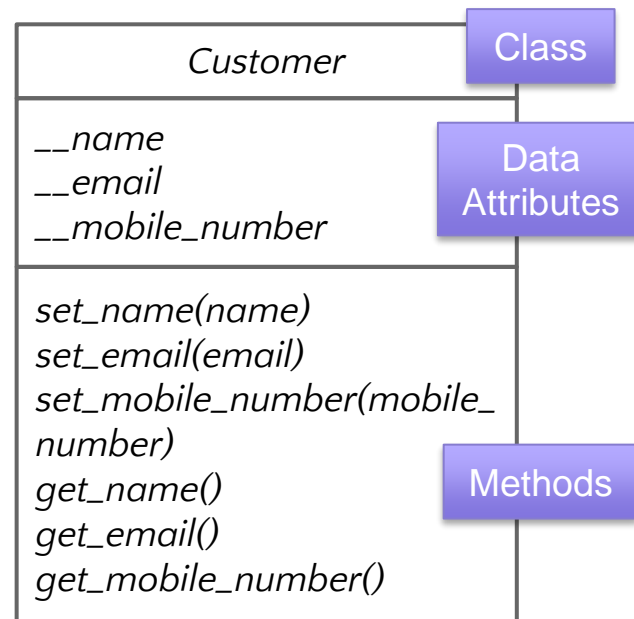
get\_mobile\_number()

set\_name(name)

set\_email(email)

set\_mobile\_number(m  
obile\_number)

UML diagram for Customer class?





## Encapsulation

```
class Customer:
    def set_name(self, name):
        self.__name = name
    def get_name(self):
        return self.__name

cust2 = Customer()
cust2.set_name('Ah Beng')
print(cust2.get_name())
```

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

cust1 = Customer('Ah Beng')
print(cust1.name)
```

`self.__name` and `self.name` are the attributes of the instance

Attribute `name` can be accessed externally directly from another program.

Attribute `__name` cannot be accessed externally from another program.

`name` is the passed in parameter from the calling program

Encapsulation



## Encapsulation

```
class Customer:
    def set_name(self, name):
        if name.isalpha():
            self.__name = name
        else:
            print('Only alphabets are allowed.')

    def get_name():
        return self.__name
```

```
cust1 = Customer()
cust1.set_name('Beng')
print('Name: ' + cust1.get_name())
```

Beng

```
cust2 = Customer()
cust2.set_name('123456')
print('Name: ' + cust2.get_name())
```

Only  
alphabets  
are allowed.

`isalpha()` is a built in function for testing whether string contains only alphabets.

For cust1, the name contains only alphabets, therefore the `name` will be printed.

For cust2, the name contains numbers, therefore **error message** will be printed.

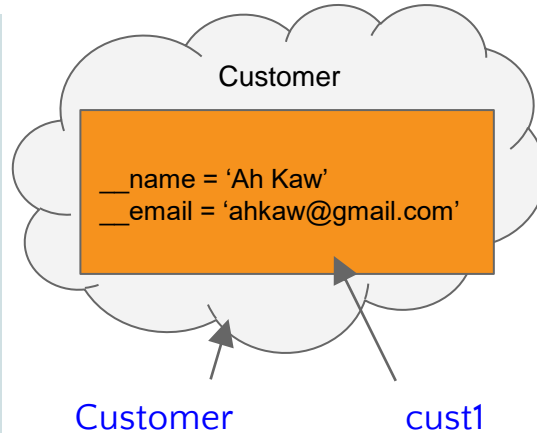
Validation



## Passing Objects as Arguments

```
def display_customer_info(customer):  
    print(customer.get_customer_info())
```

```
cust1 = Customer()  
cust1.set_name('Ah Kaw')  
cust1.set_email('ahkaw@gmail.com')  
display_customer_info(cust1)
```



```
class Customer:  
    def set_name(self, name):  
        self.__name = name  
    def set_email(self, email):  
        self.__email = email  
    def get_customer_info(self):  
        return self.__name, self.__email
```

When developing applications that work with objects, you often need to write functions and methods that accept objects as arguments.

When you pass an object as an argument, the thing that is passed into the parameter variable is a reference to the object.



## Recap: Identifying a class's responsibilities

A class's responsibilities are

- the things that the class is responsible for knowing
- the actions that the class is responsible for doing

### Customer class

#### Things to know

- customer's name
- customer's address
- customer's mobile

#### Actions to do

- initializer
- accessor / mutator methods
- methods



# Abstraction





## Abstraction

### Abstraction

- hides unnecessary details from users
- allows implementation of more complex logic without having the need to understand the hidden details

```
class Customer:
    def set_name(self, name):
        self.__name = name

    def set_email(self, email):
        self.__email = email

    def get_customer_info(self):
        return 'Name: ' + self.__name + ', Email: ' + self.__email

c = Customer()
c.set_name('Ah Kaw')
c.set_email('ahkaw@gmail.com')
print(c.get_customer_info())
```

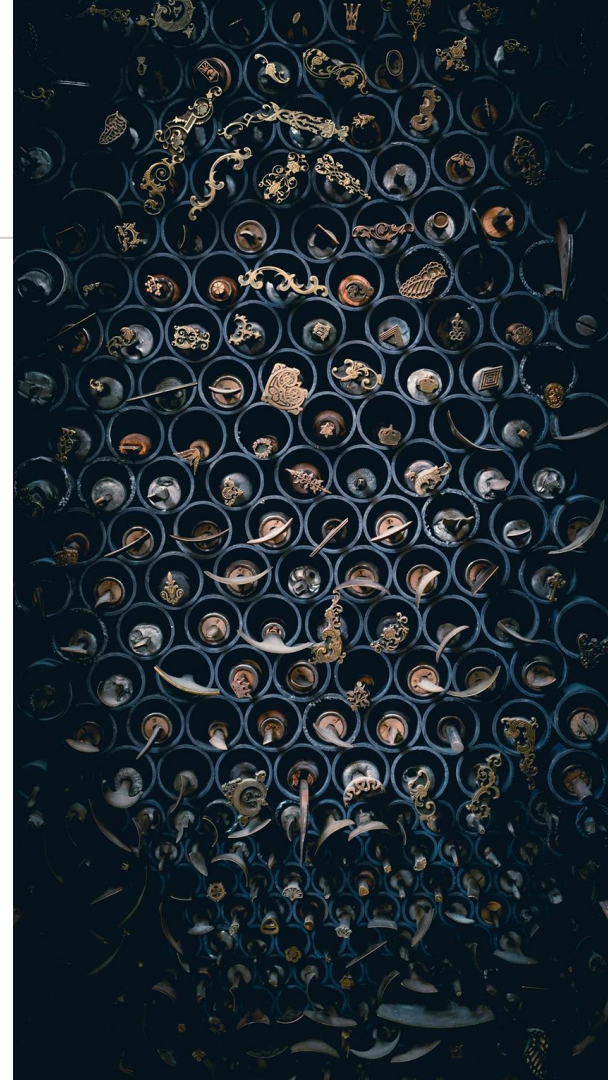


## Storing Classes in Modules

Organize class definitions by storing them in modules in a separate file.  
Import modules into any program that need to use the classes they contain.

```
import random
```

```
if random.randint(0,1) == 0:  
    print('Head')  
else:  
    print('Tail')
```



### # Stored in Customer.py

```
class Customer:
    def __init__(self, name, email):
        self.__name = name
        self.__email = email
    def get_name(self):
        return self.__name
    def get_email(self):
        return self.__email
    def get_customer_info(self):
        return 'Name: ' + self.__name + ', Email: ' +
            self.__email
```

How do you create a Customer instance from Customer class that is stored in a module?

### Three ways to import Customer module

1. `import Customer`
2. `from Customer import *`
3. `import Customer as c`

```
cust1 = Customer.Customer('Ah Kaw', 'ahkaw@gmail.com')
```

```
cust1 = Customer('Ah Kaw', 'ahkaw@gmail.com')
```

```
cust1 = c.Customer('Ah Kaw', 'ahkaw@gmail.com')
```



## Built-in Function

### `__str__`

A built-in function used for string representation of object

```
class Customer:
    def __init__(self, name, email):
        self.__name = name
        self.__email = email
    def __str__(self):
        s='Name: {}, Email: {}'.format(self.__name, self.__email)
        return s
```

```
cust1 = Customer('Ah Kaw', 'ahkaw@gmail.com')
print(cust1)
cust2 = Customer('Ah Hua', 'ahhua@gmail.com')
print(cust2)
```

Output:

Name: Ah Kaw, Email: ahkaw@gmail.com

Name: Ah Hua, Email: ahhua@gmail.com



## Checkpoint : class or object?

1	___ is a blueprint from which _____ are created.	_____ is an instance of a _____.
2	_____ is a real world entity such as pen, laptop, mobile, bed, keyboard, mouse, chair etc.	_____ is a group of similar _____.
3	_____ is a logical entity.	_____ is a physical entity.



## Checkpoint : class or object?

4	_____ is created many times as per requirement.	_____ is declared once.
5	_____ allocates memory when it is created.	_____ doesn't allocated memory when it is created.
6	There is only one way to define _____ in python	There are many ways to create _____ in python.



## Checkpoint : abstraction and encapsulation

	Question	True / False
7	Abstraction means that one knows about the complex logics of methods that are not implemented by you	
8	Encapsulation hides data attributes and allows validation before changing the values	





## Summary

---

- Explain the OO concepts of classes, objects, methods and messages
- Implement a class with instance variables, instance methods and constructors
- Explain the concept of abstraction and encapsulation
- Construct a program using classes, objects, methods and messages