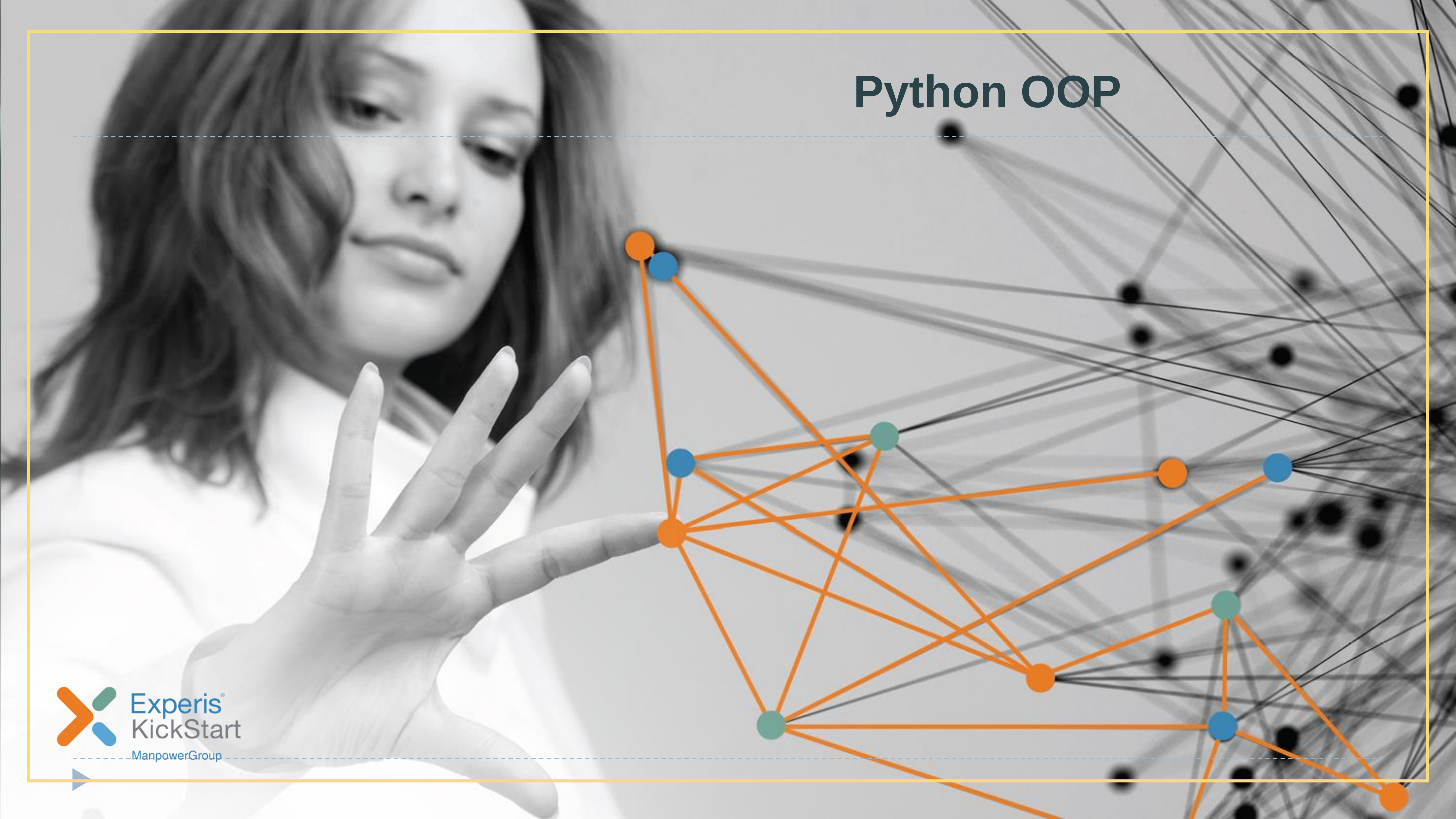
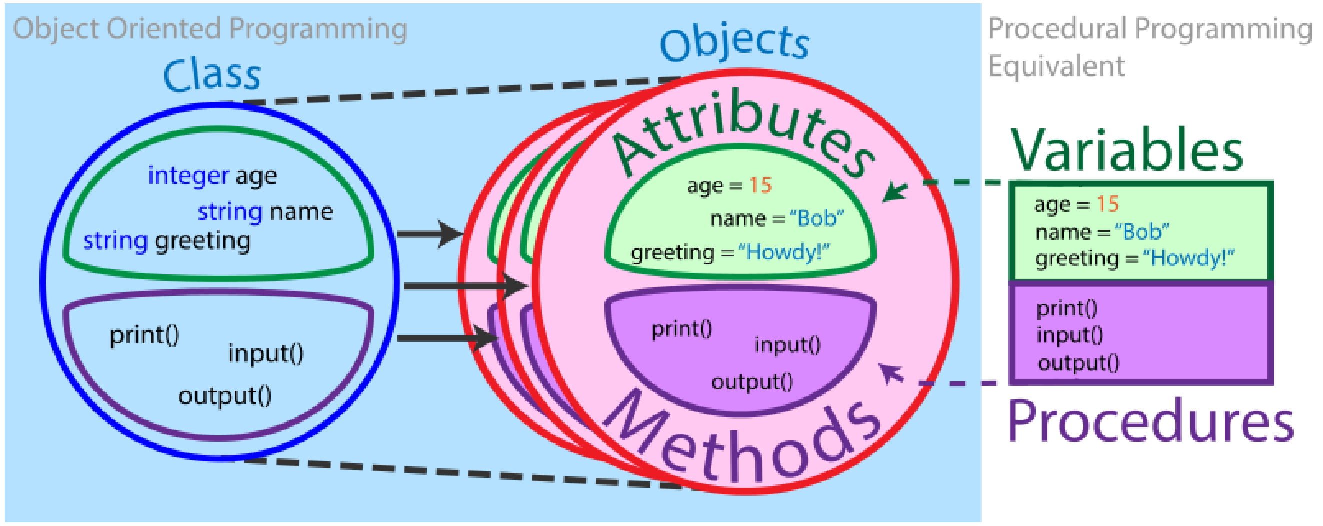
Python OOP



Experis® KickStart
ManpowerGroup

Object-Oriented Programming (OOP)

- OOP is a programming paradigm based on the concept of "objects"
- An object represents an entity in the program, which holds data about itself (**attributes**), and defines functions (**methods**) for manipulating the data
- An object is created (*instantiated*) from a "blueprint" called a **class**, which dictates its behavior by defining its attributes and methods



Objects in Python

- In fact, as we have already pointed out, everything in Python is an object
- So, for example, a Python string is an instance of the `str` class
- A `str` object possesses its own data (the sequence of characters making up the string) and provides (“*exposes*”) a number of methods for manipulating that data
- For example, the capitalize() method returns a new string object created from the original string by capitalizing its first letter:

```
str = "hello world"
str.capitalize()
```

```
'Hello world'
```

- Even indexing a string is really to call the method `__getitem__`:

```
str.__getitem__(6)
```

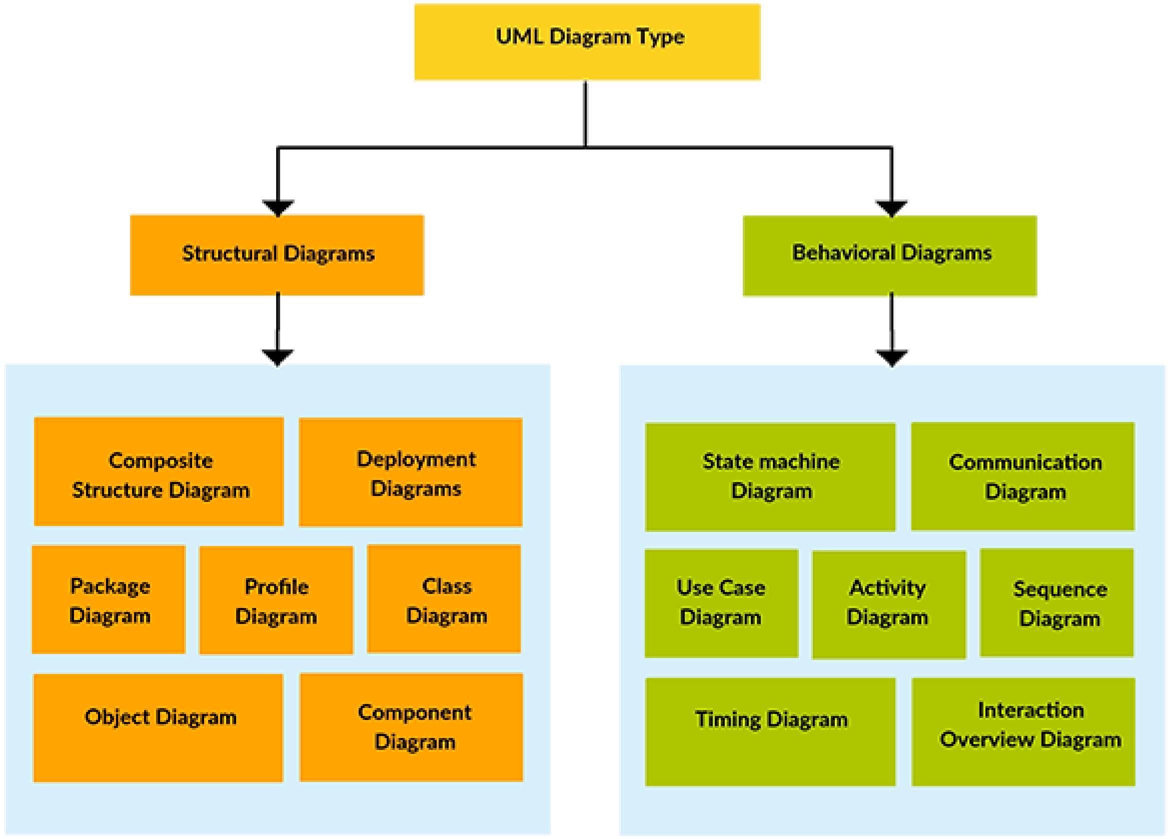
```
'w'
```

Object-Oriented Design

- **Object-oriented design** is the process of planning a system of interacting objects for the purpose of solving a software problem
- Using the OOP approach you can break a problem down into units of data and operations that it is appropriate to carry out on that data
- For example, a retail bank deals with people who have bank accounts
- A natural object-oriented approach to managing a bank would be to define:
  - `BankAccount` class, with attributes such as an account number, balance and owner, and methods for allowing (or forbidding) transactions depending on its balance
  - `Customer` class with attributes such as a name, address, and date of birth, and methods for calculating the customer's age from their date of birth for example

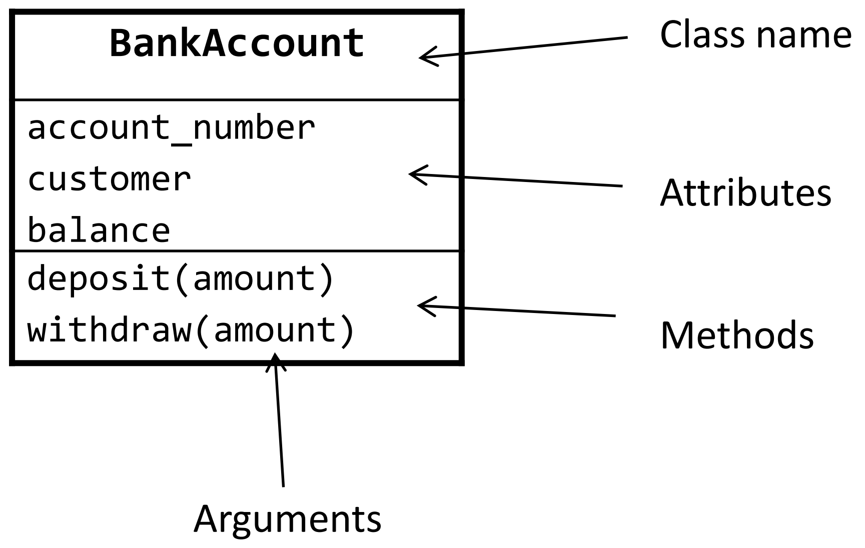
UML

- **UML (Unified Modeling Language)** is a general-purpose, modeling language that provides a standard way to visualize the design of a system



UML Class Diagram

- A **class diagram** describes the structure of a system by showing the system's classes, their attributes, methods, and the relationships among objects



Object-Oriented Concepts

- **Information hiding/Encapsulation**: The ability to protect some components of the object from external entities
- **Inheritance**: The ability for a class to extend or override functionality of another class
- **Protocol/Interface**: A common means for unrelated objects to communicate with each other. These are definitions of methods and values which the objects agree upon in order to co-operate, part of an API.
- **Polymorphism**: The ability to replace an object with its subobjects
- **Dependency injection**: If an object depends upon having an instance of some other object, then the needed object is "injected" into the dependent object

Defining a Class

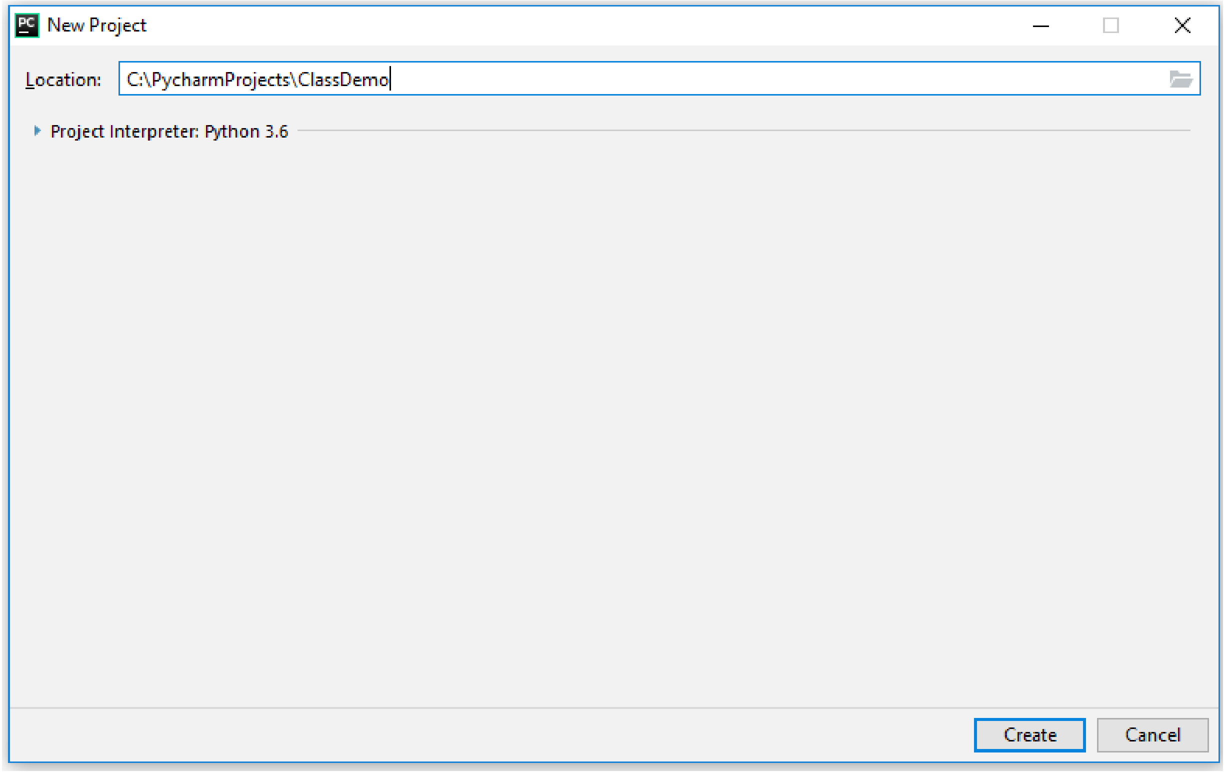
- A class is defined using the **class** keyword and indenting the body of statements (attributes and methods) in a block following this declaration:

```
class ClassName:
    <attribute_1 definition>
    ...
    <attribute_N definition>
    <method_1 definition>
    ...
    <method_N definition>
```

- It is conventional to write classes names in *CamelCase*
- It is a good idea to follow the **class** statement with a docstring describing what the class does
- Class methods are defined using the familiar def keyword, but the first argument to each method should be a variable named self, which refers to the object that invoked the method

Class Example

- Create a new project in PyCharm named ClassDemo



Location: C:\PycharmProjects\ClassDemo

Project Interpreter: Python 3.6

- Add a file named bank_account.py to the project

Class Example

- Define the following BankAccount class:

```python
class BankAccount:
    """A class representing a bank account"""
    def __init__(self, customer, account_number, balance=0):
        self.customer = customer
        self.account_number = account_number
        self.balance = balance

    def deposit(self, amount):
        """Deposit amount into the bank account"""
        if amount > 0:
            self.balance += amount
        else:
            print("Invalid deposit amount:", amount)

    def withdraw(self, amount):
        """Withdraw amount from the bank account, ensuring there are 
sufficient funds """
        if amount > 0:
            if amount > self.balance:
                print("Insufficient funds")
            else:
                self.balance -= amount
        else:
            print("Invalid withdrawal amount:", amount)
```

- __init__() is special method called **initializer** or **constructor**
- It is invoked every time after a new object is created in memory
- Its purpose is to create the object's attributes with some initial values

Inside the class we use **self** to access the object's attributes and methods

Importing the Class

- To use the BankAccount class in a Notebook, copy the script bank_account.py to C:\Notebooks
- Create a new Notebook named TestBankAccount
- Import the BankAccount class from bank_account.py by writing:

```
from bank_account import BankAccount
```

- This notebook can now create BankAccount objects and manipulate them by calling the methods described earlier

Creating Objects

- An *instance* of a class is created with the syntax:

```
object = ClassName(args)
```

  - If the class has an __init__ method, the arguments must match the parameters of the __init__ method (without self)
- In our example, an account is opened by creating a BankAccount object:

```
my_account = BankAccount("Joe Smith", 34572881)
```

- This statement does the following:
  - Creates an object of the BankAccount class
  - Invokes the __init__ method, passing the newly created BankAccount object to **self**, and the values "Joe Smith" and 3452881 to the customer and account_id parameters of __init__
  - Adds the attributes customer, account_number, and balance to the newly created object (balance gets the default value 0)
  - Assigns the reference of the BankAccount object to the variable my_account

Attributes and Methods

- Both attributes and methods of the object are accessed using the dot notation:

```
object.attribute                # access an attribute
object.method(arguments)        # call a method
```

- For example:

```
my_account.account_number  # access an attribute of my_account
```

```
34572881
```

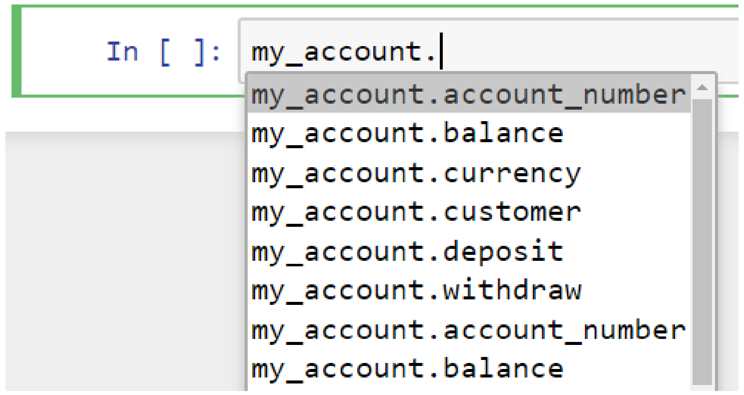
```
my_account.deposit(250)    # call a method of my_account
```

```
my_account.balance
```

```
250
```

Attributes and Methods

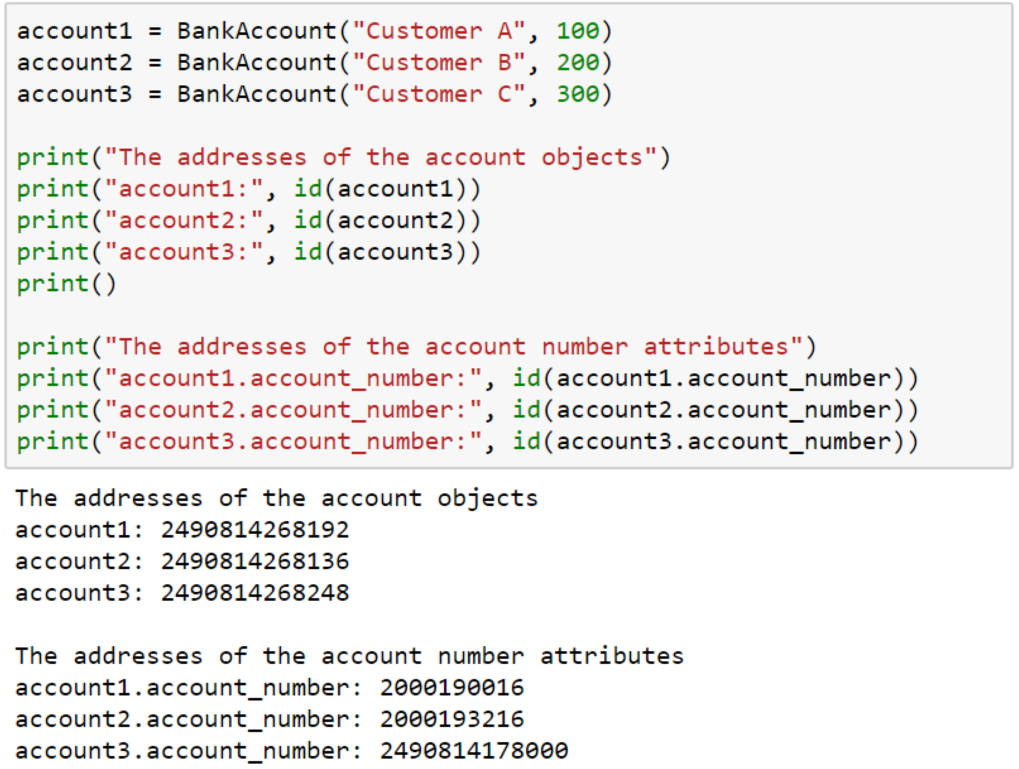
- In Jupyter Notebook, after typing the object name and dot (.), you can press Tab to examine all the object's attributes and methods:



```
In [ ]: my_account.
        my_account.account_number
        my_account.balance
        my_account.currency
        my_account.customer
        my_account.deposit
        my_account.withdraw
        my_account.account_number
        my_account.balance
```

Attributes and Methods

- We can create as many objects as we want, each object with its own set of attributes
- Changing the attributes for one object will not affect the attributes of other objects



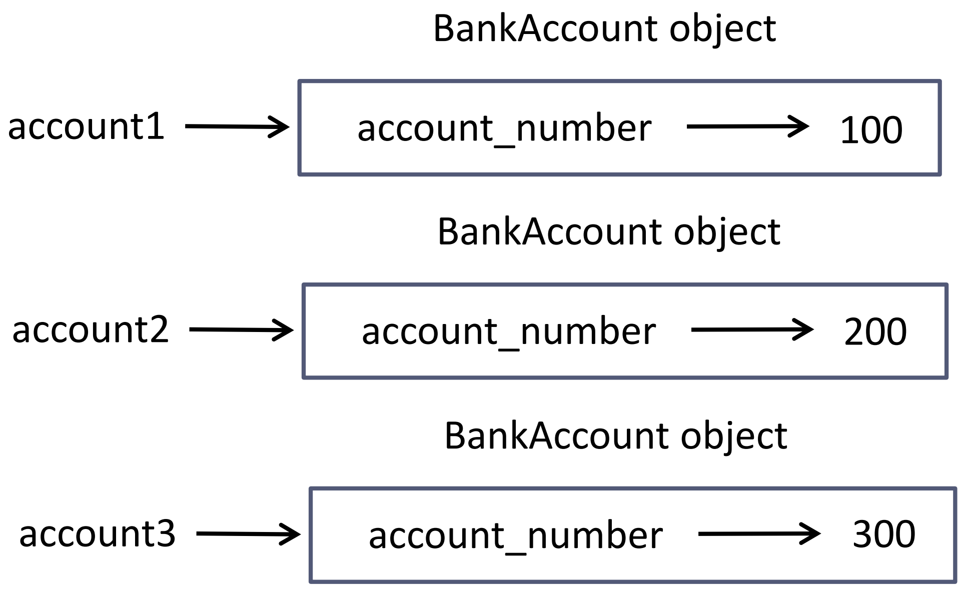
```python
account1 = BankAccount("Customer A", 100)
account2 = BankAccount("Customer B", 200)
account3 = BankAccount("Customer C", 300)

print("The addresses of the account objects")
print("account1:", id(account1))
print("account2:", id(account2))
print("account3:", id(account3))
print()

print("The addresses of the account number attributes")
print("account1.account_number:", id(account1.account_number))
print("account2.account_number:", id(account2.account_number))
print("account3.account_number:", id(account3.account_number))
```

```
The addresses of the account objects
account1: 2490814268192
account2: 2490814268136
account3: 2490814268248

The addresses of the account number attributes
account1.account_number: 2000190016
account2.account_number: 2000193216
account3.account_number: 2490814178000
```



$$\begin{tikzcd}
\text{account1} \arrow[r] & \boxed{\text{account\_number} \longrightarrow 100} \\
\text{account2} \arrow[r] & \boxed{\text{account\_number} \longrightarrow 200} \\
\text{account3} \arrow[r] & \boxed{\text{account\_number} \longrightarrow 300}
\end{tikzcd}$$

Passing Objects as Arguments to Functions

- Just like built-in objects, we can pass objects of user-defined classes to functions
- The following shows how to pass BankAccount objects to a function named print_account_info()

```python
def print_account_info(account):
    print("###########################")
    print("Customer:", account.customer)
    print("Account number:", account.account_number)
    print("Balance:", account.balance)
    print("###########################")
```

```python
print_account_info(my_account)
```

```
###########################
Customer: Joe Smith
Account number: 34572881
Balance: 250
###########################
```

Exercise

- Champion Motors Ltd. both sells and rents vehicles
- It rents two types of vehicles – Car and Bus
- Every vehicle has a license number, a make, a model and price
- Write the class Vehicle that represents a motor vehicle, and includes:
  - A constructor that gets (as parameters) a vehicle's number, a make, a model and price, and initializes the attributes of the vehicle
  - An method named get_vehicle_name() that returns the vehicle's name in the form make, model (e.g. "Suzuki Swift")
  - A display() method which prints the vehicle's details in the following format:

```
License #: DL9087
Vehicle Name: Suzuki Swift
Price: $15,990.00
```

Exercise

- Create a Notebook that defines a list with the following vehicles, and display the list contents:

License #	Vehicle Name	Price
FG1000	Honda Civic	$21,875
GH7000	Mercedes BClass	$42,700
AY3000	Bugatti Veyron	$1,500,000
HI2000	Hyundai Elantra	$16,950

Class and Instance Variables

- **Instance variable** is a variable for which each object of the class has a separate copy
  - e.g., customer, account_id and balance in the BankAccount class are instance variables
- **Class variable** is shared by all instances of the class
  - They are defined inside the class but outside any of its methods
- Both types of variables are accessed using the object.attr notation
- For example, let's add to our BankAccount class a class variable named currency, and a third method, for printing the balance of the account

Class and Instance Variables

```
class BankAccount:
    """A class representing a bank account"""
    currency = '$' # a class variable

    def __init__(self, customer, account_number, balance=0):
        …

    def deposit(self, amount):
        …

    def withdraw(self, amount):
        …

    def check_balance(self):
        """Print a statement of the account balance."""
        print(f"The balance of account number {self.account_number} is "
              f"{self.currency}{self.balance:.2f}")
```

```
my_account = BankAccount("Joe Smith", 34572881)
my_account.deposit(525)
```

```
my_account.check_balance()
```

```
The balance of account number 34572881 is $525.00
```

Static Methods

- A **static method** belongs to the class rather than an object of the class
- A static method can be invoked without the need for creating an instance of a class
- To declare a static method, use the following syntax:

```
class C:
    @staticmethod
    def f(arg1, arg2, ...): …
```

  - @staticmethod is an example for a function decorator
    - Decorators are functions which modify the functionality of another function
- A static method doesn’t receive an implicit self argument
  - Thus it cannot access instance variables
- A static method can be called either on the class (C.f()) or on an instance (C().f())
- Use static methods sparingly
  - In many cases it’s clearer to define module-level functions than static methods

Static Method Example

- As an example for a static method, let’s add the method transfer_money() to our BankAccount class that will transfer money between two bank accounts:

```
@staticmethod
def transfer_money(account1, account2, amount):
    """Transfer amount from account1 to account2"""
    if account1.withdraw(amount):
        account2.deposit(amount)
```

- We need to modify the withdraw() method to indicate whether it succeeded or not:

```
def withdraw(self, amount):
    """Withdraw amount from the bank account, ensuring there are sufficient funds."""
    if amount > 0:
        if amount > self.balance:
            print("Insufficient funds")
        else:
            self.balance -= amount
            return True
    else:
        print("Invalid withdrawal amount:", amount)
```

Static Method Example

- Example for usage of transfer_money():

```
from bank_account import BankAccount
```

```
account1 = BankAccount("Joe Smith", 34572881, 1000)
account2 = BankAccount("Helen Smith", 5799236)
```

```
BankAccount.transfer_money(account1, account2, 200)
```

```
account1.check_balance()
```

```
The balance of account number 34572881 is $800.00
```

```
account2.check_balance()
```

```
The balance of account number 5799236 is $200.00
```

```
BankAccount.transfer_money(account1, account2, 1000)
```

```
Insufficient funds
```

```
account1.check_balance()
```

```
The balance of account number 34572881 is $800.00
```

Class Methods

- Just like a static method, a class method is bound to the class rather than its object
- However, a class method receives the class as implicit first argument, just like an instance method receives the instance
- To declare a class method, use the following syntax:

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

- A class method can be called either on the class (C.f()) or on an instance (C().f())
- The instance is ignored except for its class
- If a class method is called for a derived class, the derived class object is passed as the implied first argument

Class Methods

- Class methods are typically used as factory methods, which generate objects for different use cases (like constructors)
- This allows you to provide alternate constructors for your class

```
class MyData:
    def __init__(self, data):
        """Initialize MyData from a sequence"""
        self.data = data

    @classmethod
    def from_file(cls, filename):
        """Initialize MyData from a file"""
        data = open(filename).readlines()
        return cls(data)

    @classmethod
    def from_dict(cls, data_dict):
        """Initialize MyData from a dict's items"""
        return cls(list(data_dict.items()))
```

```
from mydata import MyData
```

```
MyData([1, 2, 3]).data
```

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

```
MyData.from_file("test.txt").data
```

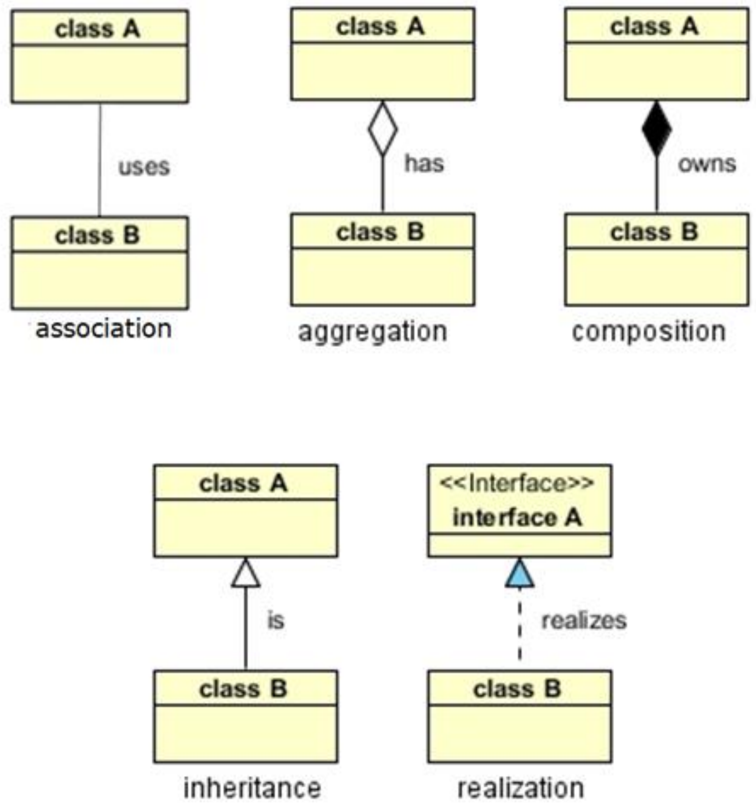
```
['First item\n', 'Second item\n', 'Third item']
```

```
MyData.from_dict({"spam": "ham"}).data
```

```
[('spam', 'ham')]
```

Class Relationships

- There are five key relationships between classes:



Class Relationships in Python

- **Composition** is represented when class A contains an object of class B and is responsible for its lifetime

```python
class B: pass
class A:
    def __init__(self):
        self.b = B()
```

- **Aggregation** is represented when class A stores a reference to class B for later use

```python
class B: pass
class A:
    def __init__(self, b):
        self.b = b

b = B()
a = A(b)
```

- **Association** is represented when a reference to class B is passed as a method parameter to class A

```python
class B: pass
class A:
    def doSomething(self, b):
        pass

b = B()
a = A()
a.doSomething(b)
```

Composition Example

- As an example for composition relationship, we'll define a class Rectangle that is composed of two point objects, which represent its top-left and bottom-right corners
- First, let's define a Point class with attributes x and y:

```
# point.py
class Point:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
```

Composition Example

- Now we'll define a Rectangle class, composed of two point objects:

```
# rectangle.py
from point import Point

class Rectangle:
    def __init__(self, p1, p2):
        self.p1 = Point(p1.x, p1.y)   # top-left corner
        self.p2 = Point(p2.x, p2.y)   # bottom-right corner

    def get_width(self):
        return self.p2.x - self.p1.x

    def get_height(self):
        return self.p1.y - self.p2.y

    def get_area(self):
        return self.get_width() * self.get_height()
```

The rectangle object holds its own copy of the point objects

Composition Example

- Creating a rectangle object and testing its methods:

```
from point import Point
from rectangle import Rectangle
```

```
p1 = Point(3, 10)
p2 = Point(7, 2)
r1 = Rectangle(p1, p2)
```

```
r1.get_width()
```

```
4
```

```
r1.get_height()
```

```
8
```

```
r1.get_area()
```

```
32
```

```
# r1 is not affected from changing p1's coordinates
p1.x = 5
r1.get_width()
```

```
4
```

Aggregation Example

- As an example for aggregation relationship, we'll define a Customer class with attributes name, address, and date of birth, and a method to calculate his/her age
- The BankAccount class will keep a reference to an instance of the Customer class, but will not controls the object's lifetime
- Note that it was possible to instantiate a BankAccount object by passing a string literal as customer
  - This is a consequence of Python's dynamic typing: no check is made that the object passed as an argument to the class constructor is of any particular type

Aggregation Example

- First, let’s define the Customer class:

```python
# customer.py
from datetime import datetime
class Customer:
    """A class representing a bank customer."""
    def __init__(self, name, address, date_of_birth):
        self.name = name
        self.address = address
        self.date_of_birth = datetime.strptime(date_of_birth, '%Y-%m-%d')

    def get_age(self):
        """Calculate and return the customer's age"""
        today = datetime.today()
        try:
            birthday = self.date_of_birth.replace(year=today.year)
        except ValueError:
            # birthday is 29 Feb but today’s year is not a leap year
            birthday = self.date_of_birth.replace(year=today.year,
                                                  day=self.date_of_birth.day - 1)
        if birthday > today:
            return today.year - self.date_of_birth.year - 1
        return today.year - self.date_of_birth.year
```

Aggregation Example

- Then we can pass Customer objects to our BankAccount constructor:

```
from bank_account import BankAccount
from customer import Customer
```

```
customer1 = Customer("Helen Smith",
                     "76 The Warren, Blandings, Sussex",
                     "1976-02-29")
```

```
account1 = BankAccount(customer1, 2145288, 1000)
```

```
account1.customer.get_age()
```

```
42
```

```
print(account1.customer.address)
```

```
76 The Warren, Blandings, Sussex
```

Class DocStrings

- The docstring for a class should summarize its behavior and list the public methods and instance variables
- The class constructor should be documented in the docstring for its __init__ method
- Individual methods should be documented by their own docstring

Operator Overloading

- Operator overloading lets you redefine the meaning of operators with respect to your own classes
- For example, operator overloading allow us to use the operator + to add two integers, as well as two string objects
  - i.e, the + operator is **overloaded** for the int class and the str class
- Operator overloading is achieved by defining a special method in the class definition
- The names of these methods start and end with double underscores (__)
  - For example, both the int class and str class implement the __add__() method
  - These methods are not private since they have both leading underscores and trailing underscores.

```
x, y = 10, 20
```

```
x.__add__(y)   # the same as x + y
```

```
30
```

Operator Overloading

- The following table lists the operators and their corresponding special methods:

Operator	Special Method	Description
+	__add__(self, other)	Addition
-	__sub__(self, other)	Subtraction
*	__mul__(self, other)	Multiplication
/	__truediv__(self, other)	Division
//	__floordiv__(self, other)	Integer Division
%	__mod__(self, other)	Modulus
**	__pow__(self, other)	Exponentiation
==	__eq__(self, other)	Equal to
!=	__ne__(self, other)	Not equal to
>	__gt__(self, other)	Greater than
>=	__lt__(self, other)	Greater than or equal to

Operator	Special Method	Description
<	__lt__(self, other)	Less than
<=	__le__(self, other)	Less than or equal to
in	__contains__(self, value)	Membership operator
[index]	__getitem__(self, index)	Index operator
len()	__len__(self)	Number of elements
str()	__str__(self)	The string representation

Operator Overloading

- The following Point class shows how you can overload operators in a class:

```python
# point.py
import math
class Point:
    def __init__(self, x = 0, y = 0):
        self.__x = x
        self.__y = y

    def get_x(self):
        return self.__x

    def set_x(self, x):
        self.__x = x

    def get_y(self):
        return self.__y

    def set_y(self, y):
        self.__y = y

    def dist_from_origin(self):
        return math.sqrt(self.__x ** 2 + self.__y * 2)
```

Operator Overloading

```python
def __add__(self, other_point):
    return Point(self.__x + other_point.__x, self.__y + other_point.__y)

def __sub__(self, other_point):
    return Point(self.__x - other_point.__x, self.__y - other_point.__y)

def __eq__(self, other_point):
    return self.__x == other_point.__x and self.__y == other_point.__y

def __gt__(self, other_point):
    return self.dist_from_origin() > other_point.dist_from_origin()

def __ge__(self, other_point):
    return self.dist_from_origin() >= other_point.dist_from_origin()

def __lt__(self, other_point):
    return self.dist_from_origin() < other_point.dist_from_origin()

def __le__(self, other_point):
    return self.dist_from_origin() <= other_point.dist_from_origin()

def __str__(self):
    return "({}, {})".format(self.__x, self.__y)
```

Operator Overloading

- Testing the class in the main script:

```
p1 = Point(4, 7)
p2 = Point(10, 6)

print("p1:", p1)
print("p2:", p2)
print("Is p1 == p2 ?", p1 == p2)
print("Is p1 > p2 ?", p1 > p2)
print("Is p1 >= p2 ?", p1 >= p2)
print("Is p1 < p2 ?", p1 < p2)
print("Is p1 <= p2 ?", p1 <= p2)

p3 = p1 + p2
p4 = p1 - p2

print()
print("p3:", p3)
print("p4:", p4)
```

Output

```
p1: (4, 7)
p2: (10, 6)
Is p1 == p2 ? False
Is p1 > p2 ? False
Is p1 >= p2 ? False
Is p1 < p2 ? True
Is p1 <= p2 ? True

p3: (14, 13)
p4: (-6, 1)
```

Compound Assignment Operators

- Special methods that start with __i, such as __iadd__, __isub__, implement the augmented arithmetic assignments (+=, -=, *=, /=, //=, %=, **=)
- These methods should attempt to do the operation in-place (modifying *self*) and return the result (which is typically *self*)
- For example, let's define the operator += in the Point class:

```
def __iadd__(self, other_point):
    self.__x += other_point.__x
    self.__y += other_point.__y
    return self
```

```
p7 = Point(5, 2)
p8 = Point(3, 4)
print("p7:", p7)
print("p8:", p7)
p7 += p8
print("p7:", p7)
```

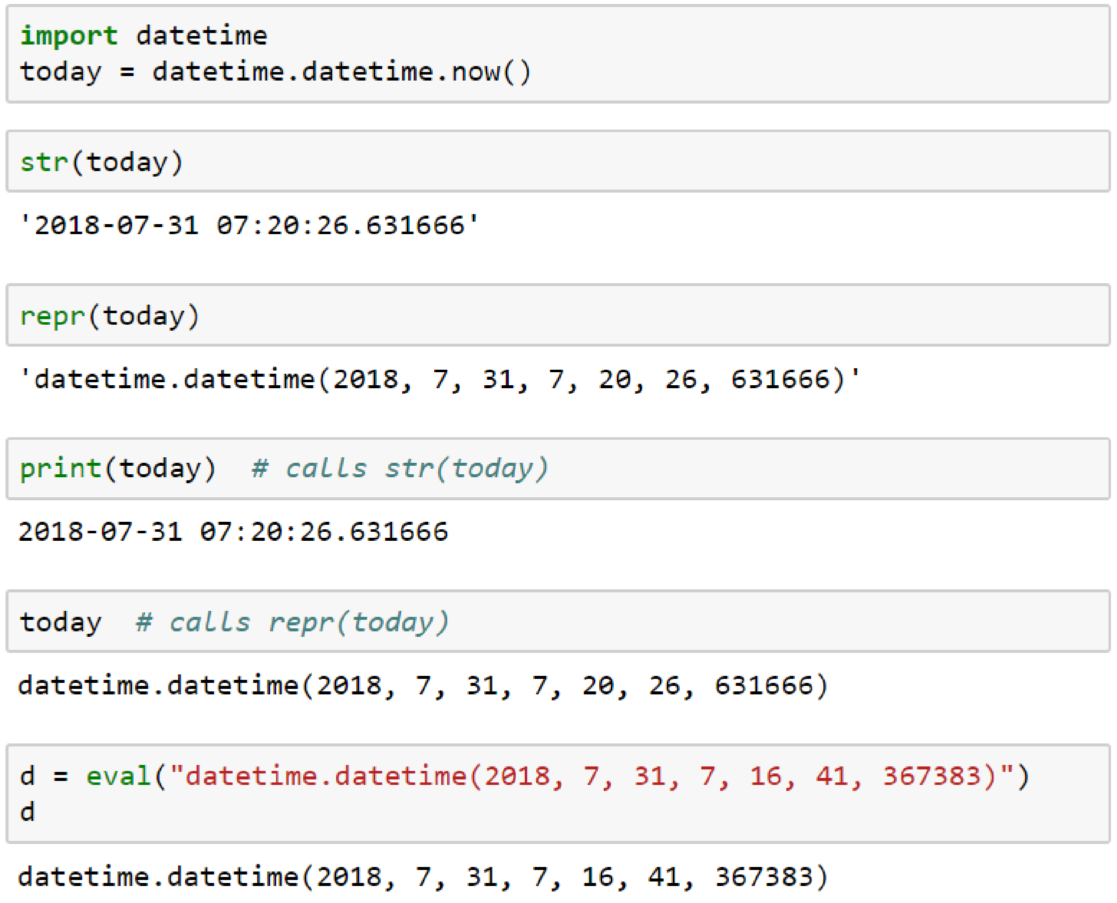
```
p7: (5, 2)
p8: (5, 2)
p7: (8, 6)
```

String Representations

- The special method **__repr__()** is called by the repr() built-in function to compute the "official" string representation of an object
- This is typically used for debugging, so it is important that the representation is information-rich and unambiguous
- It should provide enough information that could be used to recreate an object with the same value, so eval(repr(obj)) == obj
  - eval(*expression*) is a built-in function that takes a string, evaluates it as a Python expression and returns the result of the evaluated expression
- __repr__() is intended to be unabmiguous, while __str__() is intended to be readable
- If a class defines __repr__() but not __str__(), then __repr__() is also used when an "informal" string representation of instances of that class is required

String Representations

- Example for the differences between __str__ and __repr__:



```
import datetime
today = datetime.datetime.now()
```

```
str(today)
```

```
'2018-07-31 07:20:26.631666'
```

```
repr(today)
```

```
'datetime.datetime(2018, 7, 31, 7, 20, 26, 631666)'
```

```
print(today)  # calls str(today)
```

```
2018-07-31 07:20:26.631666
```

```
today  # calls repr(today)
```

```
datetime.datetime(2018, 7, 31, 7, 20, 26, 631666)
```

```
d = eval("datetime.datetime(2018, 7, 31, 7, 16, 41, 367383)")
d
```

```
datetime.datetime(2018, 7, 31, 7, 16, 41, 367383)
```

String Representations

- Let’s implement the __repr__ method in our Point class:

```
def __repr__(self):
    return "Point({}, {})".format(self.__x, self.__y)
```

```
p9 = Point(1, 2)
print("p9:", p9)
s = repr(p9)
print(s)
p10 = eval(s)
print("p10:", p10)
```

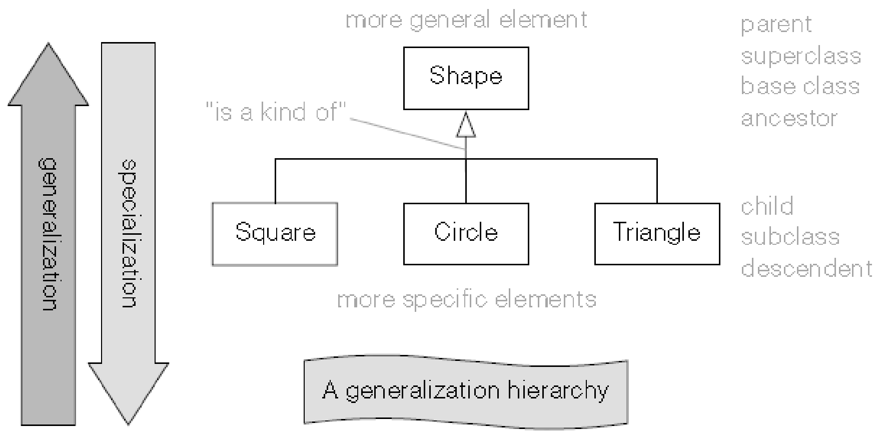
```
p9: (1, 2)
Point(1, 2)
p10: (1, 2)
```

Exercise

- Create a class named Rational that represents a rational number
  - a rational number is any number that can be expressed as the fraction *p/q* of two integers: a numerator *p* and a non-zero denominator *q*
- The class should support the following operators:
  - Addition, subtraction, multiplication, and division of two rationals
  - Adding an integer to a rational (both x + p/q and p/q + x should be supported)
  - The compound assignment operator +=
  - Checking if two rationals are equal
    - Note that 1/2 = 2/4 = 4/8, etc.
- Provide both informal and formal string representations for Rational

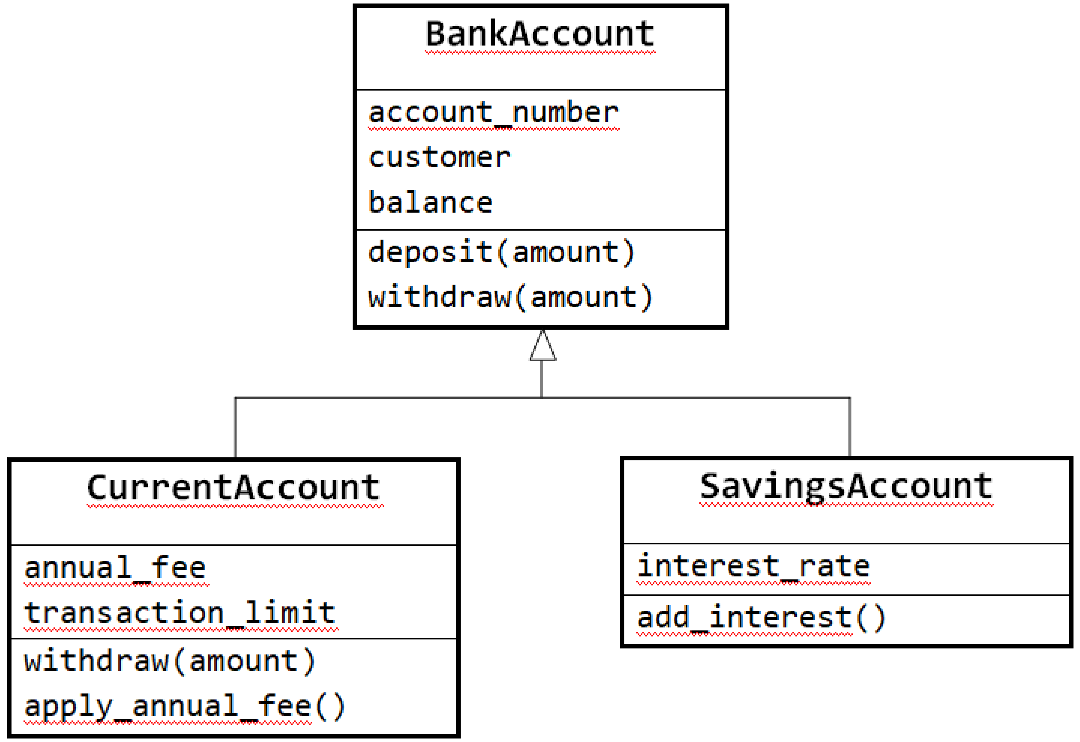
Class Inheritance

- In OOP, **inheritance** enables new objects to take on the properties of existing objects
- A class that is used as the basis for inheritance is called a **superclass** or **base class**
- A class that inherits from a superclass is called a **subclass** or **derived class**
- Typically, a general type of object is defined by a base class, and then customized classes with more specialized functionality are derived from it
- The UML graphical representation of generalization is a hollow triangle shape on the superclass end of the line (or tree of lines) that connects it to one or more subtypes



Class Inheritance

- We'll create the following class hierarchy:



Class Inheritance in Python

- The syntax for a derived class definition looks like this:

```
class DerivedClassName(BaseClass):
    <statement-1>
    …
    <statement-N>
```

- They can be defined in the same file that defines
- BankAccount or in a different Python file which imports BankAccount.

Class Inheritance Demo

- Let’s start with the SavingsAccount class:

```
from bank_account import BankAccount

class SavingsAccount(BankAccount):
    """A class representing a savings account"""

    def __init__(self, customer, account_number, interest_rate, balance=0):
        """Initialize the savings account"""
        super().__init__(customer, account_number, balance)
        self.interest_rate = interest_rate

    def add_interest(self):
        """Add interest to the account at the rate self.interest_rate"""
        self.balance *= (1 + self.interest_rate / 100)
```

SavingsAccount overrides the __init__ method to allow the interest_rate to be initialized

The new __ init __ method calls the base class’s __ init __ method in order to set the other attributes

The SavingsAccount class adds a new attribute, interest_rate, and a new method, add_interest() to its base class

The built-in function super() allows us to refer to the parent base class.
This is useful for accessing inherited methods that have been overridden in a class.

Class Inheritance Demo

- Our new SavingsAccount might be used as follows:

```
from savings_account import SavingsAccount
```

```
my_savings = SavingsAccount("Matthew Walsh", 41522887, 3.5, 1000)
```

```
my_savings.check_balance()
```

```
The balance of account number 41522887 is $1000.00
```

```
my_savings.add_interest()
```

```
my_savings.check_balance()
```

```
The balance of account number 41522887 is $1035.00
```

Class Inheritance Demo

- The second subclass, CurrentAccount, has a similar structure:

```
from bank_account import BankAccount

class CurrentAccount(BankAccount):
    """A class representing a current (checking) account"""

    def __init__(self, customer, account_number, annual_fee,
                 transaction_limit, balance=0):
        """Initialize the current acount"""
        super().__init__(customer, account_number, balance)
        self.annual_fee = annual_fee
        self.transaction_limit = transaction_limit

    def apply_annual_fee(self):
        """Deduct the annual fee from the account balance."""
        self.balance = max(0, self.balance - self.annual_fee)
```

Class Inheritance Demo

```python
def withdraw(self, amount):
    """Withdraw amount if sufficient funds exist in the account
       and amount is less than the transaction limit"""
    if amount <= 0:
        print("Invalid withdrawal amount")
        return

    if amount > self.balance:
        print("Insufficient funds")
        return

    if amount > self.transaction_limit:
        print(f"{self.currency}{amount:.2f} exceeds the transaction limit of "
              f"{self.currency}{self.transaction_limit:.2f}")
        return

    self.balance -= amount
```

Class Inheritance Demo

- Note what happens if we call withdraw on a CurrentAccount object:

```
from current_account import CurrentAccount
```

```
my_current = CurrentAccount("Alison Wicks", 78300991, 20, 200)
```

```
my_current.withdraw(220)
```

```
Insufficient funds
```

```
my_current.deposit(750)
my_current.check_balance()
```

```
The balance of account number 78300991 is $750.00
```

```
my_current.withdraw(220)
```

```
$220.00 exceeds the transaction limit of $200.00
```

  - The withdraw method called is that of the CurrentAccount class, as this method overrides that of the same name in the base class, BankAccount

The type() Function

- The type(*object*) built-in function returns the type of an *object*
- It is essentially the same as *object*.__class__
- Examples:

```
type(my_savings)
```

```
savings_account.SavingsAccount
```

```
type("hello")
```

```
str
```

isinstance()

- The **isinstance**(*object*, *classinfo*) returns true if the *object* argument is an instance of the *classinfo* argument, or a subclass of it
- This function is recommended for testing the type of an object, because it takes subclasses into account
- Examples:

```
isinstance(my_savings, SavingsAccount)
```

```
True
```

```
isinstance(my_savings, BankAccount)
```

```
True
```

```
isinstance(my_savings, CurrentAccount)
```

```
False
```

issubclass()

- A natural complement to isinstance() is the ability to determine whether one class has another class somewhere in its inheritance chain
- The **issubclass**(*class, classinfo*) returns true if *class* is a subclass of *classinfo*
- A class is considered a subclass of itself

```
issubclass(SavingsAccount, BankAccount)
```

```
True
```

```
issubclass(int, object)
```

```
True
```

- The following relationship between isinstnace() and issubclass() is always true:

```
isinstance(obj, cls) == issubclass(type(obj), cls)
```

Exercise

- Write the class VehicleForSale as a subclass of Vehicle
- In addition to the attributes that are declared in the base class, VehicleForSale should have a depreciation value
- It should also have a method called setDepreciation() that accepts the percentage rate, calculates the depreciation value (which is rate * price of vehicle) and set this attribute on the object
- Override the __str__ method that formats its returned value in the following manner:

Abstract Base Classes

- **Abstract classes** are classes that contain one or more abstract methods
- **An abstract method** is a method that is declared, but contains no implementation
- Abstract classes cannot be instantiated, and require subclasses to provide implementations for the abstract methods
- A class that is derived from an abstract class cannot be instantiated, unless all of its abstract methods are overridden
- The module **abc** provides the infrastructure for defining **Abstract Base Classes** (ABCs)
- In the following example, we'll define the class Shape as an abstract base class

Abstract Base Classes

- The Shape base class:

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    def __init__(self, location, color):
        self.location = location # of type Point
        self.color = color

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

    @abstractmethod
    def get_area(self):
        pass

    @abstractmethod
    def display(self):
        print(f"Location: ({self.location.x}, {self.location.y})")
        print(f"Color: {self.color}")
        print(f"Area: {self.get_area():.3f}")
```

An abstract method can provide some basic functionality in the abstract class.
Subclasses will still be forced to override the method's implementation.
However, they will be able to invoke the abstract method using super().

Abstract Base Classes

- Trying to create a Shape object will raise an exception that Shape can’t be instantiated:

```
from point import Point
from shape import Shape
```

```
s = Shape(Point(2, 5), "blue")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-15403aa0c3b5> in <module>()
----> 1 s = Shape(Point(2, 5), "blue")

TypeError: Can't instantiate abstract class Shape with abstract methods draw,
 get_area
```

Abstract Base Classes

- We'll now define a Circle subclass of Shape
- Circle overrides Shape's abstract methods, which turns it into a **concrete class** (a class that can be instantiated)

```python
from shape import Shape
import math

class Circle(Shape):
    def __init__(self, location, color, radius):
        super().__init__(location, color)
        self.radius = radius

    def get_area(self):
        return math.pi * self.radius ** 2

    def display(self):
        super().display()
        print(f"Radius: {self.radius}")
```

Abstract Base Classes

- Now it is possible to create Circle objects:

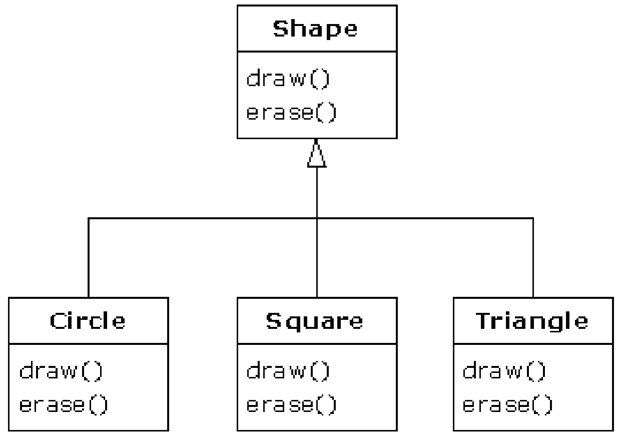
```
from circle import Circle
c = Circle(Point(3, 4), "red", 2)
```

```
c.display()
```

```
Location: (3, 4)
Color: red
Area: 12.566
Radius: 2
```

Polymorphism

- **Polymorphism** is the ability of objects of different types to respond, each in its own way, to identical messages (method calls)
- For example, all shapes will draw when calling their draw() method, but each draw() is implemented in a different way



- Polymorphism allows for flexibility and loose coupling so that code can be extended and easily maintained over time

Exercise

- Write a program that gets an input file of zoo animal data and interactively answers queries from a user
- The first line of each animal data record in the input file is in the following format:

```
Name AnimalType Species Mass
```

  - Name: The animal's name. Contains no white space.
  - AnimalType: The type of animal. One of: Mammal, Reptile, Bird.
  - Species: The animal's species. Contains no white space.
  - Mass: The animal's mass (weight) in kgs. An integer number ≥ 1.
- Here are examples of first lines of animal data records:

```
Bob Mammal Bear 150
Lucy Reptile Lizard 1
Oliver Bird Ostrich 35
```

Exercise

- The data following the first line in each animal record depends on the AnimalType
- Mammals:
  - The first line of data for animals of type Mammal is followed by: LitterSize, which is the average number of offspring the mammal has (an integer number ≥ 1)
- Reptiles:
  - The first line of data for animals of type Reptile is followed by: VenomousOrNot, which indicates whether the reptile has a venomous bite
- Birds:
  - The first line of data for animals of type Bird is followed by: Wingspan (the wingspan of the bird in centimeters), and TalksOrMute (indicates whether the bird talks)
  - Birds that talk have an additional line of data: Phrase, which is what the bird says
    - May contain whitespace, but is no more than one line long.

Exercise

- Here is an example of a data file to be read by the program:

```
Bob Mammal Bear 300
2
Lucy Reptile Lizard 2
Nonvenomous
Carl Reptile Cottonmouth 3
Venomous
Oliver Bird Ostrich 75
60 Mute
Polly Bird Parrot 1
2 Talks
I want a cracker
Doug Mammal Dog 20
4
```

Exercise

- Once the program has read in the data file, it should request and process interactive queries from the user
- It should request queries in the following format:
- Query animal species[s], mass[m], litter[l], venom[v], wingspan[w], talk[t] or exit session[e]?
- The following example demonstrates all varieties of queries and responses, using the data provided in the above example file
- The program should gracefully handle cases in which a user does not respond with one of s, m, l, v, w, t, or e to the program's query request, or provides the name of an animal not in the database

Exercise

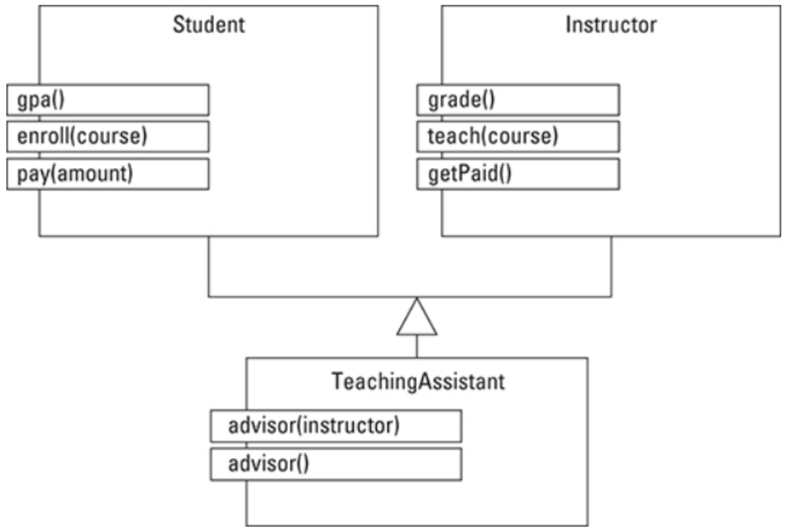
```
Query animal species[s], mass[m], litter[l], venom[v], wingspan[w], talk[t] or exit session[e]? s
Animal Name? Bob
Bob species is Bear
Query animal species[s], mass[m], litter[l], venom[v], wingspan[w], talk[t] or exit session[e]? m
Animal Name? Lucy
Lucy mass is 2
Query animal species[s], mass[m], litter[l], venom[v], wingspan[w], talk[t] or exit session[e]? l
Animal Name? Bob
Bob litter size is 2
Query animal species[s], mass[m], litter[l], venom[v], wingspan[w], talk[t] or exit session[e]? v
Animal Name? Carl
Carl is Venomous
Query animal species[s], mass[m], litter[l], venom[v], wingspan[w], talk[t] or exit session[e]? v
Animal Name? Lucy
Lucy is Nonvenomous
Query animal species[s], mass[m], litter[l], venom[v], wingspan[w], talk[t] or exit session[e]? w
Animal Name? Oliver
Oliver Wing Span is 60
Query animal species[s], mass[m], litter[l], venom[v], wingspan[w], talk[t] or exit session[e]? t
Animal Name? Oliver
Oliver is Mute
Query animal species[s], mass[m], litter[l], venom[v], wingspan[w], talk[t] or exit session[e]? t
Animal Name? Polly
Polly talks.
Polly says I want a cracker
Query animal species[s], mass[m], litter[l], venom[v], wingspan[w], talk[t] or exit session[e]? e
Goodbye!
```

Multiple Inheritance

- Python also supports a form of multiple inheritance
- A class definition with multiple base classes looks like this:

```
class DerivedClassName(BaseClass1, BaseClass2, ...):
    <statement-1>
    …
    <statement-N>
```

- For example, most universities offer jobs to graduate students as teaching assistants
- These individuals are still students and have all the properties of a student — but, in addition, they are also afforded the features of an instructor
- Thus, logically, their inheritance tree looks like this:



Multiple Inheritance

- Let’s build this hierarchy in Python
- We’ll start with the Student class definition:

```python
class Student:
    """A class representing a student"""
    def __init__(self, id, name):
        self.id = id
        self.name = name
        self.grades = []

    def add_grade(self, grade):
        self.grades.append(grade)

    def grades_average(self):
        return sum(self.grades) / len(self.grades)
```

Multiple Inheritance

- We now define the Instructor class:

```
class Instructor:
    """A class representing an instructor"""
    def __init__(self, id, name, salary):
        self.id = id
        self.name = name
        self.salary = salary
        self.courses = []

    def add_course(self, course):
        self.courses.append(course)

    def get_courses(self):
        return self.courses
```

Multiple Inheritance

- Lastly, TeachingAssistant inherits both from “Student” and “Instructor”:

```
from student import Student
from instructor import Instructor

class TeachingAssistant(Student, Instructor):
    """A class representing a teaching assistant"""
    def __init__(self, id, name, salary, advisor):
        Student.__init__(self, id, name)
        Instructor.__init__(self, id, name, salary)
        self.advisor = advisor
```

Calling the __init__() method of both parents

Multiple Inheritance

- A sample test:

```
from ta import TeachingAssistant
```

```
ta1 = TeachingAssistant(3358123, "John Smith", 10000, "Ellen DeGeneres")
```

```
ta1.add_grade(80)
ta1.add_grade(90)
ta1.grades_average()
```

```
85.0
```

```
ta1.add_course("Algebra")
ta1.add_course("Python")
ta1.add_course("C++")
ta1.get_courses()
```

```
['Algebra', 'Python', 'C++']
```

Iterables and Iterators

- An object is considered iterable if it can yield objects one at a time, typically within a for loop
- In Python, an object is iterable if passing it into the **iter()** function returns an **iterator**
- Internally, iter() inspects the object passed in, looking for an __iter__() method
- If such a method is found, it's called without any arguments and is expected to return an iterator
- For example, since a list is iterable, calling iter() on a list returns its iterator:

```
list1 = iter([1, 2, 3])
list1
```

```
<list_iterator at 0x2141d5c3e10>
```

- Each time we call the **__next__** method on the iterator, it gives us the next element
- If there are no more elements, it raises a *StopIteration*

Iterables and Iterators

```
it.__next__()
```

```
1
```

```
it.__next__()
```

```
2
```

```
it.__next__()
```

```
3
```

```
it.__next__()
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-9-74e64ed6c80d> in <module>()
----> 1 it.__next__()

StopIteration:
```

Iterables and Iterators

- Iterators are implemented as classes
- The required interface consists of just two methods:
- **__iter__()** – iterators should be iterable on their own as well
  - Typically this method just returns self
- **__next__()** – this method returns a new value for the next pass in the loop
  - When there are no more items to provide, the method should raise a StopIteration exception
  - When this is raised, the loop is considered complete, and execution resumes on the next line after the end of the loop

Iterables and Iterators

- For example, let's create a class Range which provides a similar functionality as the built-in range() function

```python
class Range:
    def __init__(self, count):
        self.count = count

    def __iter__(self):
        return RangeIter(self.count)

class RangeIter:
    def __init__(self, count):
        self.count = count
        self.current = 0

    def __iter__(self):
        return self

    def __next__(self):
        value = self.current
        self.current += 1
        if self.current > self.count:
            raise StopIteration
        return value
```

Iterables and Iterators

- We can now use the Range class as the built-in range() function:

```python
from myrange import Range
```

```python
r = Range(5)
```

```python
list(r)
```

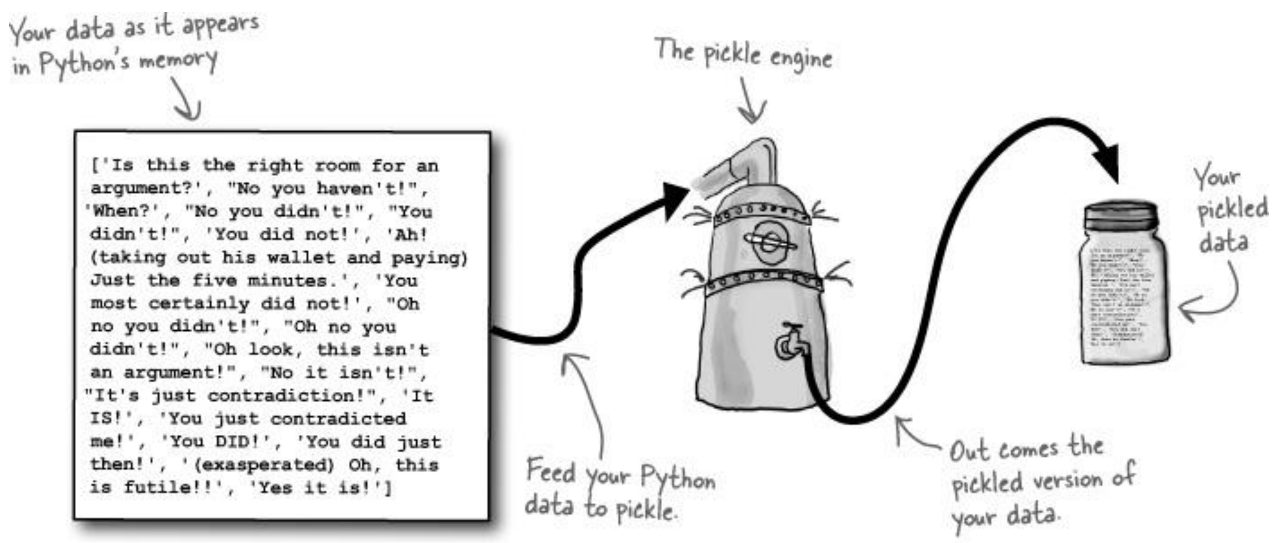
```
[0, 1, 2, 3, 4]
```

```python
for i in r:
    print(i, end=" ")
```

```
0 1 2 3 4
```

Pickling

- Pickling is the name of the object serialization process in Python
- By pickling we can convert an object hierarchy to a binary format that can be stored
- Unpickling is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy


Your data as it appears in Python's memory
The pickle engine
Your pickled data
Feed your Python data to pickle.
Out comes the pickled version of your data.

```
['Is this the right room for an
argument?', "No you haven't!",
'When?', "No you didn't!", "You
didn't!", 'You did not!', 'Ah!
(taking out his wallet and paying)
Just the five minutes.', 'You
most certainly did not!', "Oh
no you didn't!", "Oh no you
didn't!", "Oh look, this isn't
an argument!", "No it isn't!",
"It's just contradiction!", 'It
IS!', 'You just contradicted
me!', 'You DID!', 'You did just
then!', '(exasperated) Oh, this
is futile!!', 'Yes it is!']
```

Pickling Objects

- To pickle an object we just need to import the pickle module and call the **dumps()** function passing the object to be pickled as a parameter:

```
import pickle
```

```
pickle.dumps(1)
```

```
b'\x80\x03K\x01.'
```

```
pickle.dumps([1, 2, 3])
```

```
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

  - The pickled output contains more information than the original value, because it also needs to store the type, so the object can be reconstituted later

- To store the pickled object in a file, call the **dump()** function, which also takes an opened binary file in which the pickling result will be stored automatically

```
with open("mydata.bin", "wb") as my_data:
    pickle.dump([1, 2, 3], my_data)
```

Unpickling Objects

- The unpickling process is done by using the **load()** or **loads()** functions of the pickle module, which return a complete object hierarchy from a simple bytes array
- The difference between the two is similar to the dump functions: load() accepts a readable file object, while loads() accepts a string

```
pickled = pickle.dumps(1)
pickled
```

```
b'\x80\x03K\x01.'
```

```
pickle.loads(pickled)
```

```
1
```

```
with open("mydata.bin", "rb") as my_data:
    my_list = pickle.load(my_data)
print(my_list)
```

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

- The same pickling and unpickling process can be applied to your own objects

Unpicklable Objects

- Not every object is picklable
- Some objects like db connections and handles to opened files can’t be pickled, and trying to pickle them raises a pickle.PickleError exception
- If there are unpicklable objects in the hierarchy of the object you want to pickle, this prevents you from serializing (and storing) the entire object
- Other attributes that should avoid being pickled are initialization values, system-specific details, and other transient information which is not part of the object’s value directly
- Fortunately, Python offers you a way to specify what you want to pickle and how to unpickle

Customizing Pickling

- Python provides two special methods to control how individual objects are pickled and restored
- **__getstate__**() controls what gets included in the pickled value
  - For complex objects, the value will typically be a dictionary or a tuple
  - If __getstate__() is absent, the instance's __dict__ is pickled as usual
- **__setstate__**(*state*) is called upon unpickling, and accepts the state of the object to restore
  - The state is exactly the same value that was returned from __getstate__()
  - If __setstate__() is absent, the pickled state must be a dictionary and its items are assigned to the new instance's dictionary
- If __getstate__() returns a false value, the __setstate__() method will not be called upon unpickling

Customizing Pickling Example

- For example, a currency conversion class might contain the current amount as well as a string to indicate the currency being represented
- In addition, it would likely have access to a dictionary of current exchange rates, so that it can convert the amount to a different currency
- Because the currency conversion values aren’t specific to the instance at hand, and they’ll change over time anyway, there’s no reason to store them in the pickled string
- Thus, we can use __getstate__() to provide just those values that are actually important

Customizing Pickling Example

```python
class Money:
    def __init__(self, amount, currency):
        self.amount = amount
        self.currency = currency
        self.conversion = { 'USD': 1, 'CAD': 1.31, 'EUR': 0.87 }

    def convert(self, currency):
        ratio = self.conversion[currency] / self.conversion[self.currency]
        return Money(self.amount * ratio, currency)

    def __str__(self):
        return f"{self.amount:.2f} {self.currency}"

    def __repr__(self):
        return f"Money({self.amount} {self.currency})"

    def __getstate__(self):
        return self.amount, self.currency

    def __setstate__(self, state):
        self.amount = state[0]
        self.currency = state[1]
```

Customizing Pickling Example

- Sample usage:

```
from money import Money
```

```
my_money = Money(250, "USD")
print(my_money)
print(my_money.convert("EUR"))
```

```
250.00 USD
217.50 EUR
```

```
with open("money.bin", "wb") as my_file:
    pickle.dump(my_money, my_file)
```

```
with open("money.bin", "rb") as my_file:
    saved_money = pickle.load(my_file)
```

```
print(saved_money)
```

```
250.00 USD
```