Advanced Python

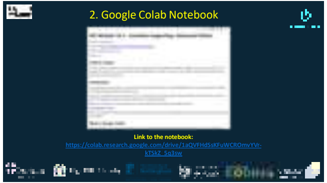
Module 7A.2. Part 4

1. What We'll Cover

In Part 4 we will move onto some advanced topics,

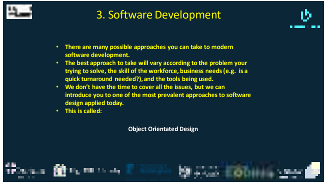
- Object-orientated Programming.
- Classes in Python.
- Instance variables & functions.
- Objects and their relationship to classes.
- Inheritance in Python.
- UML Diagrams.

The aim: to get you writing sophisticated programs by yourselves, using modern programming methods.



2. Google Colab Notebook

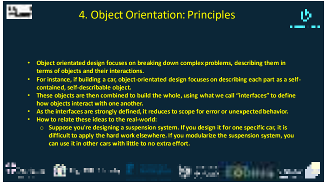
Link to the notebook:
https://colab.research.google.com/drive/1aQVFHdSsKFuWCROmvYVr-kTSkZ_5q3sw



3. Software Development

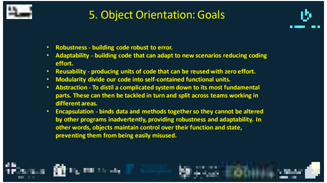
- There are many possible approaches you can take to modern software development.
- The best approach to take will vary according to the problem your trying to solve, the skill of the workforce, business needs (e.g. is a quick turnaround needed?), and the tools being used.
- We don’t have the time to cover all the issues, but we can introduce you to one of the most prevalent approaches to software design applied today.
- This is called:

Object Orientated Design



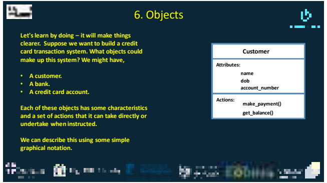
4. Object Orientation: Principles

- Object orientated design focuses on breaking down complex problems, describing them in terms of objects and their interactions.
- For instance, if building a car, object-orientated design focuses on describing each part as a self-contained, self-describable object.
- These objects are then combined to build the whole, using what we call “interfaces” to define how objects interact with one another.
- As the interfaces are strongly defined, it reduces to scope for error or unexpected behavior.
- How to relate these ideas to the real-world:
  - Suppose you're designing a suspension system. If you design it for one specific car, it is difficult to apply the hard work elsewhere. If you modularize the suspension system, you can use it in other cars with little to no extra effort.



5. Object Orientation: Goals

- Robustness - building code robust to error.
- Adaptability - building code that can adapt to new scenarios reducing coding effort.
- Reusability - producing units of code that can be reused with zero effort.
- Modularity divide our code into self-contained functional units.
- Abstraction - To distil a complicated system down to its most fundamental parts. These can then be tackled in turn and split across teams working in different areas.
- Encapsulation - binds data and methods together so they cannot be altered by other programs inadvertently, providing robustness and adaptability. In other words, objects maintain control over their function and state, preventing them from being easily misused.



6. Objects

Let's learn by doing – it will make things clearer. Suppose we want to build a credit card transaction system. What objects could make up this system? We might have,

- A customer.
- A bank.
- A credit card account.

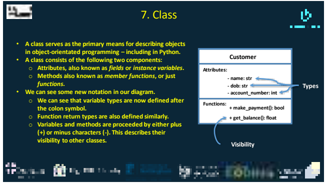
Each of these objects has some characteristics and a set of actions that it can take directly or undertake when instructed.

We can describe this using some simple graphical notation.

Customer

Attributes:
name
dob
account_number

Actions:
make_payment()
get_balance()



7. Class

- A class serves as the primary means for describing objects in object-orientated programming – including in Python.
- A class consists of the following two components:
  - Attributes, also known as *fields* or *instance variables*.
  - Methods also known as *member functions*, or just *functions*.
- We can see some new notation in our diagram.
  - We can see that variable types are now defined after the colon symbol.
  - Function return types are also defined similarly.
  - Variables and methods are proceeded by either plus (+) or minus characters (-). This describes their visibility to other classes.

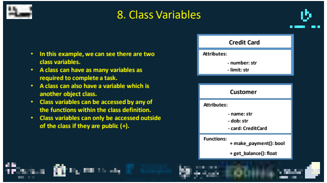
Customer

Attributes:
- name: str
- dob: str
- account_number: int

Functions:
+ make_payment(): bool
+ get_balance(): float

Types

Visibility



8. Class Variables

- In this example, we can see there are two class variables.
- A class can have as many variables as required to complete a task.
- A class can also have a variable which is another object class.
- Class variables can be accessed by any of the functions within the class definition.
- Class variables can only be accessed outside of the class if they are public (+).

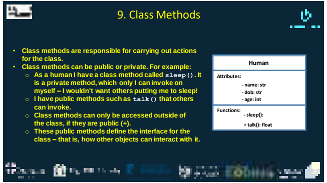
Credit Card

Attributes:
- number: str
- limit: str

Customer

Attributes:
- name: str
- dob: str
- card: CreditCard

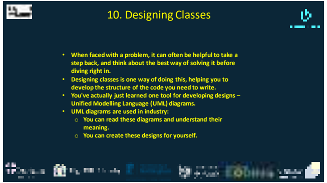
Functions:
+ make_payment(): bool
+ get_balance(): float



9. Class Methods

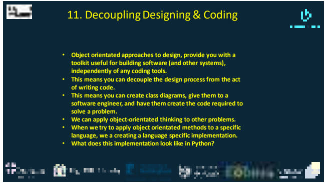
- Class methods are responsible for carrying out actions for the class.
- Class methods can be public or private. For example:
  - As a human I have a class method called `sleep()`. It is a private method, which only I can invoke on myself – I wouldn't want others putting me to sleep!
  - I have public methods such as `talk()` that others can invoke.
  - Class methods can only be accessed outside of the class, if they are public (+).
  - These public methods define the interface for the class – that is, how other objects can interact with it.

Human
Attributes:
- name: str
- dob: str
- age: int
Functions:
- sleep():
+ talk(): float



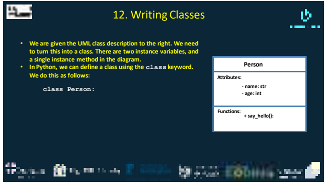
10. Designing Classes

- When faced with a problem, it can often be helpful to take a step back, and think about the best way of solving it before diving right in.
- Designing classes is one way of doing this, helping you to develop the structure of the code you need to write.
- You've actually just learned one tool for developing designs – Unified Modelling Language (UML) diagrams.
- UML diagrams are used in industry:
  - You can read these diagrams and understand their meaning.
  - You can create these designs for yourself.



11. Decoupling Designing & Coding

- Object orientated approaches to design, provide you with a toolkit useful for building software (and other systems), independently of any coding tools.
- This means you can decouple the design process from the act of writing code.
- This means you can create class diagrams, give them to a software engineer, and have them create the code required to solve a problem.
- We can apply object-orientated thinking to other problems.
- When we try to apply object orientated methods to a specific language, we a creating a language specific implementation.
- What does this implementation look like in Python?



12. Writing Classes

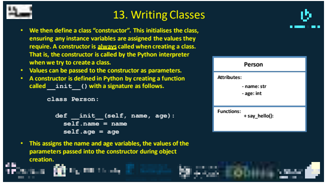
- We are given the UML class description to the right. We need to turn this into a class. There are two instance variables, and a single instance method in the diagram.
- In Python, we can define a class using the `class` keyword. We do this as follows:

```
class Person:
```

Person

Attributes:
- name: str
- age: int

Functions:
+ say_hello():



13. Writing Classes

- We then define a class “constructor”. This initialises the class, ensuring any instance variables are assigned the values they require. A constructor is always called when creating a class. That is, the constructor is called by the Python interpreter when we try to create a class.
- Values can be passed to the constructor as parameters.
- A constructor is defined in Python by creating a function called `__init__()` with a signature as follows.

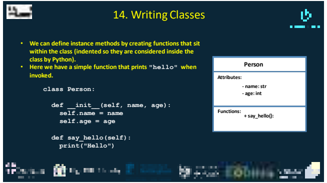
```
class Person:

  def __init__(self, name, age):
    self.name = name
    self.age = age
```

- This assigns the name and age variables, the values of the parameters passed into the constructor during object creation.

Person

Attributes:
- name: str
- age: int

Functions:
+ say_hello():



14. Writing Classes

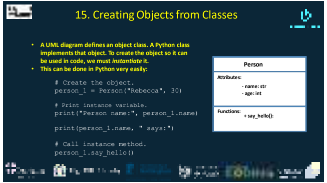
- We can define instance methods by creating functions that sit within the class (indented so they are considered inside the class by Python).
- Here we have a simple function that prints `"hello"` when invoked.

```
class Person:

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

  def say_hello(self):
    print("Hello")
```

Person

Attributes:
- name: str
- age: int

Functions:
+ say_hello():



15. Creating Objects from Classes

- **A UML diagram defines an object class. A Python class implements that object. To create the object so it can be used in code, we must *instantiate* it.**
- **This can be done in Python very easily:**

```
# Create the object.
person_1 = Person("Rebecca", 30)

# Print instance variable.
print("Person name:", person_1.name)

print(person_1.name, " says:")

# Call instance method.
person_1.say_hello()
```

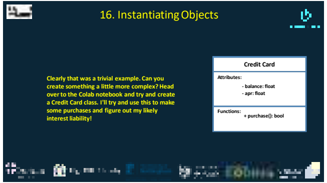
Person

Attributes:

- name: str
- age: int

Functions:

+ say_hello():



16. Instantiating Objects

Clearly that was a trivial example. Can you create something a little more complex? Head over to the Colab notebook and try and create a Credit Card class. I'll try and use this to make some purchases and figure out my likely interest liability!

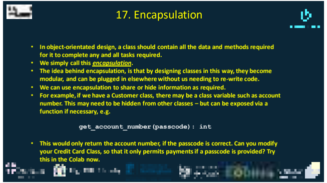
Credit Card

Attributes:

- balance: float
- apr: float

Functions:

+ purchase(): bool

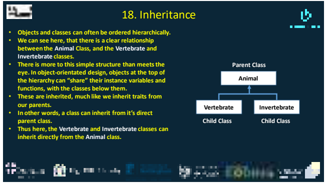


17. Encapsulation

- **In object-orientated design, a class should contain all the data and methods required for it to complete any and all tasks required.**
- **We simply call this *encapsulation*.**
- **The idea behind encapsulation, is that by designing classes in this way, they become modular, and can be plugged in elsewhere without us needing to re-write code.**
- **We can use encapsulation to share or hide information as required.**
- **For example, if we have a Customer class, there may be a class variable such as account number. This may need to be hidden from other classes – but can be exposed via a function if necessary, e.g.**

```
get_account_number(passcode): int
```

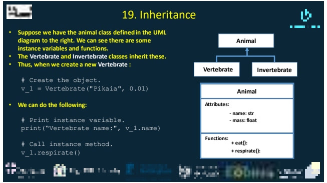
- **This would only return the account number, if the passcode is correct. Can you modify your Credit Card Class, so that it only permits payments if a passcode is provided? Try this in the Colab now.**



18. Inheritance

- Objects and classes can often be ordered hierarchically.
- We can see here, that there is a clear relationship between the Animal Class, and the Vertebrate and Invertebrate classes.
- There is more to this simple structure than meets the eye. In object-orientated design, objects at the top of the hierarchy can "share" their instance variables and functions, with the classes below them.
- These are inherited, much like we inherit traits from our parents.
- In other words, a class can inherit from it's direct parent class.
- Thus here, the Vertebrate and Invertebrate classes can inherit directly from the Animal class.

Parent Class
Animal
Vertebrate
Child Class
Invertebrate
Child Class



19. Inheritance

- Suppose we have the animal class defined in the UML diagram to the right. We can see there are some instance variables and functions.
- The Vertebrate and Invertebrate classes inherit these.
- Thus, when we create a new Vertebrate :

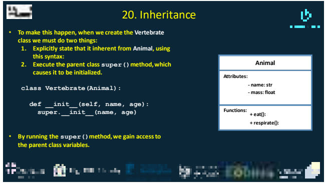
```
# Create the object.
v_1 = Vertebrate("Pikaia", 0.01)
```

- We can do the following:

```
# Print instance variable.
print("Vertebrate name:", v_1.name)

# Call instance method.
v_1.respirate()
```

Animal
Vertebrate
Invertebrate

Animal
Attributes:
- name: str
- mass: float
Functions:
+ eat():
+ respirate():



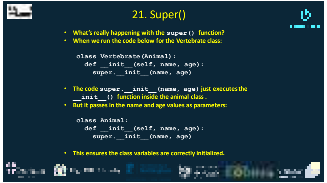
20. Inheritance

- To make this happen, when we create the Vertebrate class we must do two things:
  1. Explicitly state that it inherent from Animal, using this syntax:
  2. Execute the parent class `super()` method, which causes it to be initialized.

```
class Vertebrate(Animal):

  def __init__(self, name, age):
    super.__init__(name, age)
```

- By running the `super()` method, we gain access to the parent class variables.

Animal
Attributes:
- name: str
- mass: float
Functions:
+ eat():
+ respirate():



21. Super()

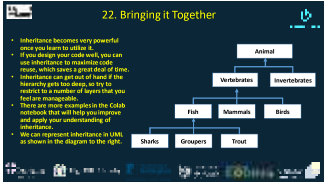
- What's really happening with the `super()` function?
- When we run the code below for the Vertebrate class:

```
class Vertebrate(Animal):
  def __init__(self, name, age):
    super.__init__(name, age)
```

- The code `super.__init__(name, age)` just executes the `__init__()` function inside the animal class .
- But it passes in the name and age values as parameters:

```
class Animal:
  def __init__(self, name, age):
    super.__init__(name, age)
```

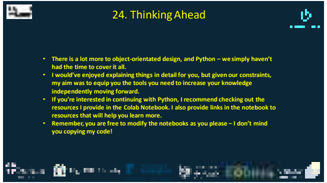
- This ensures the class variables are correctly initialized.



22. Bringing it Together

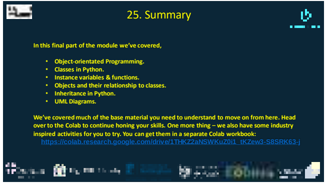
- Inheritance becomes very powerful once you learn to utilize it.
- If you design your code well, you can use inheritance to maximize code reuse, which saves a great deal of time.
- Inheritance can get out of hand if the hierarchy gets too deep, so try to restrict to a number of layers that you feel are manageable.
- There are more examples in the Colab notebook that will help you improve and apply your understanding of inheritance.
- We can represent inheritance in UML as shown in the diagram to the right.

Animal
Vertebrates
Invertebrates
Fish
Mammals
Birds
Sharks
Groupers
Trout



24. Thinking Ahead

- There is a lot more to object-orientated design, and Python – we simply haven't had the time to cover it all.
- I would've enjoyed explaining things in detail for you, but given our constraints, my aim was to equip you the tools you need to increase your knowledge independently moving forward.
- If you're interested in continuing with Python, I recommend checking out the resources I provide in the Colab Notebook. I also provide links in the notebook to resources that will help you learn more.
- Remember, you are free to modify the notebooks as you please – I don't mind you copying my code!



25. Summary

In this final part of the module we've covered,

- Object-orientated Programming.
- Classes in Python.
- Instance variables & functions.
- Objects and their relationship to classes.
- Inheritance in Python.
- UML Diagrams.

We've covered much of the base material you need to understand to move on from here. Head over to the Colab to continue honing your skills. One more thing – we also have some industry inspired activities for you to try. You can get them in a separate Colab workbook:

https://colab.research.google.com/drive/1THKZ2aNSWKuZ0i1_tKZew3-S8SRK63-j