Course 262

Advanced Python Programming

Course Outline

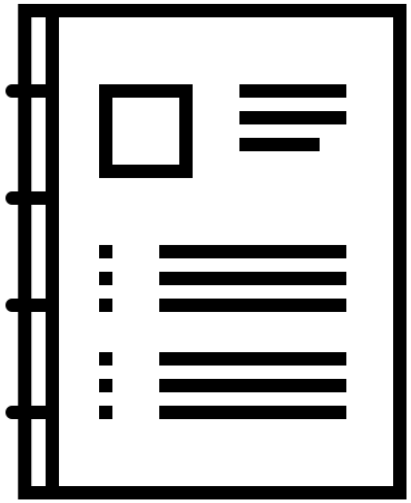
1. OOP
2. Decorators
3. Inheritance and Polymorphism
4. Exception Handling
5. Introduction to Django

Chapter 1

Object-Oriented Programming

Chapter Outline

- Object-Oriented Programming
- What is an Object?
- What is a Class?
- Instantiating Objects
- Defining Constructors
- Declaring and Initializing Attributes
- Accessing Attributes
- Declaring Methods
- Calling Methods
- Passing by Reference
- Comparing Objects



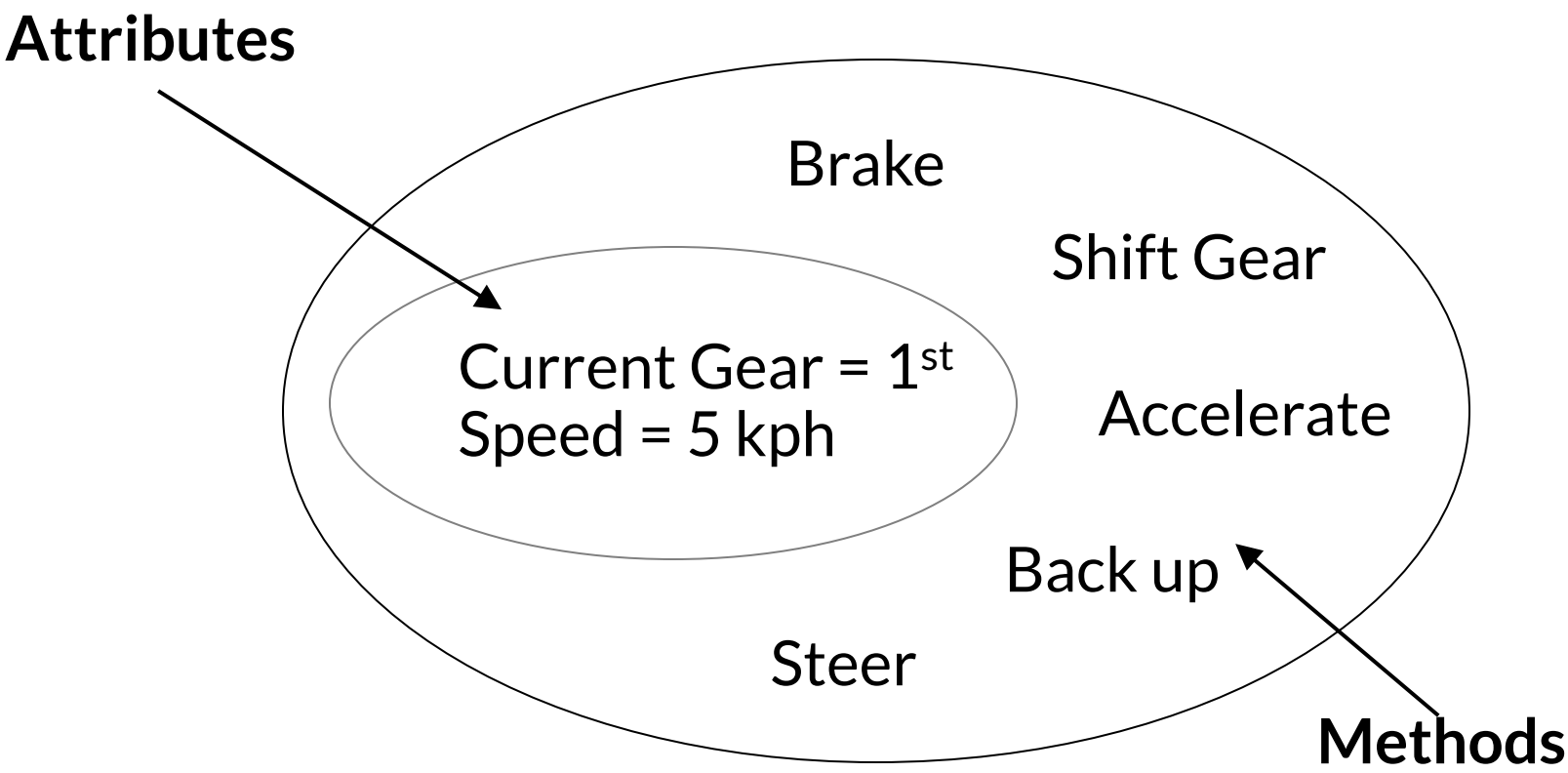
Object-Oriented Programming (OOP)

- A modular approach to computer programming and software design.
- An **object-oriented program** may be seen as composed of objects that interact with each other.
- A **procedural** or **structured** program is seen as a set of instructions to the computer.

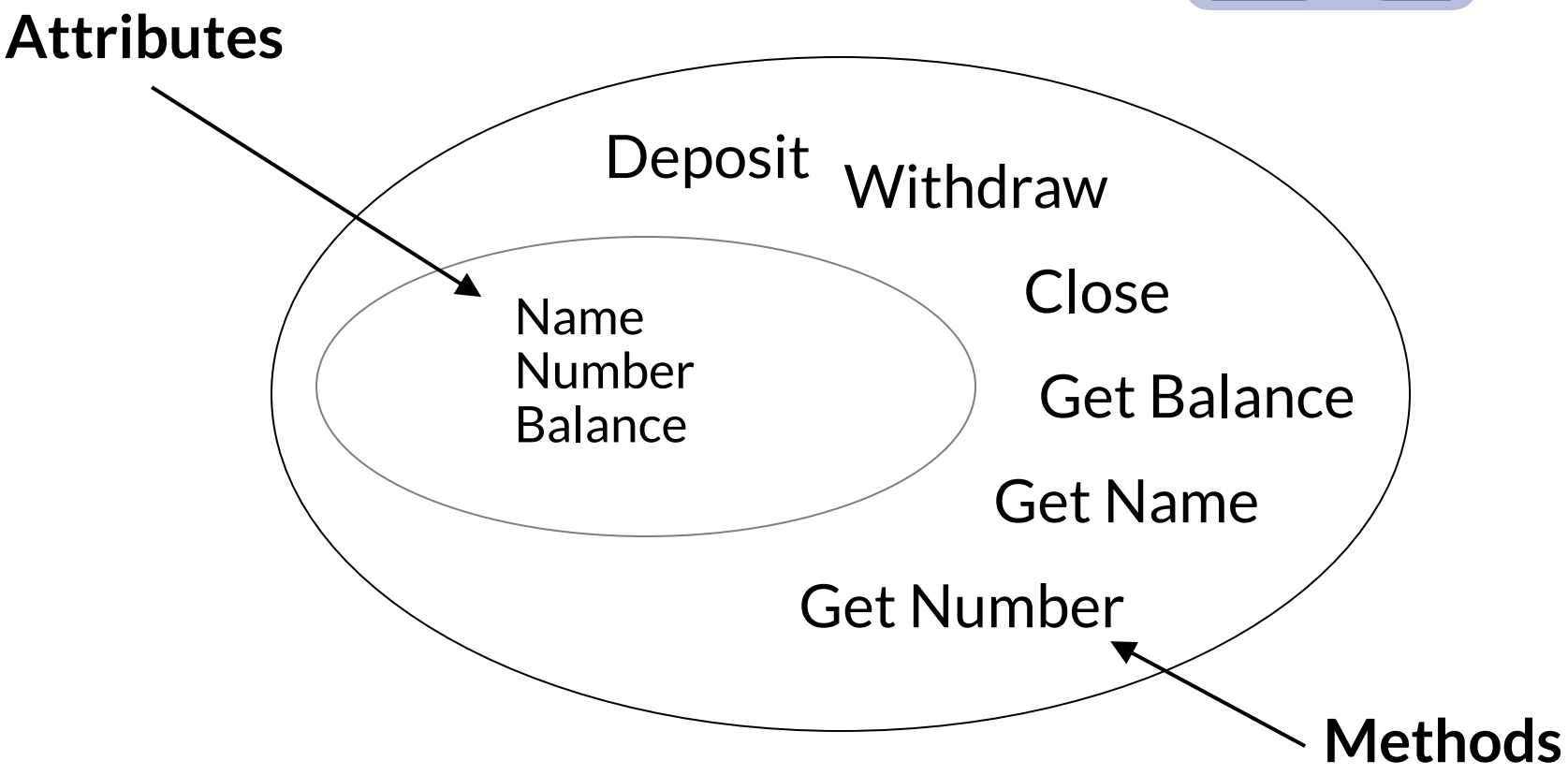
What is an Object?

- An object combines ***data*** *and* ***code*** that acts on that data
- May represent
  - Real world objects – Car, Person, Phone, Book, etc.
  - Concepts – Time, Bank account, Sale, etc.
- An object has
  - **state** = variables / data / attributes
  - **behavior** = operations / code / methods

Example: Car Object



Example: Bank Account Object



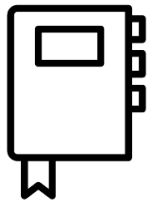
Classes

- A **class** is a ***blueprint*** or ***prototype*** that defines the attributes and methods common to all objects of a certain kind.
- Programmers can use a class repeatedly to create numerous objects.
- Classes vs. Objects
  - Class: Dog
  - Instances/Objects of Dog: Snoopy, Pluto

Classes vs Objects

	Car Class	Object Car A	Object Car B
Attributes	Plate Number	XBE 593	XJT 777
	Color	Silver	Dark Gray
	Current Gear	4th	3rd
	Current Speed	100	50
Methods	Accelerate		
	Brake		
	Shift Gear		

Terms



- **Instance** - an object created from a certain class
  - Analogy:
    - Class: Dog
    - Instance(s) of Dog: Snoopy, Pluto
- **Instantiate** – create a new object
- **Member** – general term referring to the contents of an object
  - **Attribute** – fields or properties
  - **Method** – functions or operations

Defining a Class

```
class ClassName:
    class body
```

```
class Car:
    def __init__(self):
        self.speed = 0
```

speed IS AN ATTRIBUTE OF Car

```
    def accelerate(self, kph):
        self.speed = self.speed + kph

    def brake(self):
        self.speed = 0
```

accelerate() AND brake() ARE METHODS OF Car

Coding Convention: Classes

- Class names should be:
  - nouns, in mixed case with the first letter of each internal word capitalized.
  - No underscores

```
BankTransaction, UserAccount
```

Instantiating Objects

```
object = ClassName()
```

Example:

```
class Car:
    pass

porsche = Car()
```

pass IS USED TO CREATE A CLASS WITHOUT CONTENTS

Classes vs Objects

```
class Car:
    def __init__(self):
        self.speed = 0

    def accelerate(self, kph):
        self.speed = self.speed + kph

    def brake(self):
        self.speed = 0
```

CLASS

```
porsche = Car()
bmw = Car()
```

OBJECTS OR INSTANCES OF `Car`

Constructors

- Special method that is executed when an object is created
- Always named `__init__()`
  - That's 2 underscores (_) on both ends

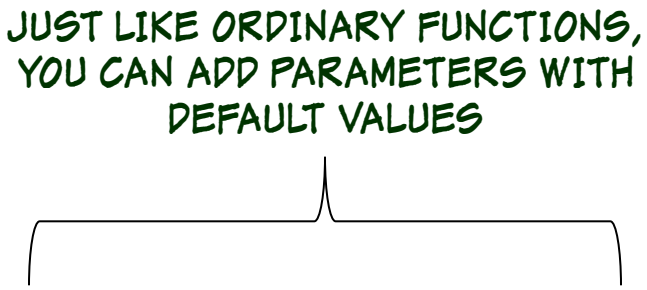
```
class ClassName:
    def __init()__(self [, param1, param2, …]):
        constructor body
```

- **TIP:** *"Dunder"* is the shortcut term used for "double underscores" ☺
  - You will often hear the above referred to us *"dunder init"*

Defining Constructors

```
class Car:
    def __init__(self):
        self.speed = 0
        self.gear = 'P'

porsche = Car()  ──────► RUNS Car.__init__()

print(porsche.speed)     # 0
print(porsche.gear)      # 'P'
```

Defining Constructors (cont'd)




```
class Car:
    def __init__(self, speed = 0, gear = 'P'):
        self.speed = speed
        self.gear = gear

porsche = Car(200, 'D')
print(porsche.speed)       # 200
print(porsche.gear)        # D

bmw = Car()
print(bmw.speed)           # 0
print(bmw.gear)            # P
```

Side Bar: The `self` Variable

- `self`
  - a special keyword referring to the instance currently in use

```
class Car:
    def __init__(self, speed = 0, gear = 'P'):
        self.speed = speed
        self.gear = gear

porsche = Car(200, 'D')
print(porsche.speed)       # 200
print(porsche.gear)        # D
```

IN THIS EXAMPLE, `self` REPRESENTS THE INSTANCE `porsche`.

IT'S LIKE SAYING

```
porsche.speed = speed
porsche.gear = gear
```

Side Bar: Printing Objects using __dict__

- __dict__
  - a special dictionary that exists for each Python object
  - contains the attributes of that object and their values

```
class Car:
    def __init__(self, speed = 0, gear = 'P'):
        self.speed = speed
        self.gear = gear

porsche = Car(200, 'D')
print(porsche.__dict__)
```

- Output:

```
{'speed': 200, 'gear': 'D'}
```

Declaring and Initializing Attributes

- In Python, you create attributes by initializing them inside the constructor

```
class Car:
    def __init__(self):
        self.speed = 0
        self.gear = 'P'
```

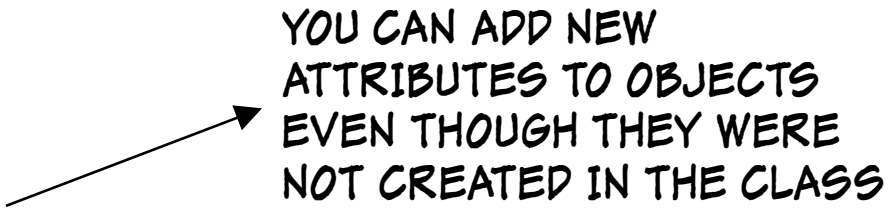
Accessing Attributes

From outside a class:

```
object_name.attribute
```

Example:

```
class Car:
    def __init__(self):
        self.speed = 0
        self.gear = 'P'

porsche = Car()
porsche.mileage = 2500
print(porsche.mileage)      # 2500
print(porsche.gear)         # P
```



Accessing Attributes within the Same Class

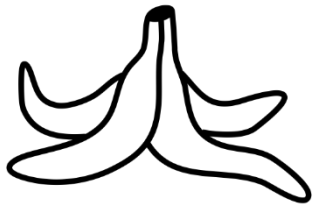
From within a class:

```
self.attribute
```

Example:

```
class Car:
    def __init__(self):
        self.speed = 0

    def accelerate(self, kph):
        self.speed = self.speed + kph

porsche = Car()
porsche.accelerate(50)
porsche.accelerate(20)
print(porsche.speed)          # 70
```

Common Pitfall: Accessing Missing Attributes



For you to access attributes within the same class, you have to create the attributes in the constructor.

```
class Car:
    def accelerate(self, kph):
        self.speed = self.speed + kph
```

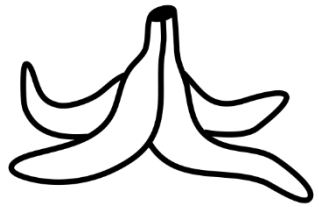
```
AttributeError: object has no
attribute 'speed'
```

Declaring Methods

```
class ClassName:
    def function_name(self [, param1, param2, …]):
        method body
```

```
class Car:
    def __init__(self, speed = 0, gear = 'P'):
        self.speed = speed
        self.gear = gear

    def accelerate(self, kph):
        self.speed = self.speed + kph
        self.gear = 'D'
```

Common Pitfall: Forgetting the `self` Parameter



- A Python method should always have `self` as the first parameter.


```
class Car:
    def __init__(speed = 0, gear = 'P'):
        self.speed = speed
        self.gear = gear
```

Calling Methods

```
object_name.method_name()
object_name.method_name(arguments)
```

```
class Car:
    def __init__(self, speed = 0, gear = 'P'):
        self.speed = speed
        self.gear = gear

    def accelerate(self, kph):
        self.speed = self.speed + kph
        self.gear = 'D'

porsche = Car()
porsche.accelerate(50)
print(porsche.speed)      # 50
print(porsche.gear)       # D
```

Calling Methods (cont'd)

- To further illustrate, let's use a string
- A Python string is also an object, having its own methods
- Let's take a look at string's documentation for `find()` method

`str.` **`find`**(*sub*[, *start*[, *end*]])

Return the lowest index in the string where substring *sub* is found within the slice `s[start:end]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` if *sub* is not found.

```
s = "hello"
print(s.find('e'))              # 1
print(s.find('e', 2))           # -1 (meaning not found)
```

- **TIP:** You can find the Python Standard Library reference documentation at *https://docs.python.org/3/library/*

Calling Methods From Within the Same Class

```
self.method_name()
self.method_name(arguments)
```

```
class Car:
    def __init__(self, speed = 0, gear = 'P'):
        self.speed = speed
        self.gear = gear

    def set_speed(self, speed):
        self.speed = speed

    def brake(self):
        self.set_speed(0)
```

Object Pointers

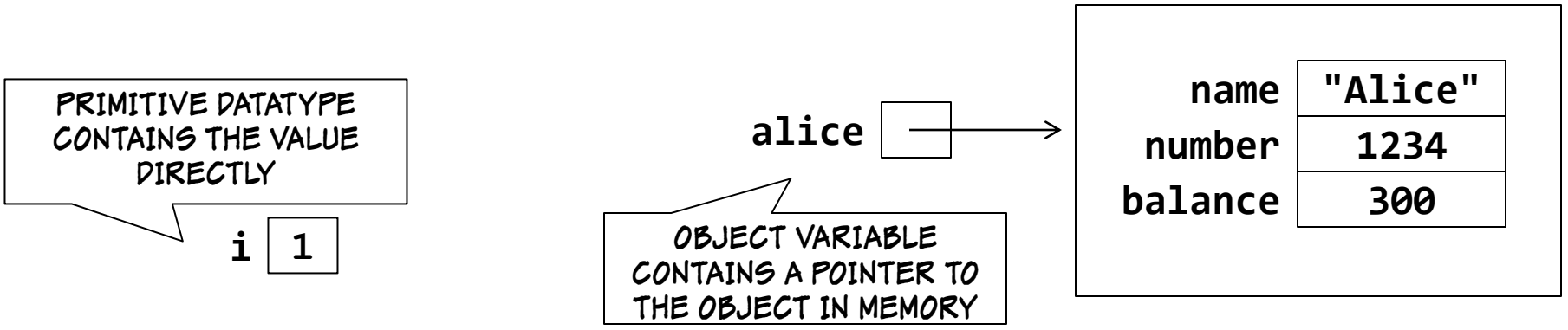
- For the rest of this chapter, we will use the following class in our examples.

```
class Account:
    def __init__(self, number, name, balance = 0):
        self.number = number
        self.name = name
        self.balance = balance
```

Object Pointers (cont'd)

- Internally, your object variable contains a reference to your object in memory instead of the actual values
- To illustrate, consider these two variables

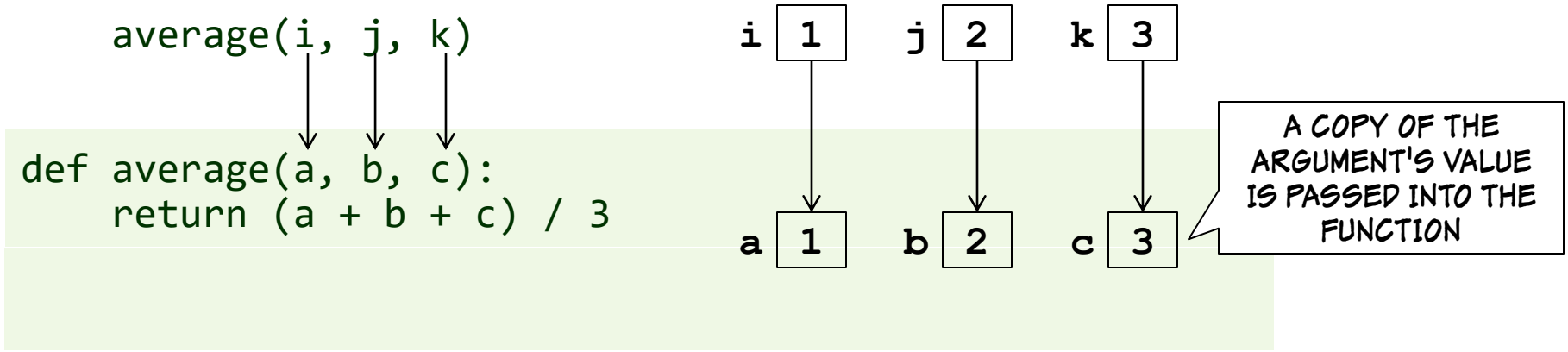
```
i = 1
alice = Account(1234, 'Alice', 300)
```



Passing by Reference

- In an earlier chapter, we talked about how arguments are passed into functions

```
i = 1; j = 2; k = 3
average(i, j, k)
```



- Passing objects is slightly more complicated than passing primitive data types

Passing by Reference (cont'd)

- Consider the following function which changes the value of a variable and of the `balance` attribute of an `Account` object

```
def parameterTest(a, account):
    a = a + 1
    account.balance = account.balance * 2
```

Passing by Reference (cont'd)

- Passing an integer and an `Account` object into the function, we see our integer does not change, but `alice.balance` does

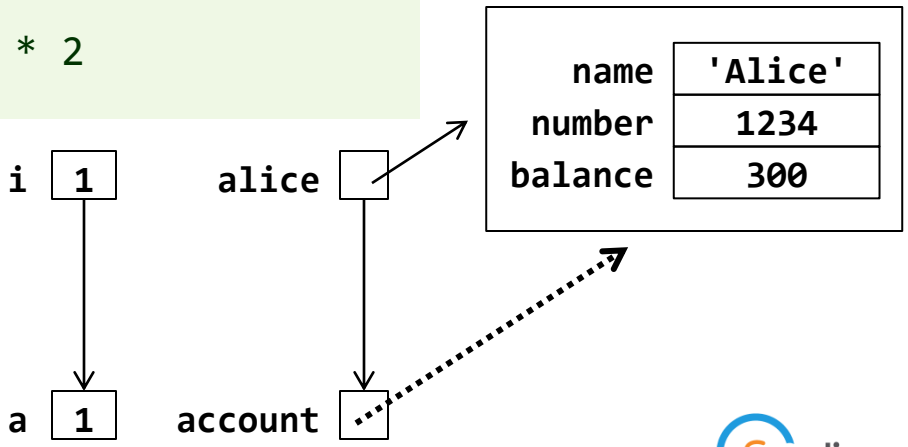
```
def parameterTest(a, account):
    a = a + 1
    account.balance = account.balance * 2


alice = Account(1234, 'Alice', 300)

i = 1
parameterTest(i, alice)
print(i)                    # 1 (unchanged)
print(alice.balance)        # 600 (changed)
```

Passing by Reference (cont'd)

- An object variable contains a pointer to the actual object. When passed into a function, the function's parameter will refer to the same object
  - Any changes made inside will reflect outside

```
parameterTest(i, alice);
```

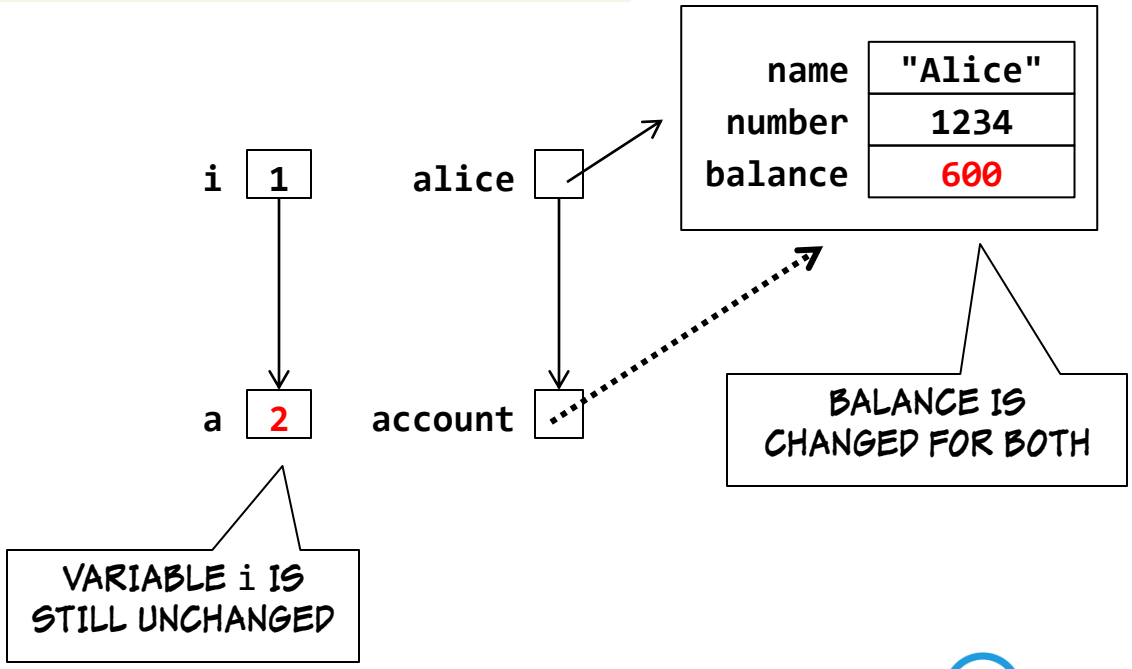
```
def parameterTest(a, account):
    a = a + 1
    account.balance = account.balance * 2
```



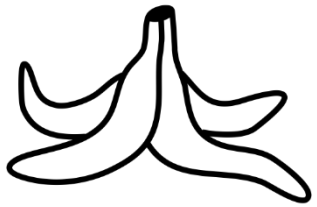
Passing by Reference (cont'd)

```
    parameterTest(i, alice);
```

```
def parameterTest(a, account):
    a = a + 1
    account.balance = account.balance * 2
```

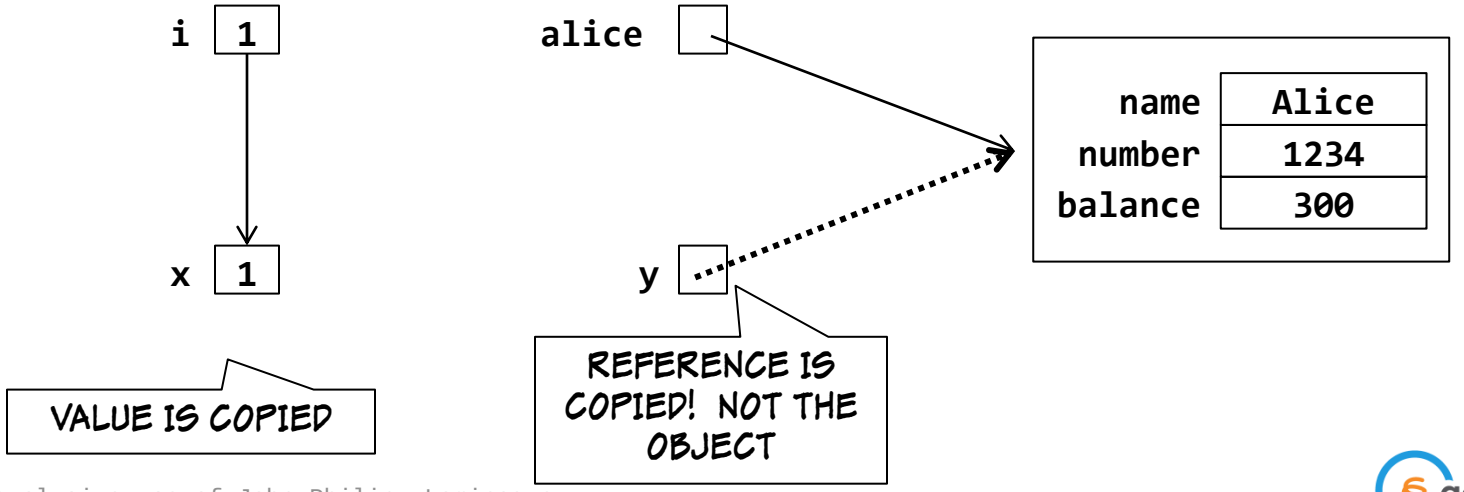


Common Pitfall: Copying an Object, But Not Really!

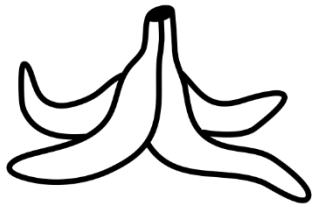


- Be careful when assigning objects to other variables

```
i = 1
x = i
alice = Account(1234, 'Alice', 300)
y = alice
```

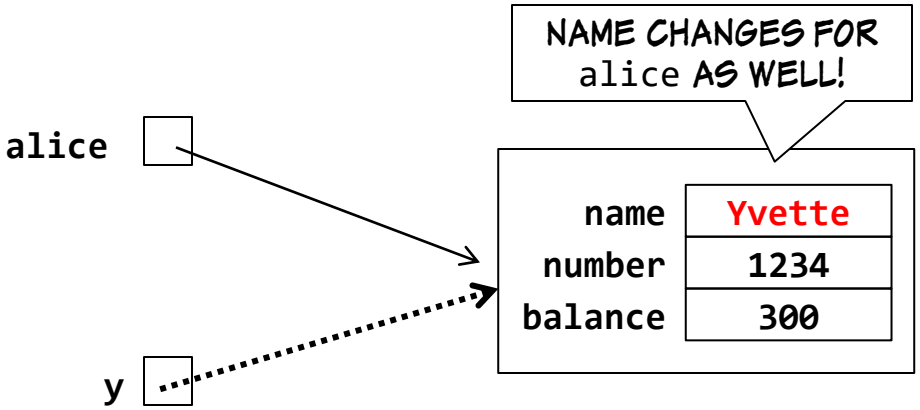


Common Pitfall: Copying an Object, But Not Really!



- Only the reference is copied, not the object itself!

```
alice = Account(1234, 'Alice', 300)
y = alice
y.name = 'Yvette'
print(alice.name)     # Yvette
```



Comparing Objects

- When == and != are applied to objects, they determine whether both sides of the operator refer to the same object.
- They do not check the values of the objects.

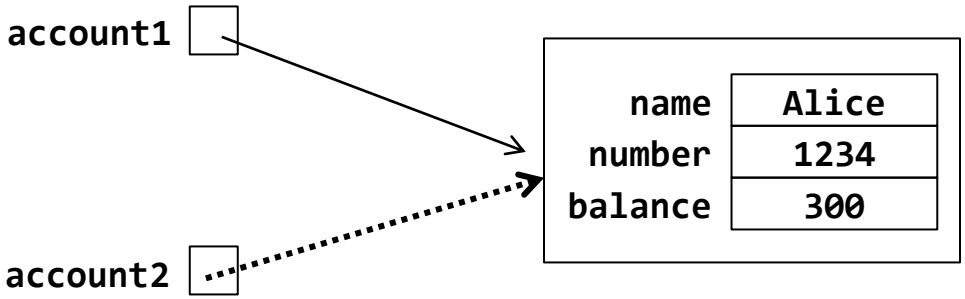
```
account1 = Account(1234, 'Alice', 300)

account2 = Account(1234, 'Alice', 300)

print(account1 == account2)    # False
```

NOTE THAT EVEN THOUGH account1 AND account2 HAVE THE SAME VALUES, THEY ARE STILL 2 DIFFERENT OBJECTS

Comparing Objects (cont'd)

```
account1 = Account(1234, 'Alice', 300)

account2 = account1

print(account1 == account2)     # True
```

THIS TIME THE COMPARISON RETURNS TRUE BECAUSE account1 AND account2 ARE REFERRING TO THE SAME OBJECT

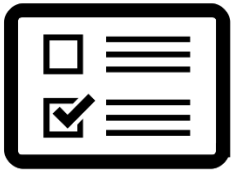


Comparing Objects (cont'd)

- You can also use the **is** or **is not** operator to compare if two variables are referring to the same object

```
alice = Account(1234, 'Alice', 300)
yvette = alice
gavin = Account(5678, 'Gavin', 1000)

print(alice is yvette)     # True
print(alice is not gavin) # True
```

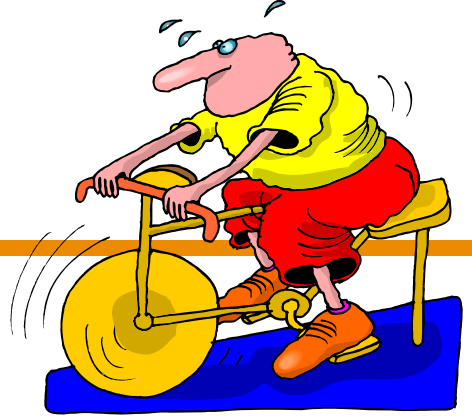
Test Your Knowledge



1. Variables that belong to a class are called _____
2. An object can also contain functions called _____
3. A _____ is a blueprint that defines objects of a certain kind
4. When an object is instantiated, what method of the class is automatically called?
5. You can print an object's attributes and their values using its _____ variable.
6. True or False. You can add attributes to an object that are not originally declared in its class
7. Methods should always have _____ as the first parameter.
8. When objects are passed to functions as parameters, they are passed by _____.

Answers

1. attributes or properties
2. methods
3. Class
4. Constructor or `__init__()`
5. `__dict__`
6. True
7. `self`
8. reference



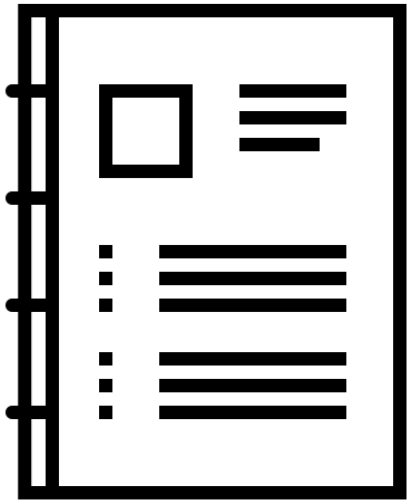
Please turn to Exercise 1 in your Exercise Manual

Chapter 2

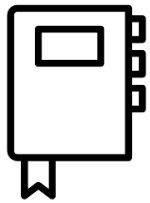
Decorators

Chapter Outline

- **Creating and Accessing Static Attributes**
- Creating and Accessing Class and Static Methods
- Creating Properties and Setter Methods

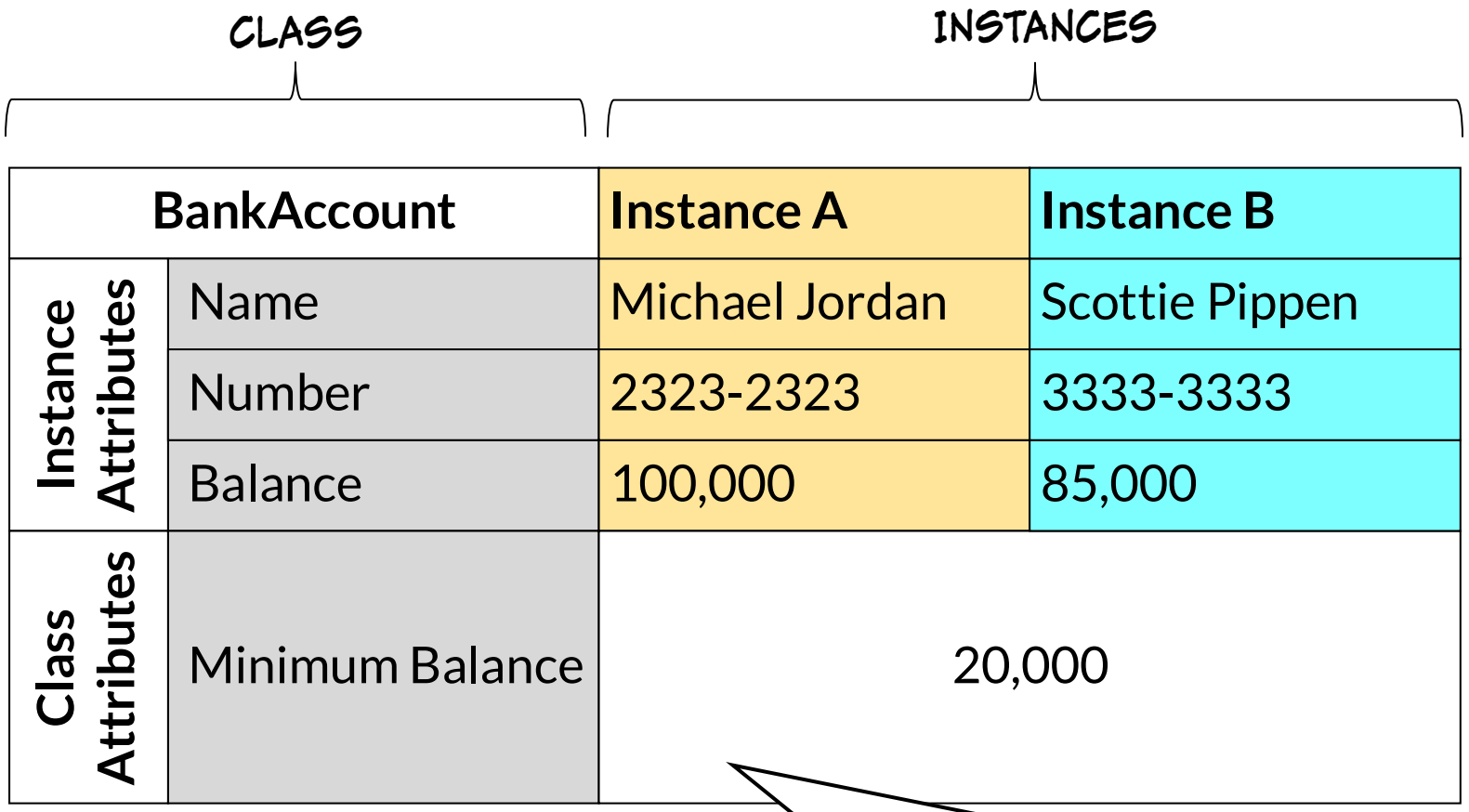


Static Attributes



- A static attribute does not belong to instances of the class, but rather it belongs to the class itself
- Also known as **Class variables**, **Class attributes**
- Allocated *once* regardless of how many objects are created

Static Attributes (cont'd)



Creating Static Attributes

- To create a static attribute, create the variable within the class, but outside any of the methods

```
class Account:
    min_balance = 10000

    def __init__(self, number, name, balance=0):
        self.number = number
        self.name = name
        self.balance = balance
```

Accessing Static Attributes

```
gavin = Account(1234, 'Gavin', 300)
print('gavin:', gavin.__dict__)
print('Account.min_balance:', Account.min_balance)
print('gavin.min_balance:', gavin.min_balance)
```

SINCE A STATIC ATTRIBUTE BELONGS TO THE CLASS, USE THE CLASS NAME TO ACCESS IT

Output:

```
gavin: {'number': 1234, 'name': 'Gavin', 'balance': 300}
Account.min_balance: 10000
gavin.min_balance: 10000
```

NOTICE THAT min_balance DOES NOT APPEAR IN gavin'S ATTRIBUTE LIST.

BUT WE WERE ABLE TO ACCESS IT USING Account.min_balance

WE ARE ALSO ABLE TO ACCESS IT USING THE INSTANCE

Accessing Static Attributes (cont'd)

- Python looks for attributes in the following priority:
  1. Instance attributes, before
  2. Class attributes

```
class Foo:
    x = 'static x'
    y = 'static y'

    def __init__(self):
        self.x = 'instance x'

foo = Foo()
print(foo.x)     # instance x
print(Foo.x)     # static x
print(foo.y)     # static y
```

Tips on Using Static Attributes

- Unless necessary, avoid using the same variable name for static and instance attributes to avoid confusion
- Even though you can access a static attribute using an instance variable, use the class name instead
  - Ex: `Account.min_balance` is preferred over `gavin.min_balance`

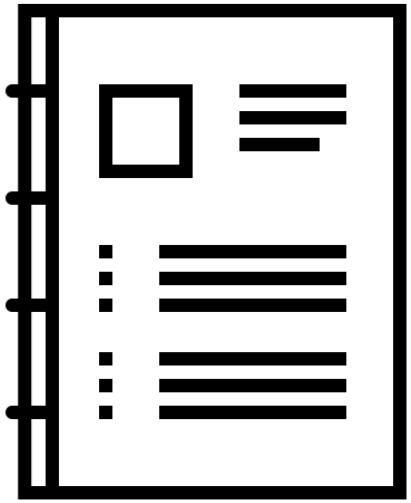
Accessing Static Attributes Within the Class

- To access a static attribute from within the class, use the class name instead of `self`

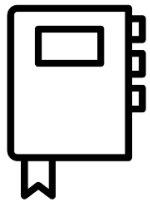
```
class Account:
    min_balance = 10000
    penalty = 100

    # __init__()

    def withdraw(self, amount):
        self.balance -= amount
        if self.balance < Account.min_balance:
            self.balance -= Account.penalty

account = Account(1234, 'Gavin', 30000)
account.withdraw(25000)
print(account.balance)      # 4900
```

Chapter Outline

- Creating and Accessing Static Attributes
- **Creating and Accessing Class and Static Methods**
- Creating Properties and Setter Methods



Class Methods



- Methods that can be invoked without instantiating the class
- To declare a class method, use the decorator `@classmethod`
- Must always have a single parameter representing the class
  - As a convention, we use `cls` and not `class` because the latter is a reserved word

```
class ClassName:

    @classmethod
    def method_name(cls):
        method body
```

NOT `self`, BECAUSE NOW, WE ARE PASSING THE CLASS ITSELF AND NOT AN INSTANCE OF THE CLASS.

Class Methods (cont'd)

- Let's create a method called `get_min_balance()` which will return the value of the static attribute `min_balance`.

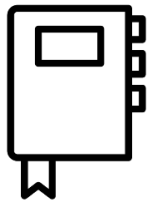
```
class Account:
    min_balance = 10000

    @classmethod
    def get_min_balance(cls):
        return cls.min_balance

print(Account.get_min_balance())
```

NOT `self`, BECAUSE NOW, WE ARE REFERRING TO THE `Account` CLASS. THIS PARAMETER IS REQUIRED FOR CLASS METHODS.

YOU CAN ALSO REFER TO THIS AS `Account.min_balance`

SINCE `get_min_balance()` IS NOW A CLASS METHOD, WE CAN CALL IT EVEN WITHOUT AN INSTANCE. IN EFFECT, WE ARE CALLING `get_min_balance(Account)`

Static Methods



- Static methods are just ordinary functions that are associated with a class because they have some logical connection with the class
- To declare a class method, use the decorator `@staticmethod`

```
class ClassName:

    @staticmethod
    def method_name(param1, param2, …):
        method body
```

NO self OR cls PARAMETER

Static Methods (cont'd)

- Let's create a method called `is_valid_account_number()` which will check if a given account number is valid.

```
class Account:
    min_balance = 10000

    @staticmethod
    def is_valid_account_number(num):
        account_number = str(num)
        if len(account_number) != 12:
            return False
        else:
            return True

print(Account.is_valid_account_number(123456789012)) # True
print(Account.is_valid_account_number(12345))        # False
```

NO self OR cls PARAMETER

Tips on Using Static and Class Methods

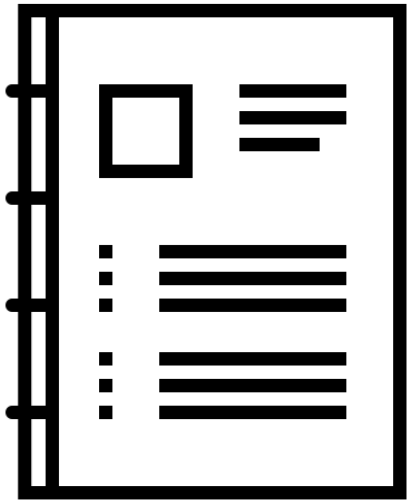
- Note that you can also use an instance variable to call static and class methods

```
Account gavin = Account(1234, 'Gavin Lim', 10000)
print(gavin.get_min_balance())                    # Valid
print(gavin.is_valid_account_number(12345))       # Valid
```

- Although this is valid, avoid using instance variables when calling static and class methods, as it can be deceiving
  - Use class name instead
    - `Account.get_min_balance()`
    - `Account.is_valid_account_number(12345)`

Chapter Outline

- Creating and Accessing Static Attributes
- Creating and Accessing Class and Static Methods
- **Creating Properties and Setter Methods**



Creating Properties

- Consider the following code:

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

- Let's create a method which will return the full name:

```
    def full_name(self):
        return '{} {}'.format(self.first_name, self.last_name)
```

- To get a Person's full name, we call `full_name()`

```
gavin = Person('Gavin', 'Lim')
print(gavin.full_name())    # Gavin Lim
```

Creating Properties (cont'd)

- We can convert `full_name()` to appear like an attribute through the `@property` decorator.

```
@property
def full_name(self):
    return '{} {}'.format(self.first_name, self.last_name)
```

- Now, we get a `Person`'s full name as if `full_name` is an attribute:

```
gavin = Person('Gavin', 'Lim')
print(gavin.full_name)    # Gavin Lim
```

Setter Methods

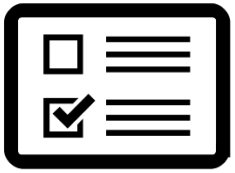
- Setter methods are used to change the value of a property
- Use the decorator *@property_name.setter*
- Let's create a setter for the `full_name` property

```
class Person:
    # code snipped

    @property
    def full_name(self):
        return '{} {}'.format(self.first_name, self.last_name)

    @full_name.setter
    def full_name(self, name):
        first, last = name.split(' ')
        self.first_name = first
        self.last_name = last
```

Setter Methods (cont'd)

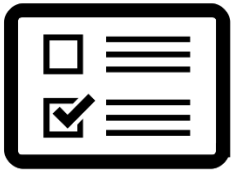
```
iron_man = Person('Tony', 'Stark')
print(iron_man.full_name)                    # Tony Stark
iron_man.full_name = 'Robert Downey'
print(iron_man.first_name)                   # Robert
print(iron_man.last_name)                    # Downey
```

Test Your Knowledge



1. True or false. To create a static attribute, you prefix the variable `@static` decorator upon creation.
2. True of false. Every instance has its own copy of a static attribute.
3. True or false. You can access class methods using an object variable and the class name.
4. The _____ decorator is used to make a method that appear as if it was an attribute that can be read.
5. Assuming you have a property named `score`, what would be the decorator for the corresponding setter method?

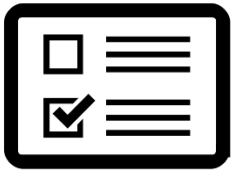
Test Your Knowledge



6. What is the output of the following code?

```
class Player:
    name = None

    @staticmethod
    def set_name(self, name):
        self.name = name

mike = Player()
mike.set_name('Michael Jordan')

print(Player.name)
```

Test Your Knowledge

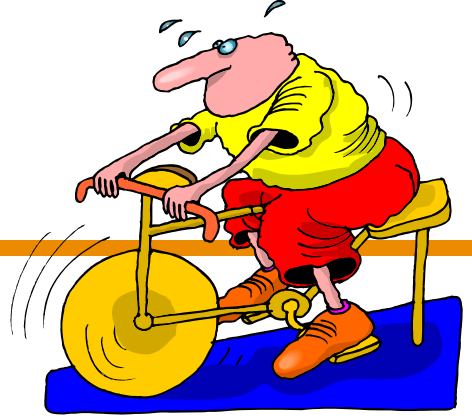


7. What is the output of the following code?

```
class Player:
    name = None

mike = Player()
mike.name = 'Michael Jordan'

pip = Player()
pip.name = 'Scottie Pippen'

print(Player.name)
```

Answers

1. False. You simply create the variable inside the class, but outside any methods.
2. False. A static attribute only has 1 copy, and it belongs to the class.
3. True
4. `@property`
5. `@score.setter`
6. Error because static methods cannot have self as a parameter
7. None



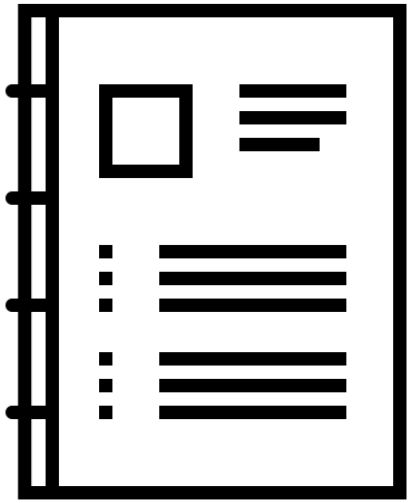
Please turn to Exercise 2 in your Exercise Manual

Chapter 3

Inheritance and Polymorphism

Chapter Outline

- **Inheritance**
- Method Overriding
- Polymorphism
- Member Visibility



Inheritance

- Suppose we have the following classes:
  - `TV`
  - `Tablet`
  - `MobilePhone`

Inheritance (cont'd)

```
class TV:
    def __init__(self):
        self.power_state = False

    def toggle_power(self):
        self.power_state = not self.power_state

    def chage_input(self):
        pass

class Tablet:
    def __init__(self):
        self.power_state = False

    def toggle_power(self):
        self.power_state = not self.power_state

    def browse(self, url):
        pass
```

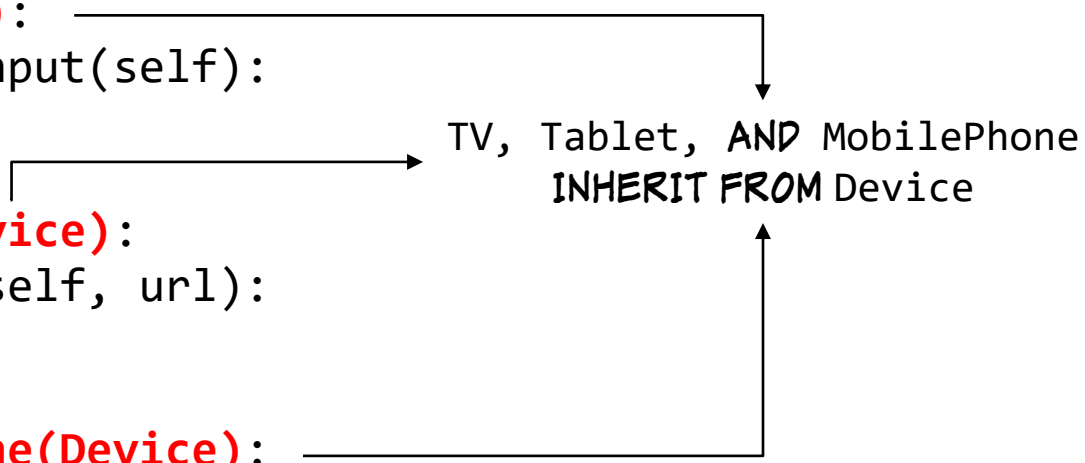
Inheritance (cont'd)

```
class MobilePhone:
    def __init__(self):
        self.power_state = False

    def toggle_power(self):
        self.power_state = not self.power_state

    def call(self, number):
        pass
```

- Notice the redundant code in all 3 classes
  - `power_state` attribute
  - `toggle_power()` method
- For this reason, we can refactor our code, so that all 3 classes inherit from a common class called `Device`.

Inheritance (cont'd)

```
class Device:
    def __init__(self):
        self.power_state = False

    def toggle_power(self):
        self.power_state = not self.power_state

class TV(Device):
    def chage_input(self):
        pass

class Tablet(Device):
    def browse(self, url):
        pass

class MobilePhone(Device):
    def call(self, number):
        pass
```



Inheritance (cont'd)

```
tv = TV()
tablet = Tablet()
tablet.toggle_power()
phone = MobilePhone()

print("tv:", tv.__dict__)
print("tablet:", tablet.__dict__)
print("phone:", phone.__dict__)
```

Output:

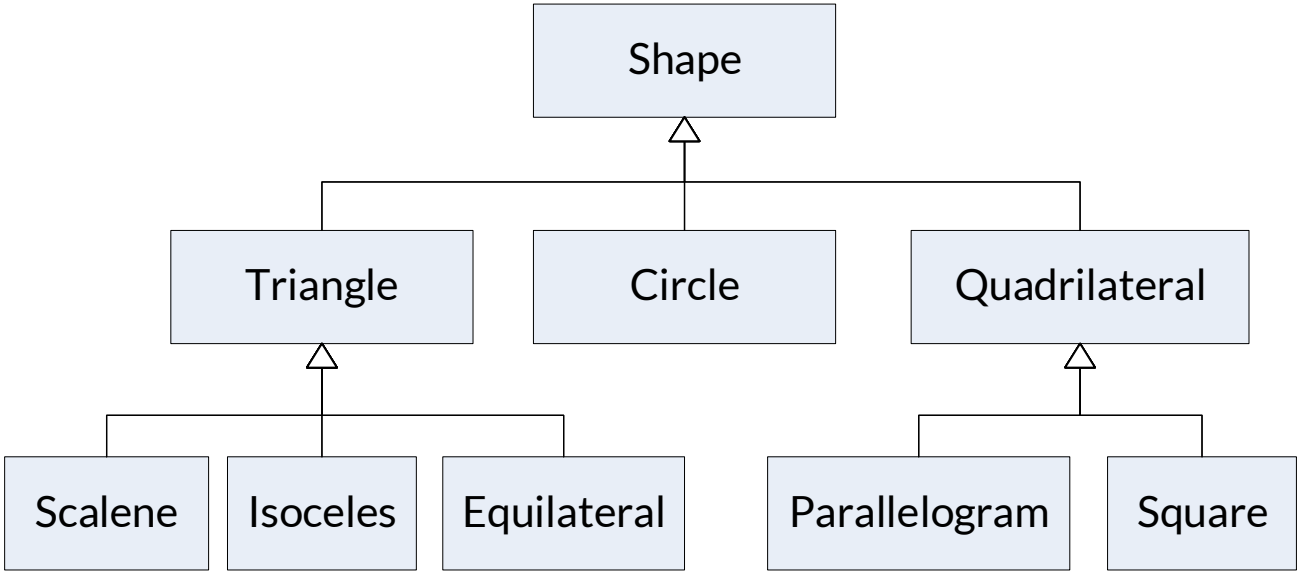
```
tv: {'power_state': False}
tablet: {'power_state': True}
phone: {'power_state': False}
```

NOTICE THAT ALL 3 OBJECTS INHERITED THE power_state ATTRIBUTE FROM DEVICE

Inheritance (cont'd)

- A feature of OOP that allows you to define a new class based on an existing class
- In OOP, inheritance is what is referred to as the *"is a"*, *"kind of"* or *"type of"* relationship.
- A *subclass* is said to *inherit* from its *superclass.*
- Terminology:
  - *Superclass*
    - any class above a specific class in the class hierarchy.
    - also called *ancestor*, *parent*, *or base* class
  - *Subclass*
    - any class below a specific class in the class hierarchy
    - also called *descendant, derived or child* class

Class Hierarchy



- Shape is a superclass of all other classes in the diagram.
- Scalene is a subclass of both Triangle and Shape.
- Parallelogram is a subclass of both Quadrilateral and Shape.
- Circle does not have any subclass.

Benefits of Inheritance

- Code Reuse
  - Once a behavior (method) is defined in a superclass, that behavior is automatically inherited by all subclasses
  - Thus, you can encode a method only once in the superclass and they can be used by all subclasses
  - A subclass can extend the behavior of its superclass by introducing new methods

Deriving a Subclass

- To derive a class, we use the following syntax:

```
class Subclass(Superclass):
    method body
```

- In order to illustrate this, let's create a `Person` class.
  - Attributes:
    - `first_name and last_name`
  - Method / Property:
    - `full_name()`

Person Class

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

    @property
    def full_name(self):
        return '{} {}'.format(self.first_name, self.last_name)
```

The `Employee` Class

- We want to create an `Employee` class.
  - Attributes:
    - `id`, `first_name`, `last_name`, `salary`
  - Methods / Properties:
    - `full_name()`
    - `pay()`
      - Returns the salary less 10% tax
- Since `Employee` and `Person` have common attributes and methods, let's just inherit from the `Person` class, and extend its behavior.

Employee Class (cont'd)

```
class Employee(Person):
    def __init__(self, id, first_name, last_name, salary=0):
        self.first_name = first_name
        self.last_name = last_name
        self.id = id
        self.salary = salary

def pay(self):
        tax = self.salary * .10
        return self.salary - tax
```

WE ADD THE NEW METHOD pay()

```
emp = Employee(123, 'Gavin', 'Lim', 50000)
print('Full Name:', emp.full_name)
print('ID:', emp.id)
print('Pay:', emp.pay())
```

NOTE THAT WE WERE ABLE TO ACCESS THE INHERITED PROPERTY full_name

```
Full Name: Gavin Lim
ID: 123
Salary: 45000.0
```

Calling a Superclass Constructor

- To reuse code, it's sometimes useful to explicitly call the superclass' constructor.
- To do this, you use:

```
super().__init__(arg1, arg2, …)
```

Calling a Superclass Constructor (cont'd)

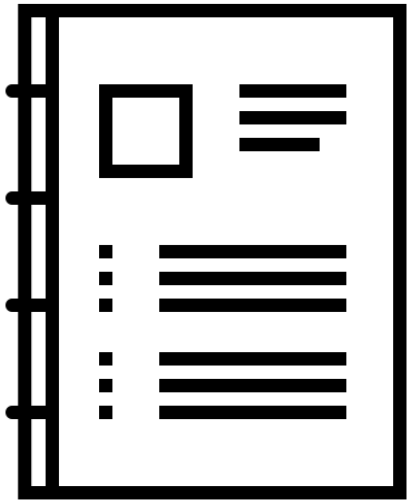
```
class Employee(Person):
    def __init__(self, id, first_name, last_name, salary=0):
        self.first_name = first_name
        self.last_name = last_name
        self.id = id
        self.salary = salary
```

```
class Employee(Person):
    def __init__(self, id, first_name, last_name, salary=0):
        super().__init__(first_name, last_name)
        self.id = id
        self.salary = salary
```

CALL `__init__()` OF THE `Person` SUPER CLASS TO SET THE EMPLOYEE'S FIRST NAME AND LAST NAME

Chapter Outline

- Inheritance
- **Method Overriding**
- Polymorphism
- Member Visibility



Method Overriding

- Redefining a method inherited from a superclass
- Sometimes it may be useful for a subclass to have a different method implementation than that of its superclass
- For example, we might create another class called `SalesEmployee`
  - Sales employees' pay are computed as follows:
    - pay = salary – 10% tax + commission
      - Assume that commission is not taxable
- We should then override the `pay()` method of `Employee`.

Method Overriding (cont'd)

```
class Employee(Person):
     # code snipped

     def pay(self):
         tax = self.salary * .10
         return self.salary - tax
```

```
class SalesEmployee(Employee):
    # code snipped

    def pay(self):
        tax = self.salary * .10
        return self.salary - tax + self.commission
```

WE REDEFINE pay() IN THE SUBCLASS. THIS IS OVERRIDING.

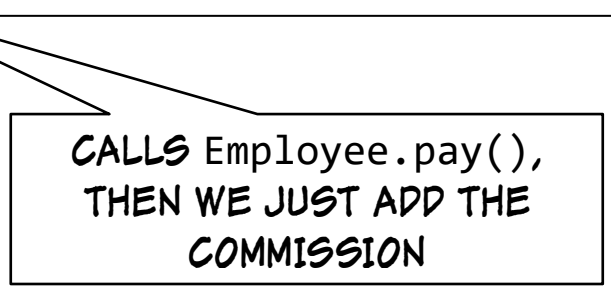
Method Overriding (cont'd)

```
emp = SalesEmployee(123, 'Gavin', 'Lim', 50000)
emp.commission = 10000
print('Pay:', emp.pay())     # 55000 (50000 - 5000 + 10000)
```

Calling an Overridden Method

- You can use `super()` to call a superclass' overridden method.

```
class SalesEmployee(Employee):
    # code snipped

    @property
    def pay(self):
        return super().pay() + self.commission
```



Side Bar: Using the `help()` function

- `help()` returns useful information about a class

```
print(help(SalesEmployee))
```

- Output:

```
class SalesEmployee(Employee)
 |  SalesEmployee(id, first_name, last_name, salary=0)
 |
 |  Method resolution order:
 |      SalesEmployee
 |      Employee
 |      Person
 |      builtins.object
 |
 |
```

SHOWS THE ORDER BY WHICH METHOD CALLS ARE RESOLVED.

OUTPUT CONTINUED ON NEXT SLIDE...

Side Bar: Using the help() Function

```
class SalesEmployee(Employee)
 |  SalesEmployee(id, first_name, last_name, salary=0)
 |
 |  Methods defined here:
 |
 |  __init__(self, id, first_name, last_name, salary=0)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  pay(self)
 |  ----------------------------------------------------------------
 |  Data descriptors inherited from Person:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  full_name
```

Side Bar: Converting Objects to Strings

- By default, printing an object will just display its memory location:

```
emp = SalesEmployee(123, 'Gavin', 'Lim', 50000)
emp.commission = 10000
print(emp)
```

- Output:

```
<__main__.SalesEmployee object at 0x00000139A9F59A48>
```

Side Bar: Converting Objects to Strings (cont'd)

- You can provide a more useful string by overriding `__str__()`

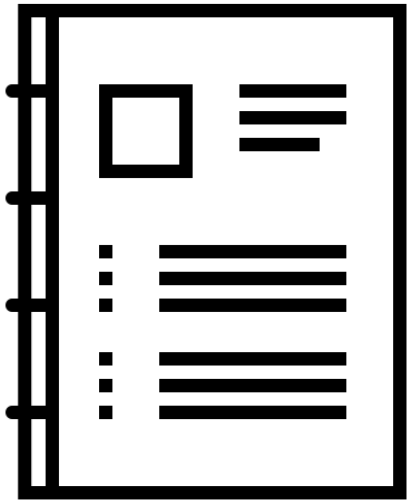
```
class SalesEmployee(Employee):

    # code snipped

    def __str__(self):
        return 'SalesEmployee({}, {}, {})'.format(\
            self.first_name, self.last_name, self.pay)

emp = SalesEmployee(123, 'Gavin', 'Lim', 50000)
emp.commission = 10000
print(emp)
```

- Output:

```
SalesEmployee(Gavin, Lim, 54000.0)
```

Chapter Outline

- Inheritance
- Method Overriding
- **Polymorphism**
- Member Visibility

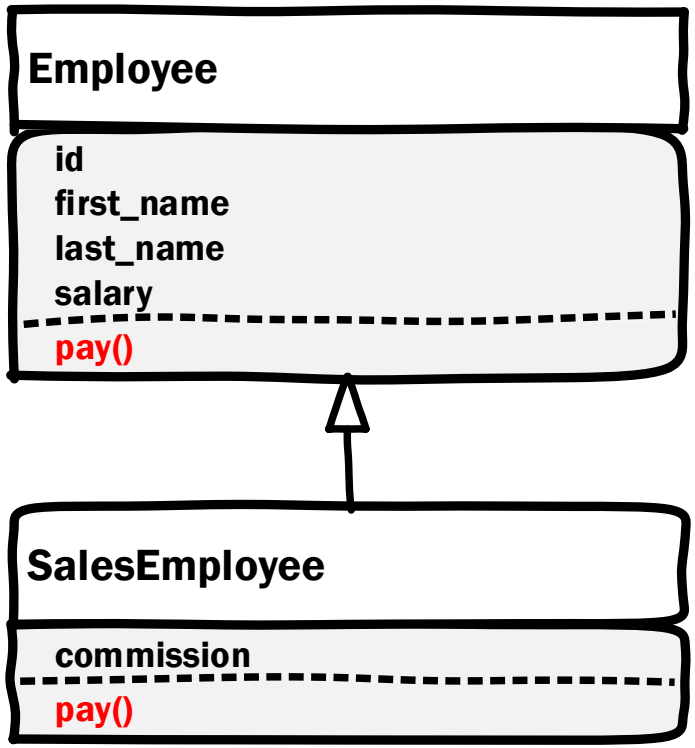


Polymorphism

- “Many forms”
- A feature of OOP that allows an object variable to behave differently depending on the type of object it is holding
- Used with a polymorphic operation
  - A method with many forms
- To illustrate polymorphism, let's take a look at an example

Polymorphism (cont'd)

- The polymorphic operation `pay()` has two forms
  - `Employee.pay()`
    - `salary - tax`
  - `SalesEmployee.pay()`
    - `salary – tax + commission`



Polymorphism (cont'd)

```
emp = Employee(123, 'Homer', 'Simpson', 20000)
print(emp.pay())    # 18000, Employee.pay()

emp = SalesEmployee(456, 'Freddy', 'Flintstone', 50000)
emp.commission = 10000
print(emp.pay())    # 55000, SalesEmployee.pay()
```

- If we assign an `Employee` object to `emp`, the `pay()` method of `Employee` is called
- If we assign a `SalesEmployee` object to `emp`, the `pay()` method of `SalesEmployee` is called

LSP – Liskov Substitution Principle

Code that works with a certain class *T*, will not break when that *T* is replaced with any of its subclasses

If we apply this to our example, it goes like this:

Code that works with the `Employee` class, will not break when `Employee` is replaced with any of its subclasses, including `SalesEmployee`

LSP and Polymorphism

- LSP along with polymorphism allows us to create highly reusable code that will work across a family of classes related by inheritance
- Suppose we have the function below, which given a list of employees, displays the pay of each employee as well as the total payroll amount.

```
def process_payroll(employees):
    total = 0
    for employee in employees:
        print(employee.full_name, employee.pay())
        total += employee.pay()
    print('Payroll Total:', total)
```

LSP and Polymorphism (cont'd)

- Notice that `process_payroll()` works with both `Employee` and `SalesEmployee` objects, as shown below:

```
homer = Employee(123, 'Homer', 'Simpson', 20000)
freddy = SalesEmployee(456, 'Freddy', 'Flintstone', 50000)
freddy.commission = 10000
employees = [homer, freddy]
process_payroll(employees)
```

```
Homer Simpson 18000.0
Freddy Flintstone 55000.0
Payroll Total: 73000.0
```

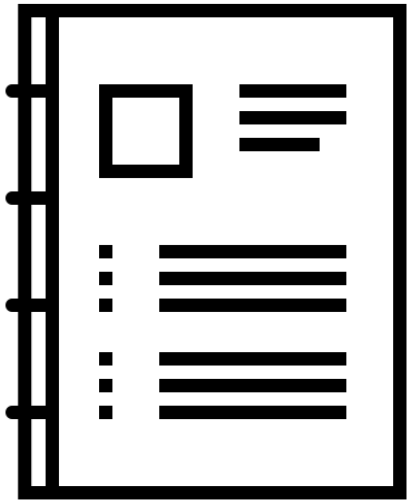
`isinstance()` Function

- **`isinstance()`** has two parameters
  - An object variable
  - Class name
- Returns `True` or `False` based on whether the object is an instance of the named class or any of that class's subclasses

```
homer = Employee(123, 'Homer', 'Simpson', 20000)
freddy = SalesEmployee(456, 'Freddy', 'Flintstone', 50000)

print(isinstance(homer, Employee))         # True
print(isinstance(freddy, Employee))        # True
print(isinstance(freddy, SalesEmployee))   # True
print(isinstance(homer, SalesEmployee))    # False
```

Chapter Outline

- Inheritance
- Method Overriding
- Polymorphism
- **Member Visibility**



Member Visibility

- Object-Oriented languages generally have the following levels of visibility for class members:

Access	class	subclass	everyone
private	✓		
protected	✓	✓	
public	✓	✓	✓

Private Members

- By default, class members have **public** accessibility
  - They can be accessed from any part of the code
- There are certain attributes and methods that should only be used internally within a class, and should not be accessible outside the class.
  - We call these **private** attributes and methods
- In Python, we specify private members by prefixing them with 2 underscores

Private Members (cont'd)

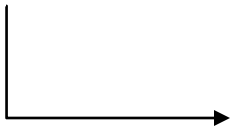
```
class Foo:
    def __init__(self):
        self.__private_attribute = 'private'

    def __private_method(self):
        print('inside private_method...')

    def public_method(self):
        print('inside public_method...')
        print(self.__private_attribute)     # OK
        self.__private_method()             # OK

foo = Foo()
foo.public_method()                     # OK
print(foo.__private_attribute)          # Error
foo.__private_method()                  # Error
```

Protected Members

- In other object-oriented languages such as Java, protected members are only accessible to:
  - the class itself
  - its subclasses
- In Python, there is no way to enforce protected members
- Instead, we prefix members with a single underscore as a hint to other developers that the member is protected and should not be accessed outside the class

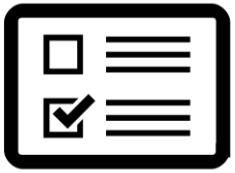
Protected Members (cont'd)

```
class Bar:
    def __init__(self):
        self.__private_attribute = 'private'
        self._protected_attribute = 'protected'

class Moo(Bar):
    def __init__(self):
        super().__init__()
        print(self._protected_attribute)  # protected

moo = Moo()                               # protected
print(moo._protected_attribute)           # protected
```



ALTHOUGH THIS WORKS, YOU SHOULD NOT BE ACCESSING PROTECTED MEMBERS OUTSIDE THE CLASS' HIERARCHY

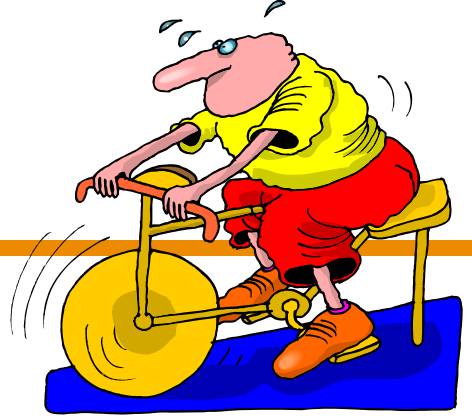
Test Your Knowledge



1. To call the constructor of the super class, you use _____
2. _____ is redefining a method that has already been in a superclass
3. True or false. When printing an object, its attributes are printed by default.
4. The _____ method allows you to return a custom string that represents your object
5. To make an attribute private, you prefix it with ____
6. True or false. Python supports protected members.

Answers

1. `super().__init__()`
2. overriding
3. False. The memory location is printed by default.
4. `__str__()`
5. A double underscore (__)
6. False.



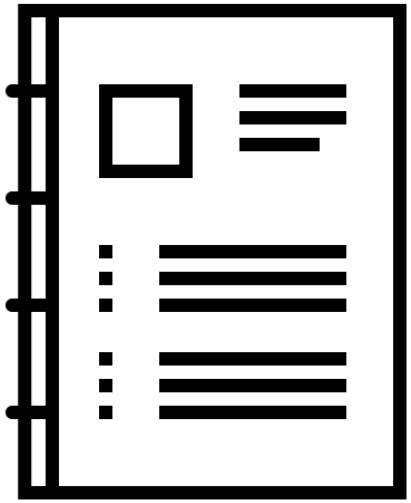
Please turn to Exercise 3 in your Exercise Manual

Chapter 4

Exception Handling

Chapter Outline

- Introducing Exceptions
- Handling Exceptions using `try-except`
- Handling Multiple Exceptions
- Passing the Exception Object
- Using the `else` Block
- Using the `finally` Block
- Raising Exceptions
- Creating Your Own Exception Types



Exceptions

- Events that interrupt the normal processing flow of a program
- Usually some error of some sort
- If unhandled, this causes our program to terminate abnormally

Exception Examples

- Some examples of exceptions:
  - **`IndexError`** which occurs if we try to access a non-existent list element

```
nums = [1, 2, 3]
num = nums[3]  // IndexError: list index out of range
```

  - **`ValueError`**, which occurs when we try to pass an invalid number to `int()`

```
num = int('abc')
// ValueError: invalid literal for int() with base 10: 'abc'
```

Exception Example

```
01 num1 = input('enter 1st number: ’)
02 num1 = int(num1)
03 num2 = input('enter 2nd number: ’)
04 num2 = int(num2)
05 quotient = int(num1) / int(num2)
06 print('{} / {} = {}'.format(num1, num2, quotient))
```

Running this program will produce the following output:

```
enter 1st number: 123
enter 2nd number: abc
Traceback (most recent call last):
  File "exceptions.py", line 4, in <module>
    num2 = int(num2)
ValueError: invalid literal for int() with base 10: 'abc'
```

Exception Example (cont'd)

Running this program will produce the following output:

```
enter 1st number: 50
enter 2nd number: 2
50 / 2 = 25.0
```

However, let's run it again and enter an invalid number:

```
enter 1st number: 10
enter 2nd number: abc
Traceback (most recent call last):
  File "exceptions.py", line 4, in <module>
    num2 = int(num2)
ValueError: invalid literal for int() with base 10: 'abc'
```

Exception Example (cont'd)

Here's another scenario, where we enter zero as the 2nd value:

```
enter 1st number: 10
enter 2nd number: 0
Traceback (most recent call last):
  File "c:/271/exceptions.py", line 5, in <module>
    quotient = int(num1) / int(num2)
ZeroDivisionError: division by zero
```

Exception Example (cont'd)

- If we don't catch the exception, the default exception handler:
  - Prints out error's description
  - The line number where the error occurred
  - Prints the stack trace
    - Hierarchy of function calls that led to the error
  - Causes the program to terminate

```
Traceback (most recent call last):
  File "exceptions.py", line 4, in <module>
    num2 = int(num2)
ValueError: invalid literal for int() with base 10: 'abc'
```

Handling Exceptions

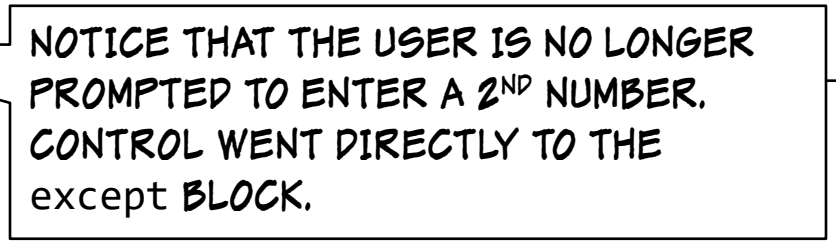
- To handle errors, we use a **`try-except`** block
  - Place the statements that can possibly generate an error inside the `try` block
  - Place the error handling code inside the `except` block

```
try:
    num1 = input('enter 1st number: ')
    num1 = int(num1)
    num2 = input('enter 2nd number: ')
    num2 = int(num2)
    quotient = int(num1) / int(num2)
    print('{} + {} = {}'.format(num1, num2, quotient))
except Exception:
    print("That's not a valid number")
```

Handling Exceptions (cont'd)

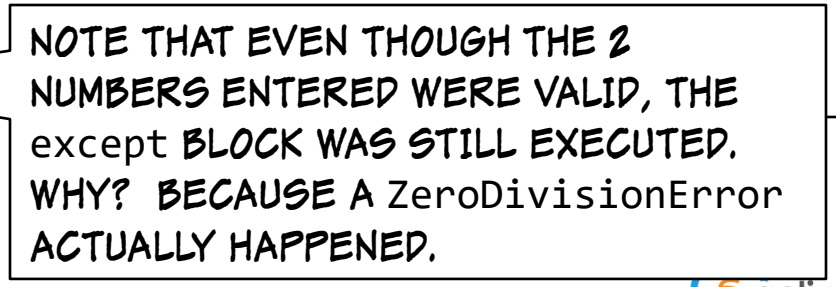
- Let's run the program again, this time with the `try-except` block:

```
enter 1st number: abc
That's not a valid number
```



- Let's run the program again, this time entering zero as the second number:

```
enter 1st number: 10
enter 2nd number: 0
That's not a valid number
```

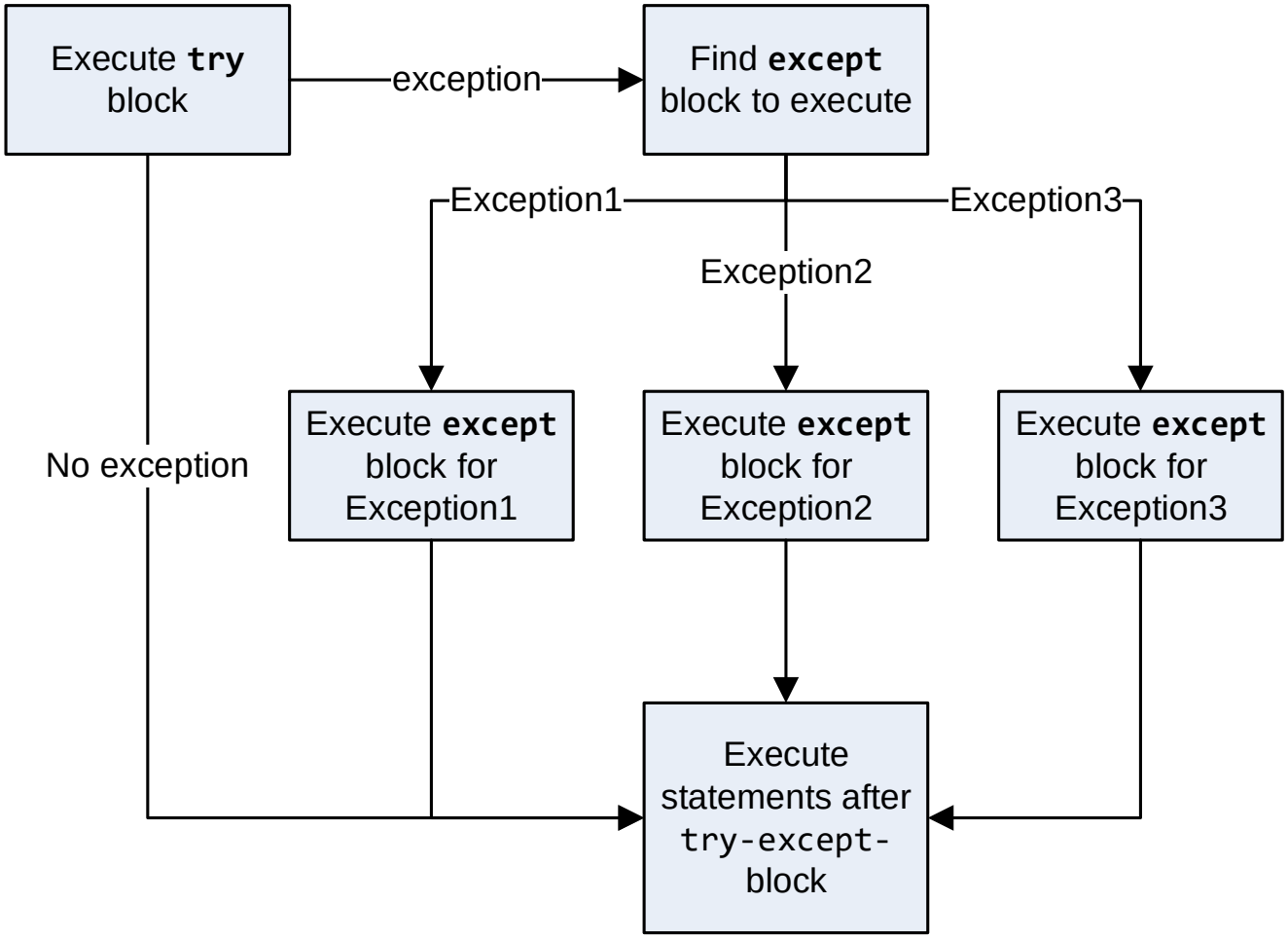


Handling Multiple Exceptions

- For each **try** block, there can be one or more **except** blocks
- Each **except** block contains code to handle a specific error
- When an error happens in the `try` block, the rest of the `try` block is skipped, and control goes to the `except` block willing to handle that error.

```
try:
    # code that can generate errors
except ErrorName1:
    // error handler for ErrorName1
except ErrorName2:
    // error handler for ErrorName2
except ErrorName3:
    // error handler for ErrorName3
```

Handling Multiple Exceptions (cont'd)



Handling Multiple Exceptions (cont'd)

```
try:
    num1 = input('enter 1st number: ')
    num1 = int(num1)
    num2 = input('enter 2nd number: ')
    num2 = int(num2)
    quotient = int(num1) / int(num2)
    print('{} / {} = {}'.format(num1, num2, quotient))
except ValueError:
    print("That's not a valid number")
except ZeroDivisionError:
    print("You can't divide by zero")
```

Handling Multiple Exceptions (cont'd)

- Now, let's try it again:

```
enter 1st number: 10
enter 2nd number: abc  ──────────► ValueError
That's not a valid number
```

```
enter 1st number: 10
enter 2nd number: 0    ──────────► ZeroDivisionError
You can't divide by zero
```

Passing the Exception Object

- If you want to get more information about the error, you can pass it to the except block like this:

```
try:
    num1 = input('enter 1st number: ')
    num1 = int(num1)
    num2 = input('enter 2nd number: ')
    num2 = int(num2)
    quotient = int(num1) / int(num2)
    print('{} / {} = {}'.format(num1, num2, quotient))
except ValueError as e:
    print("That's not a valid number")
    print(e)
except ZeroDivisionError as e:
    print("You can't divide by zero")
    print(e)
```

Passing the Exception Object (cont'd)

- Now, let's try it again:

```
enter 1st number: 10
enter 2nd number: abc
That's not a valid number
invalid literal for int() with base 10: 'abc'
```

```
enter 1st number: 10
enter 2nd number: 0
You can't divide by zero
division by zero
```

Handling Multiple Exceptions (cont'd)

- Python errors are actually classes.

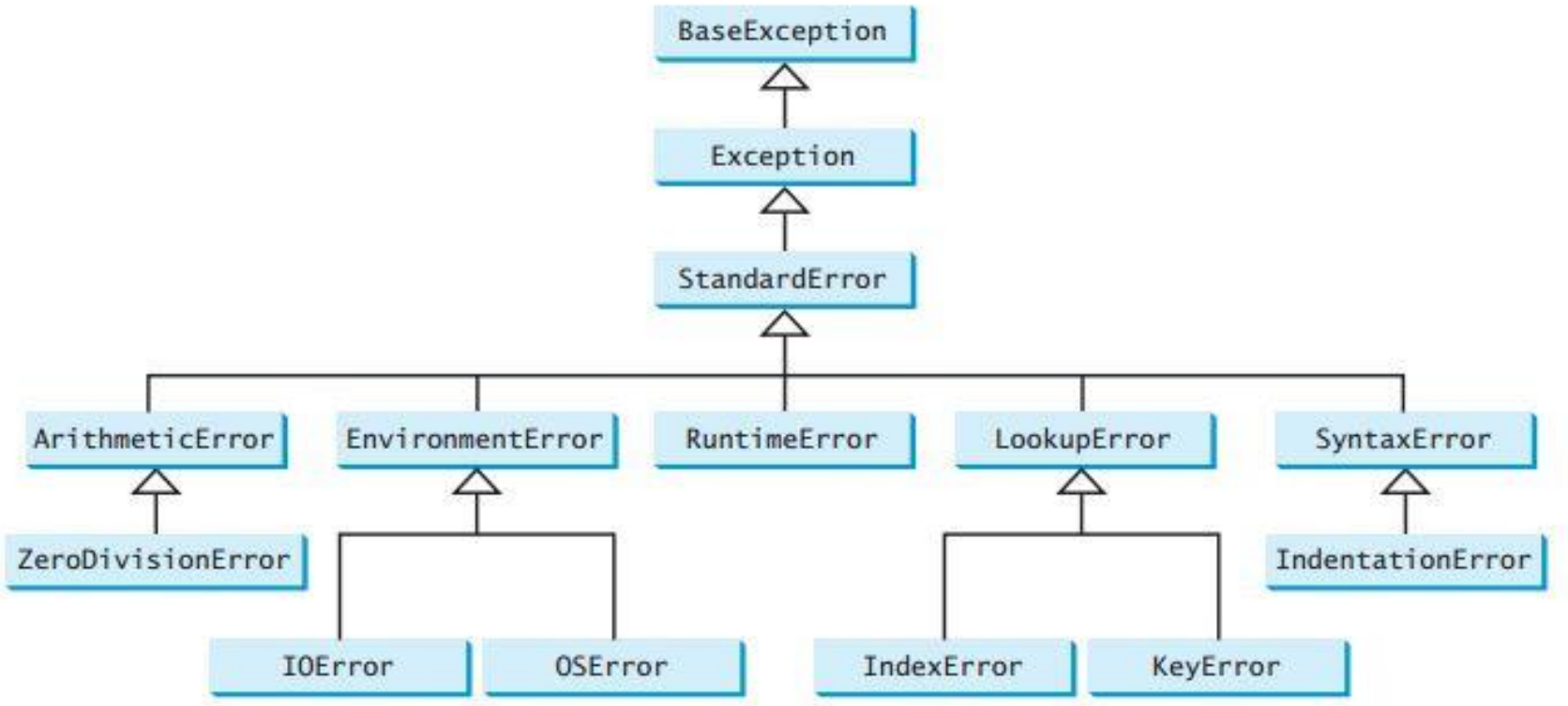
```
print(help(ZeroDivisionError))
```

- Output:

```
Help on class ZeroDivisionError in module builtins:

class ZeroDivisionError(ArithmeticError)
 |  Second argument to a division or modulo operation was zero.
 |
 |  Method resolution order:
 |      ZeroDivisionError
 |      ArithmeticError
 |      Exception
 |      BaseException
 |      object
```

The Exception Class Hierarchy

- As shown in the output, all errors inherit from the `Exception` class:



The Exception Class Hierarchy (cont'd)

- Multiple `except` blocks should be ordered from subclass to superclass.

```
try:
    num1 = input('enter 1st number: ')
    num1 = int(num1)
    num2 = input('enter 2nd number: ')
    num2 = int(num2)
    quotient = int(num1) / int(num2)
    print('{} / {} = {}'.format(num1, num2, quotient))
except ValueError:
    print("That's not a valid number")
except ZeroDivisionError:
    print("You can't divide by zero")
except Exception as e:
    print('An error occurred')
    print(e)
```

Using the `else` Block

- You can add an optional `else` block
  - Will be run only if there are no errors within the `try` block

```
try:
    num1 = input('enter 1st number: ')
    num1 = int(num1)
    num2 = input('enter 2nd number: ')
    num2 = int(num2)
    quotient = int(num1) / int(num2)
    print('{} / {} = {}'.format(num1, num2, quotient))
except ValueError:
    print("That's not a valid number")
except ZeroDivisionError:
    print("You can't divide by zero")
else:
    print('Everything went well.')
```

Using the `else` Block (cont'd)

- Sample output:

```
enter 1st number: 10
enter 2nd number: 2
10 / 2 = 5.0
Everything went well...
```

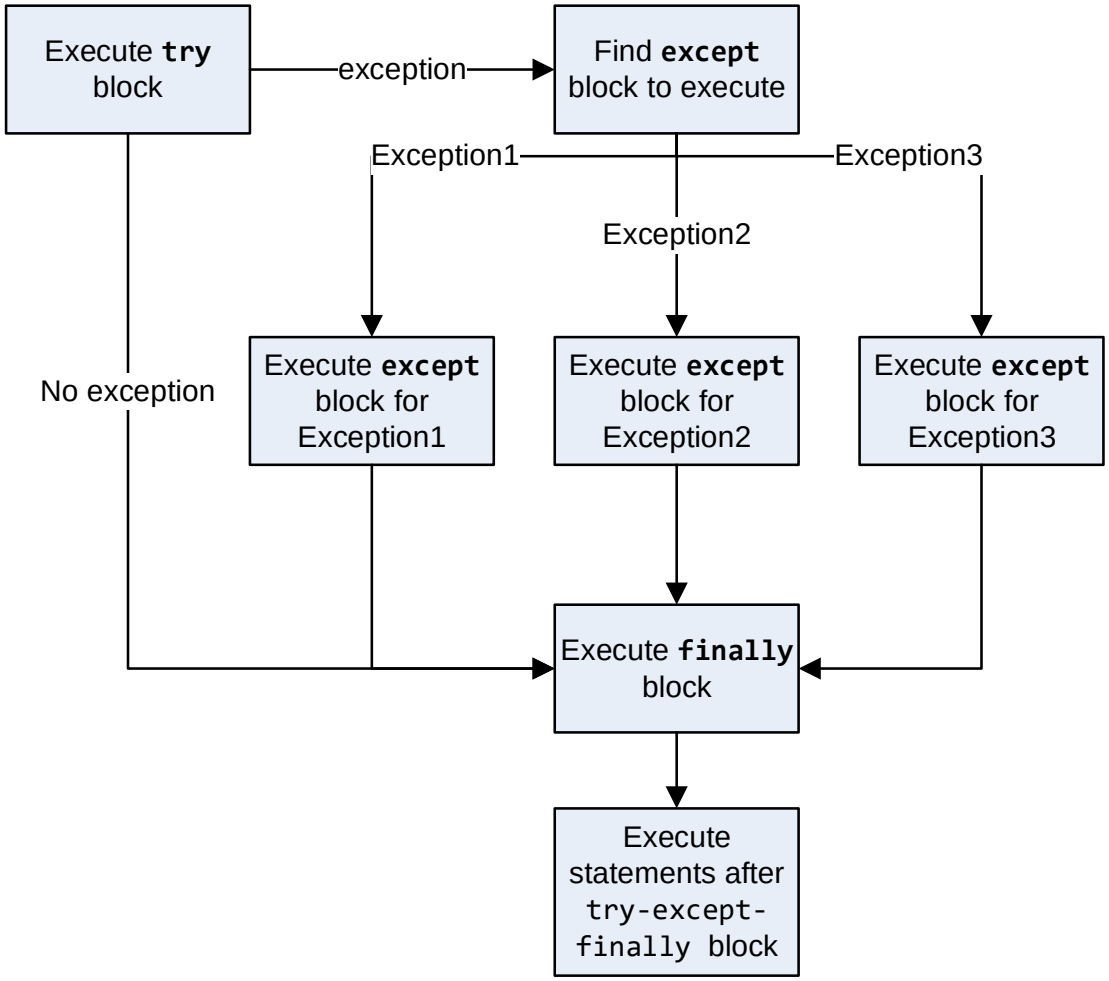
- The `else` block doesn't run if there's an error in the `try` block

```
enter 1st number: 10
enter 2nd number: 0
You can't divide by zero
```

Using the `finally` Block

- The `try-except` block can be followed by a `finally` block
- Executed whether or not an error occurs in the `try` block
- The `finally` block is a key tool for preventing resource leaks
  - When closing a resource like a file, network or database connection, place the code in a `finally` block to ensure that resource is always recovered.
- The `finally` block gets called even if you do a `return`, `continue`, or `break` within the `try` block.

Using the `finally` Block (cont'd)



Using the `finally` Block (cont'd)

```
try:
    num1 = input('enter 1st number: ')
    num1 = int(num1)
    num2 = input('enter 2nd number: ')
    num2 = int(num2)
    quotient = int(num1) / int(num2)
    print('{} / {} = {}'.format(num1, num2, quotient))
except ValueError:
    print("That's not a valid number")
except ZeroDivisionError:
    print("You can't divide by zero")
finally:
    print('Running finally...')
```

The `finally` Block (cont'd)

- Now, let's try it again:

```
enter 1st number: 10
That's not a valid number
Running finally...
```

- The `finally` block gets executed even without any errors

```
enter 1st number: 10
enter 2nd number: 5
10 / 5 = 2.0
Running finally...
```

Raising Exceptions

- It's possible for you to raise your own exceptions using the `raise` keyword

```
raise ExceptionType
```

- `raise` works much like `return`.
  - When run, the rest of the function or method is skipped, and program control returns back to the caller
  - But instead of returning a value to the caller, it raises an exception instead

Raising Exceptions

```
class Person:
    def __init__(self):
        self.name = None
        self.age = 0

    def set_age(self, age):
        if age <= 0:
            raise Exception('age cannot be negative')
        self.age = age

gavin = Person()
gavin.set_age(-1)
```

```
… in set_age
  raise Exception('age cannot be negative')
Exception: age cannot be negative
```

Creating Your Own Exception Types

- It's possible for you to create your own exception types
- Why create user-defined exception types?
- Allows you to give more descriptive names to your exceptions instead of the generic `Exception`:
  - Example: `InvalidUserException`, `ProductNotFoundException`, etc.
- Allows you to have `except` blocks specific to your own exception types:

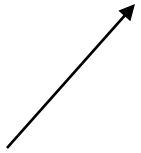
```
try:
    # code in try block
except MyOwnExceptionType:
    print('An error occurred')
```

Creating Your Own Exception Types (cont'd)

- To create your own exception type, create a class that extends the **BaseException**

```
class InvalidAgeException(BaseException):
    pass


class Person:
    def __init__(self):
        self.name = None
        self.age = 0

    def set_age(self, age):
        if age <= 0:
            raise InvalidAgeException('age cannot be negative')
        self.age = age
```

NOW YOU CAN RAISE YOUR OWN EXCEPTION INSTEAD OF GENERIC Exception

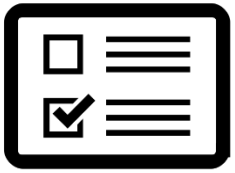


Creating Your Own Exception Types (cont'd)

- You can now have error handling code specific to your own exception type.

```
try:
    gavin = Person()
    gavin.set_age(-1)
except InvalidAgeException as e:
    print(e)
```

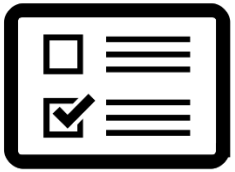
Test Your Knowledge



1. Assuming an `IndexError` occurs on the call to `foo()`, what is the output of the following code?

```
try:
    foo()                       # IndexError occurs here
    print('yahoo')
except Exception:
    print('zillion')
except IndexError:
    print('deno')
finally:
    print('alpha')
print('bazooka')
```

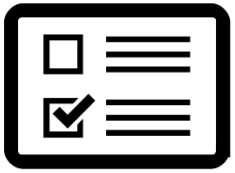
Test Your Knowledge



2. Assuming there are no errors upon execution, what is the output of the following code?

```
try:
    foo()
    print('yahoo')
except Exception:
    print('zillion')
except IndexError:
    print('deno')
else:
    print('omega')
finally:
    print('alpha')
print('bazooka')
```

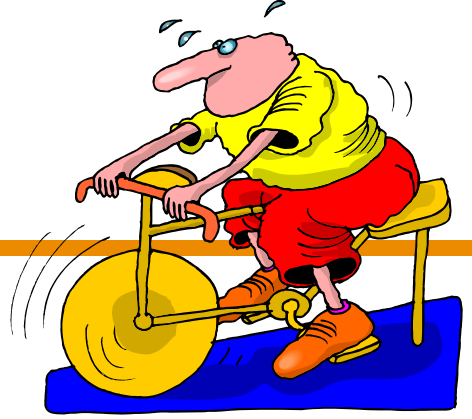
Test Your Knowledge



3. If you want to trigger an exception, you use the _____ keyword.
4. If you want to create your own exception type, you create a class that inherits from _____.

Answers

1. zillion
   alpha
   bazooka
2. yahoo
   omega
   alpha
   bazooka
3. raise
4. BaseException



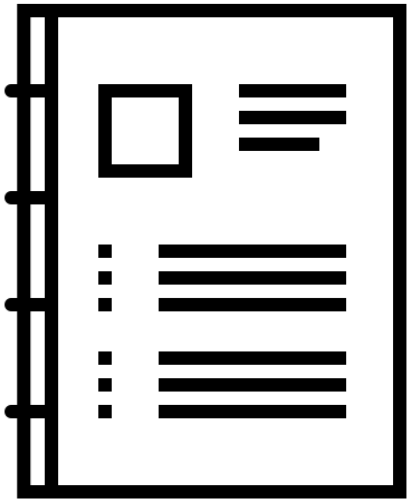
Please turn to Exercise 4 in your Exercise Manual

Chapter 5

Introduction to Django

Chapter Outline

- What is Django?
- Setting Up a Django Application
- Creating a Django Project
- Running Your Django Project
- Creating a View
- Mapping URLs
- Using Templates
- Passing Data to Templates



What is Django?

- Django is one of the most popular web application frameworks in Python
- What is a framework?
  - There are several overhead tasks that we have to do in every web application, which include:
    - Authentication and Authorization
    - Database Access
    - Form Validation
    - Pagination
- A web application framework such as Django helps us with these tasks so we can focus more on our application's business logic.

Setting Up Django

- First, import the Django packages by executing the following command in the command line:

```
pip install django
```

- **What is PIP?**
  - PIP is a package manager that allows you to download 3rd party packages and modules
  - Installed by default in Python 3.4 and above
    - Located in the *Scripts* folder of your Python installation
- To verify that you have correctly installed Django, run the following command to display the version installed:

```
python -m django --version
```

Setting Up Django (cont'd)

- Run `django-admin` to list the commands used to manage a Django application

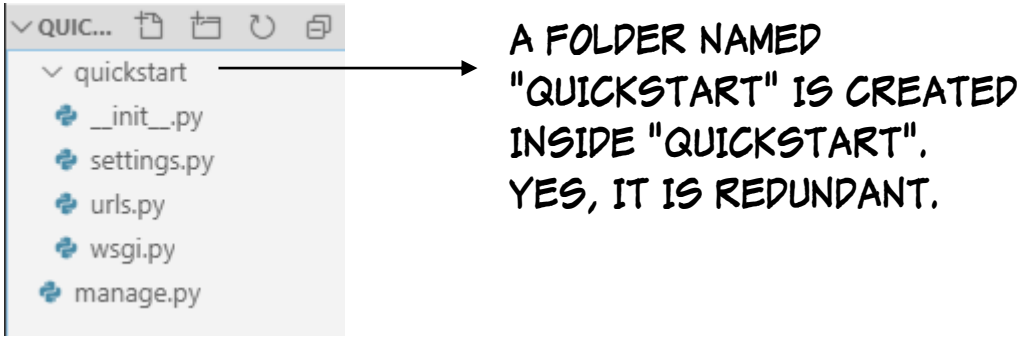
```
C:\271>django-admin

Type 'django-admin help <subcommand>' for help on a specific
subcommand.

Available subcommands:

[django]
    check
    squashmigrations
    …
    startapp
    startproject
    test
    testserver
```

Creating a Django Project

- To create a Django project named "quickstart", run the following command:

```
C:\271>django-admin startproject quickstart
```

- This will create a folder named "quickstart", with the following files and folders within it:

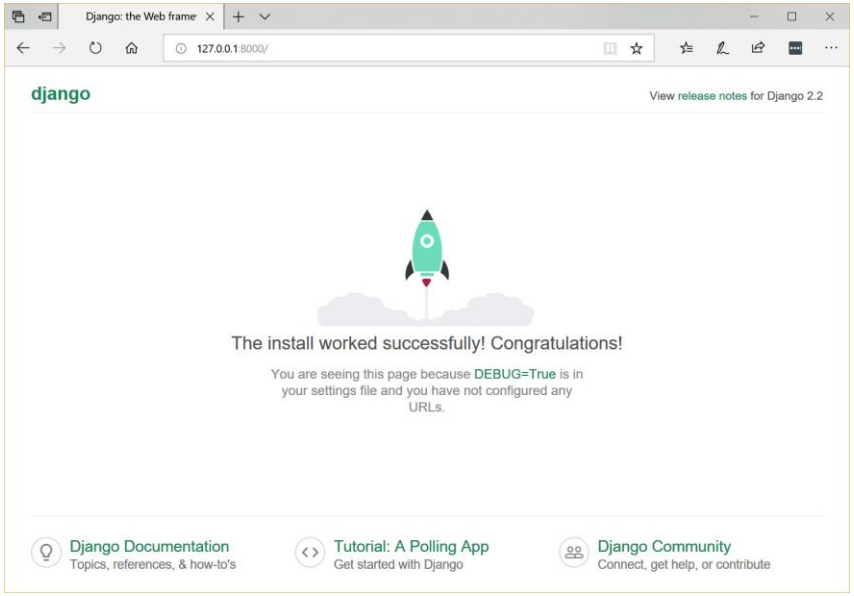


Running Your Django Project

- To run the project, make sure you're inside your projects' directory, and run the command:

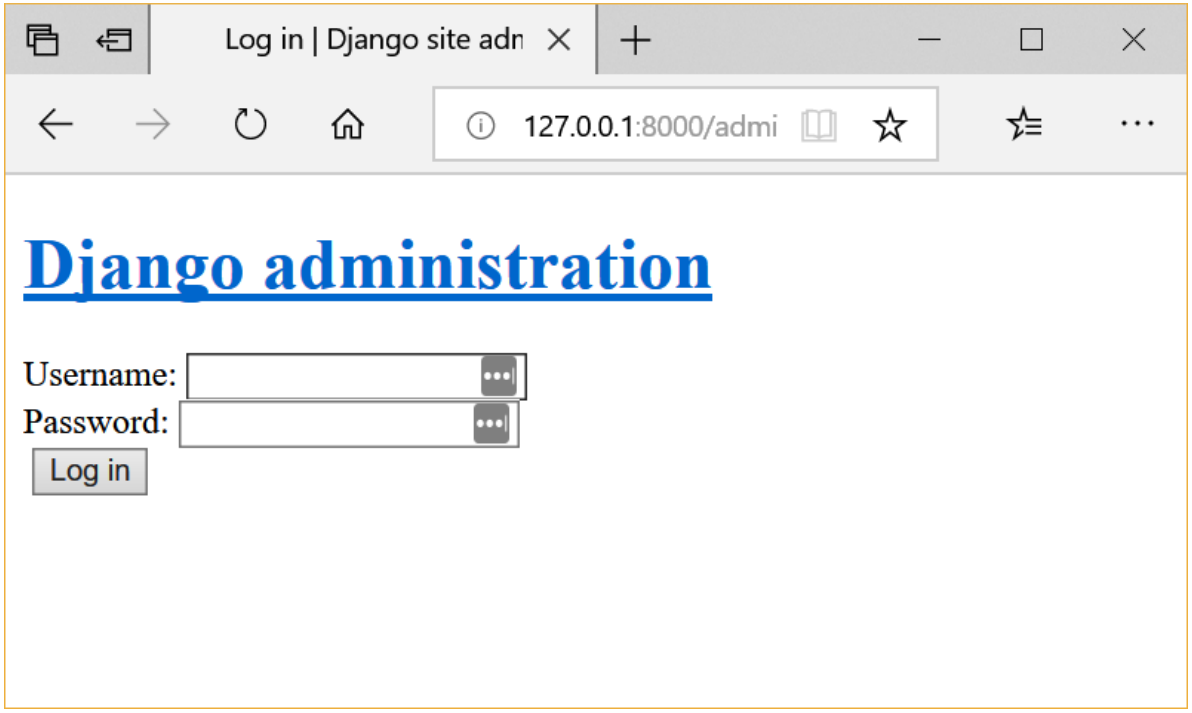
```
C:\271\quickstart>python manage.py runserver
```

- You can then visit http://127.0.0.1:8000 using a browser:



Running Your Django Project (cont'd)

- Every Django app has a default admin page.
  - Visit 127.0.0.1:8000/admin



Apps

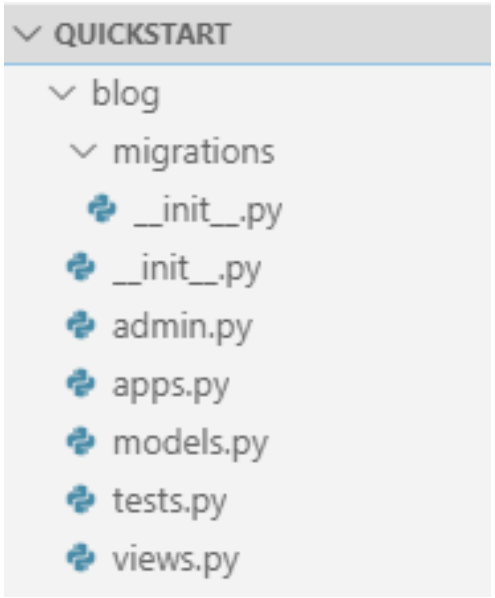
- A Django project can contain multiple **apps**
- Example:
  - Blog
  - Store
  - Orders
- You can use an app in multiple projects

Creating an App within a Project

- To create an app called *blog*, make sure you're in the project's directory and run the command:

```
C:\271\quickstart>python manage.py startapp blog
```

- This creates the *blog* folder within your project's folder, containing the following files:



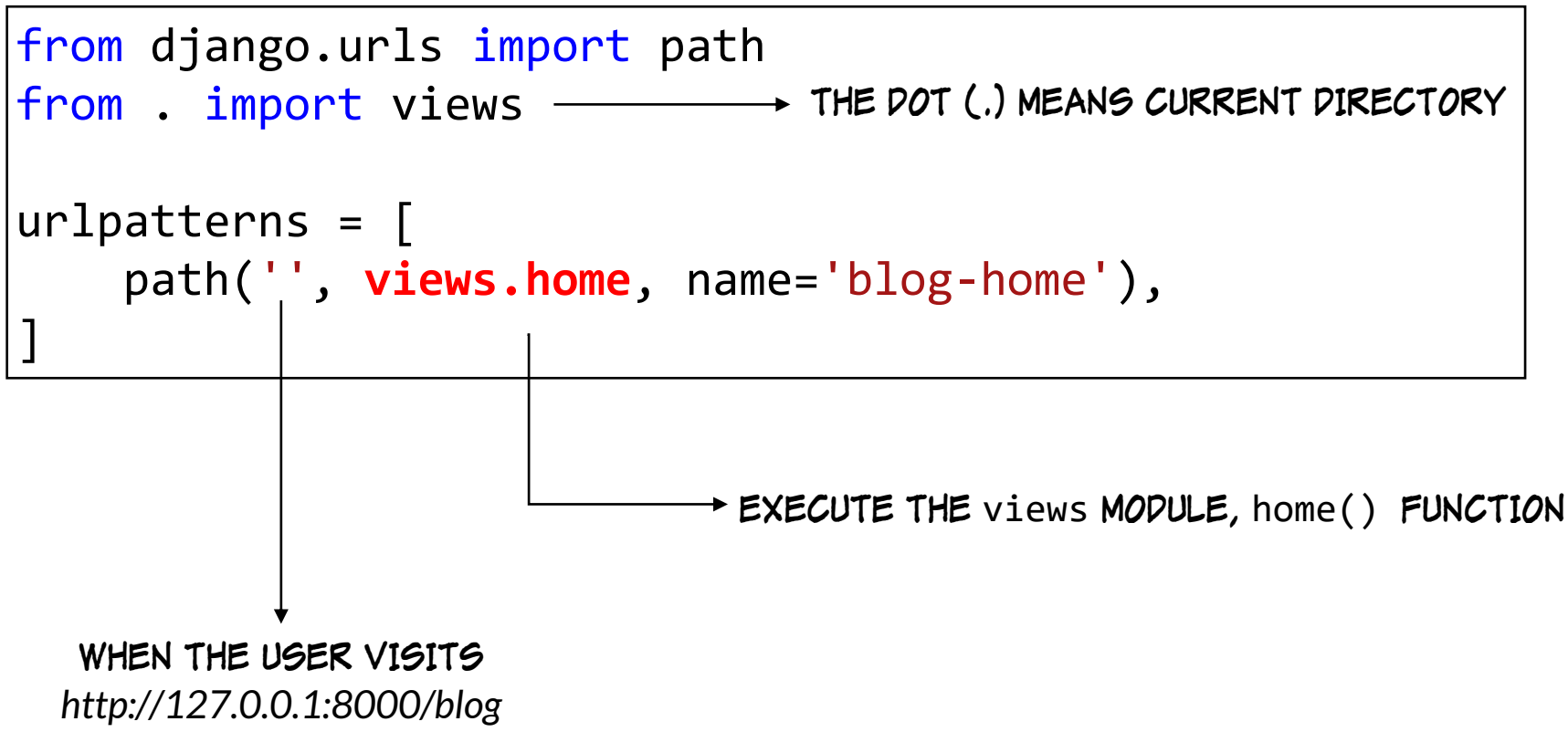
Creating a View

- This is where you define your views
  - We refer to the user interface as a **view**
  - By default, no views are defined
- Let's create a new view by defining the `home()` function in *blog/views.py*:

```
from django.shortcuts import render
from django.http import HttpResponse

def home(request):
    return HttpResponse('<h1>Blog Home</h1>')
```

Mapping URLs to Views

- In *blog/urls.py*, we map the blog app's URLs to your views:



Mapping URLs to Apps

- In *quickstart/urls.py*, we map URLs to our project's apps:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('blog/', include('blog.urls')),
]
```

ADD THIS FUNCTION

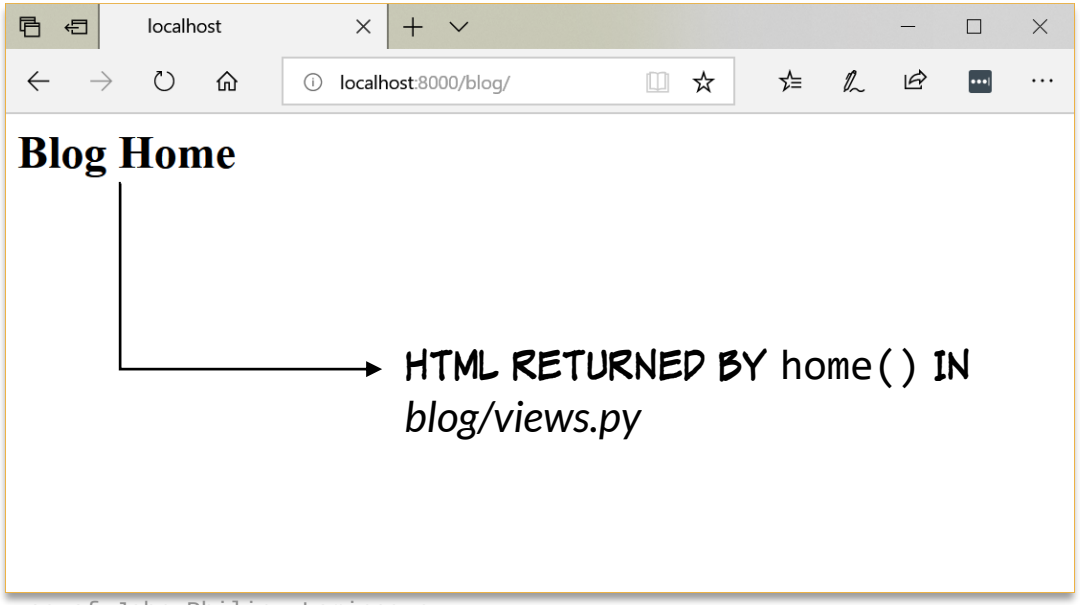
WHEN THE USER VISITS *127.0.0.1/blog*,
USE *blog/urls.py* TO MAP TO A VIEW

Testing It Out

- Run the project again by running:

```
C:\271\quickstart>python manage.py runserver
```

- Visit *localhost:8000/blog*.
  - "localhost" is the same as 127.0.0.1



What Happened?

- When you visited *http://localhost:8000/blog*, the following sequence happened:
  1. *quickstart/urls.py* mapped "/blog" to *blog/urls.py*

```
path('blog/', include('blog.urls'))
```

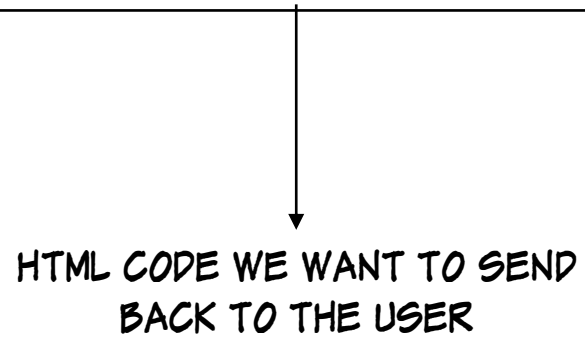
  2. *blog/urls.py* mapped the remainder of the URL after /blog/, in this case ('') to *views.py* home( ) function.

```
urlpatterns = [
    path('', views.home, name='blog-home'),
]
```

What Happened? (cont'd)

3. In *blog/views.py*, `home()` is executed, which displays "Blog Home"

```
def home(request):
    return HttpResponse('<h1>Blog Home</h1>')
```



Setting the Default App

- Let's modify *quickstart/urls.py* to make the blog app our default app

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('blog.urls')),
]
```

CHANGED FROM 'blog/' TO ''

NOW YOU CAN ACCESS THE BLOG APP BY SIMPLY VISITING localhost:8000

Using Templates

1. Inside the *blog* directory, create a *templates* directory
2. Inside the *templates* directory, create a *blog* directory
   - By this time, you should have:
     - *C:\271\quickstart\blog\templates\blog*
3. Create *quickstart/blog/templates/blog/**home.html***

```
<!DOCTYPE html>
    <head>
        <title></title>
    </head>
    <body>
        <h1>Blog home template!</h1>
    </body>
</html>
```

Using Templates (cont'd)

4. Add the following line in *quickstart/settings.py*

```
INSTALLED_APPS = [
    'blog.apps.BlogConfig',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

DEFINED IN *blogs/appy.py*

Using Templates (cont'd)

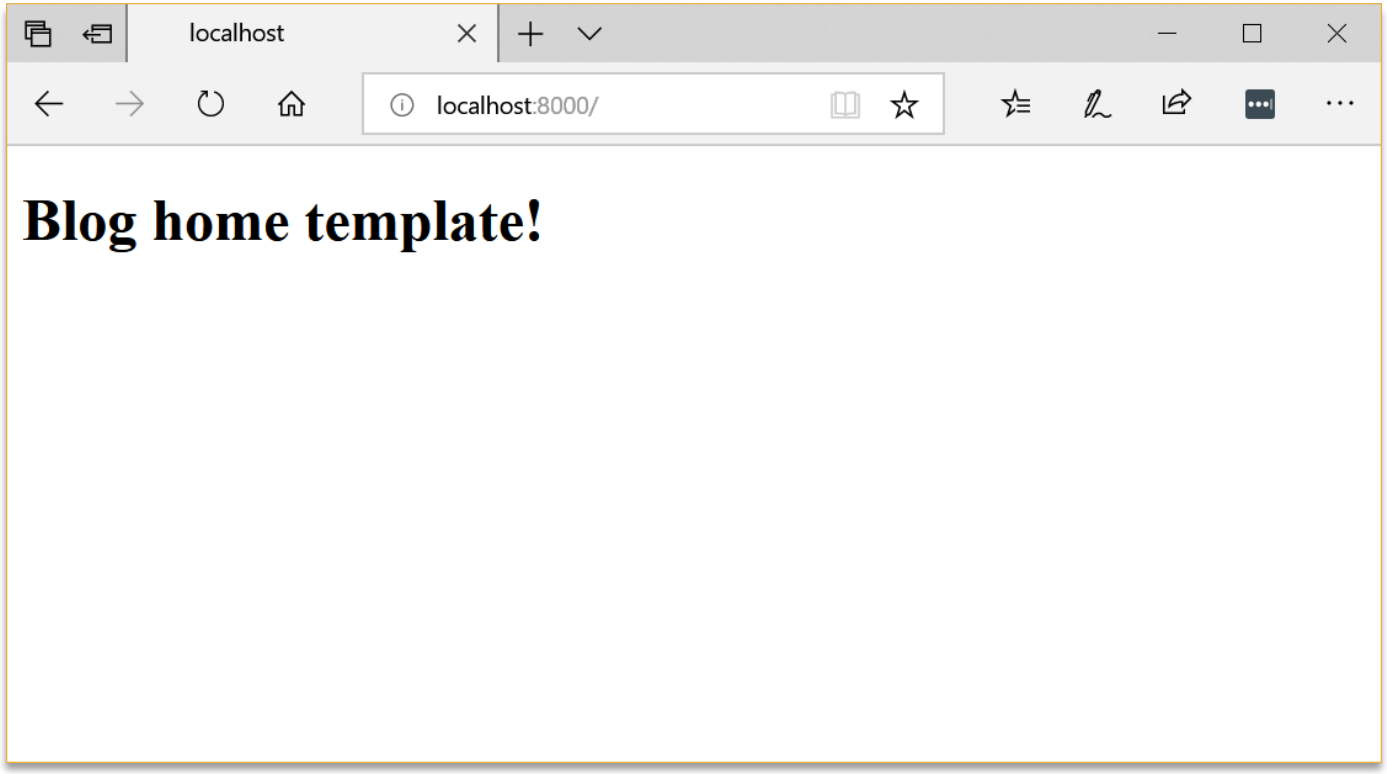
5. In *blog/view.py*, instead of hard-coding HTML inside the `home()` function, use our template instead.

```
from django.shortcuts import render
from django.http import HttpResponse

def home(request):
    return render(request, 'blog/home.html')
```

REFERS TO
templates/blog/home.html

Using Templates (cont'd)

6. Test it out by visiting *http://localhost:8000*



Passing Data to Templates

- The previous example just displayed fixed content
- Applications however, usually generate pages that are generated in real time, with data coming from a data source such as a database
- Let's take a look at how to pass data to our template, and have that data displayed by our template

Passing Data to Templates (cont'd)

- In *blog/views.py*, let's create a list of data called `data`.

```
from django.shortcuts import render

def home(request):
    data = {
        'title': 'ActiveLearning Launches Python Training',
        'content': 'Very comprehensive course on Python and OOP'
    }
    return render(request, 'blog/home.html', data)
```


ADD A 3RD ARGUMENT, WHICH IS A DICTIONARY CONTAINING THE DATA THAT YOU WANT TO PASS TO THE TEMPLATE

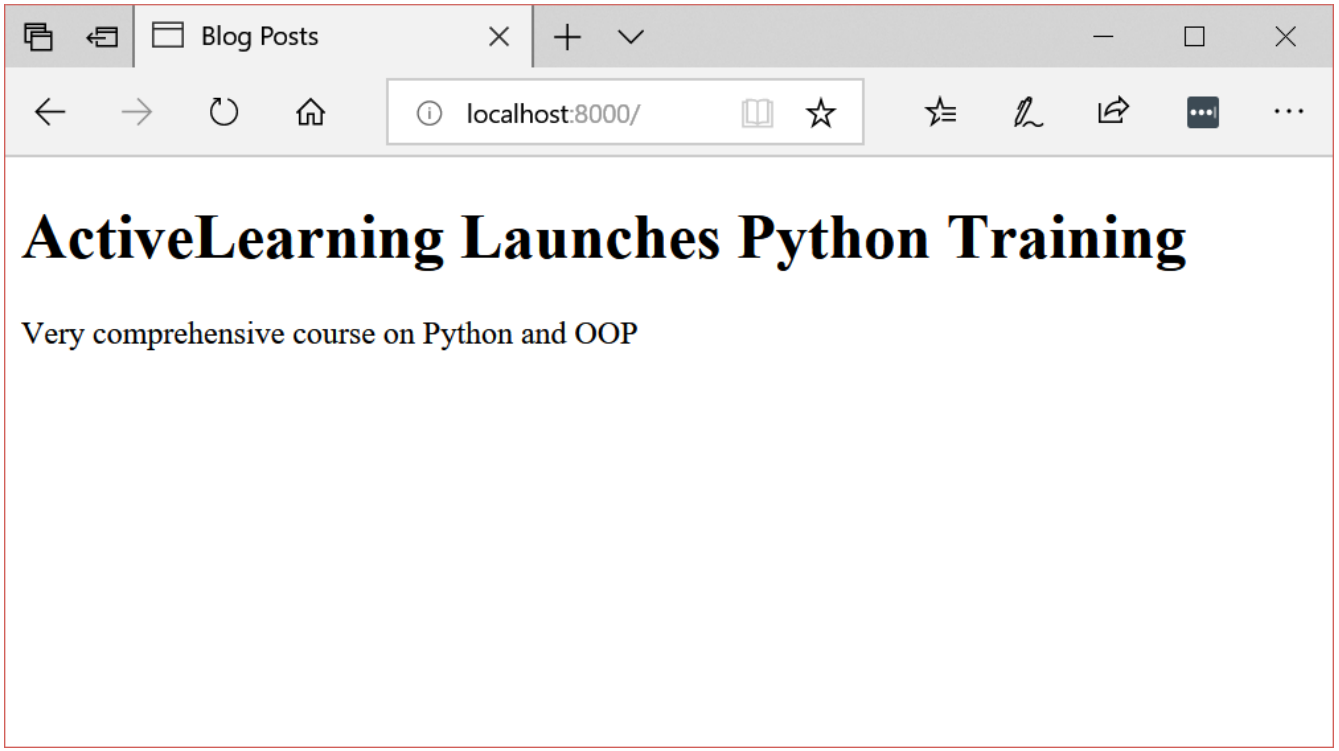
Passing Data to Templates (cont'd)

blog/templates/blog/home.html

```
<!DOCTYPE html>

<head>
    <title>Blog Posts</title>
</head>

<body>
    <h1>{{title}}</h1>
    <p>
        {{content}}
    </p>
</body>

</html>
```

Passing Data to Templates (cont'd)

- Test it out by visiting *http://localhost:8000*



Passing Data to Templates (cont'd)

- In *blog/views.py*, let's pass a list of blog posts this time.

```
from django.shortcuts import render

blog_posts = [
    {
        'author': 'Gavin',
        'title': 'Blog Post 1',
        'content': 'First post content',
        'date_posted': 'Sep 1, 2020'
    },
    {
        'author': 'Julie',
        'title': 'Blog Post 2',
        'content': '2nd post content',
        'date_posted': 'Sep 12, 2020'
    }
]
```

Passing Data to Templates (cont'd)

- In home( ), we pass data to the template.

```
def home(request):
    data = {
        'posts': blog_posts
    }
    return render(request, 'blog/home.html', data)
```

ADD A 3RD ARGUMENT, WHICH IS A DICTIONARY CONTAINING THE DATA THAT YOU WANT TO PASS TO THE TEMPLATE

Passing Data to Templates (cont'd)

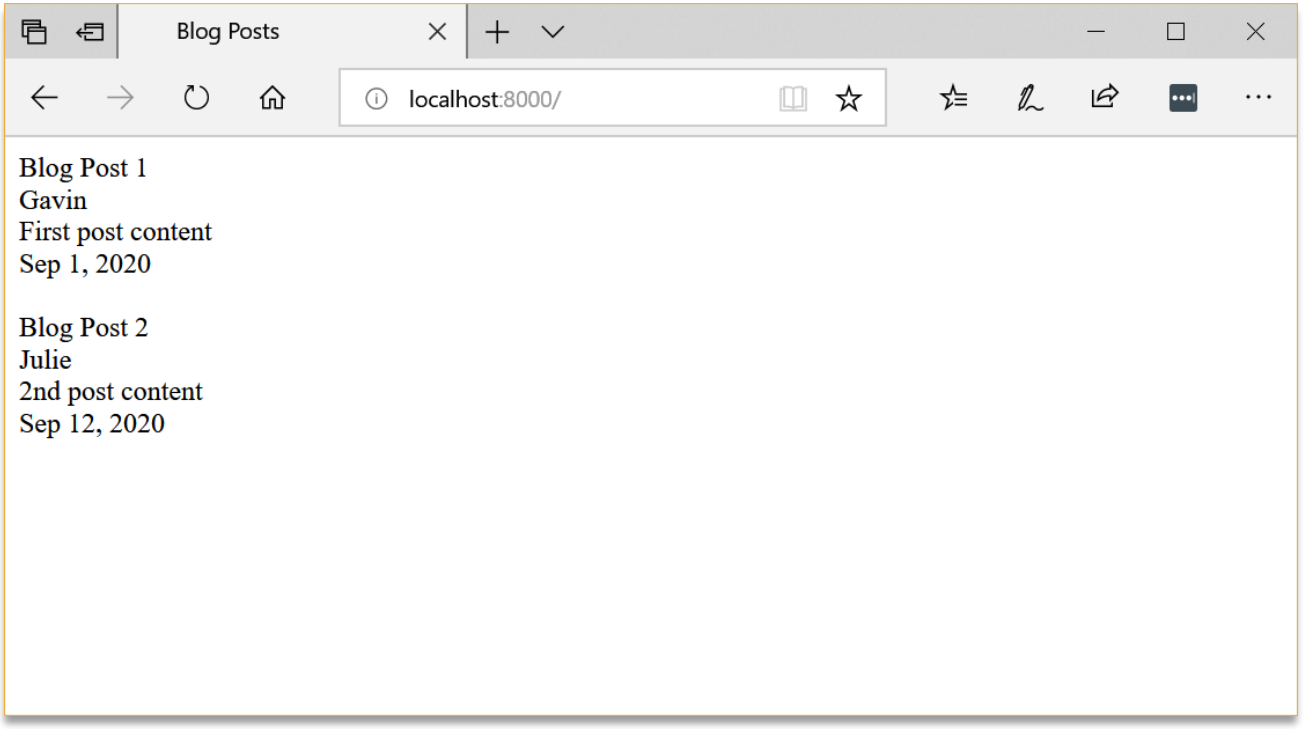
blog/templates/blog/home.html

```
<!DOCTYPE html>
<head>
    <title>Blog Posts</title>
</head>

<body>
    {% for post in posts %}
        {{post.title}}<br>
        {{post.author}}<br>
        {{post.content}}<br>
        {{post.date_posted}}<br>
        <br>
    {% endfor %}
</body>

</html>
```

THIS IS THE ITEM WITH THE KEY 'posts' IN THE DICTIONARY PASSED BY home()

Passing Data to Templates (cont'd)

- Test it out by visiting *http://localhost:8000*



Test Your Knowledge

Before you go...

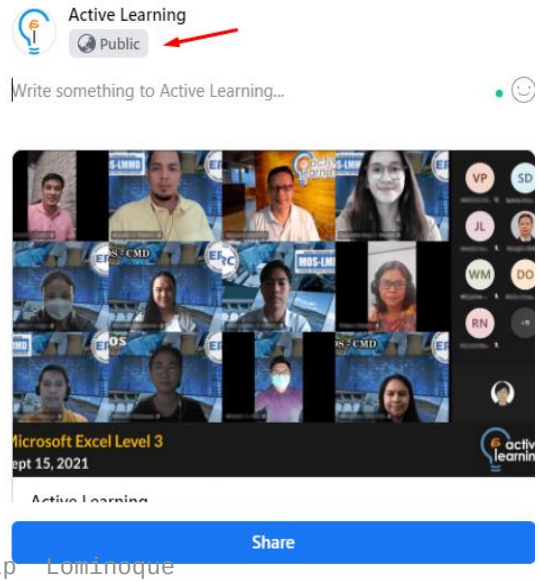
Upgrade yourself further…

Visit *www.activelearning.ph/courses*

Win an ActiveLearning Shirt!

Visit *https://www.facebook.com/active.learning.phil/*

- Like the Page!
- Like the Class Photo
- Share the Class Photo
  - **Note:** Make sure you share to **Public**



Course Evaluation

Your Feedback Matters to Us

Visit *www.tinyurl.com/aleval*

*Note: Certificates will only be released upon completion of the evaluation form.

active learning

And Lastly…

Your firmware has just been upgraded!