Classes

Wrestling with Python
Using Classes

What we can do so far...

- Store data (using variables)
- Change data (using expressions)
- Make decisions (using conditions)
- Create loops (using while and for)
- Write methods (using def)
- Store data in lists (using er, lists)

Processing Cricket Scores

- Earlier we created a program to process cricket scores
- We read each score value in turn and used them to work out the highest and lowest scores, along with the total score and average

Storing the player name

- The user of our cricket score processing program would like it to store the name of each player as well as their score
- Then it could work out the name of the player with the highest score, as well as the score itself

Storing a single score

- We can score a single score in one value:

```
score = 10
```

- This will create a variable which can hold a single integer value
- The variable has the identifier **`score`**
- The variable holds the value **`10`**

Storing a single name

- We can score a single name in one value:

```
name = 'Fred'
```

- This will create a variable which can hold a single name value
- The variable has the identifier **`name`**
- The variable holds the value **`"Fred"`**

Storing lots of scores

- We could put the score values in a list

```
scoreList = []
scoreList.append(10)
```

- This puts the score of the first player on the end of the scores list

Storing lots of names

- We could use the same trick to store the names

```
nameList = []
nameList.append('Fred')
```

- This puts the name of the first player (Fred) on the end of the scores list

Working with two lists is tricky

- We could work with two lists like this
- However it would be hard to manage
  - If the names and score lists ever got out of step (for example when we try to sort them) then the program would display invalid results
- We really need a way of lumping the score and the name together

Python classes

- To solve this problem Python lets you create *classes*
  - A class describes the contents and behaviours of an object
- A class can contains *attributes* (data that is held in a class instance) and *methods* (behaviours the class provides)

A player class

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

- This is a `player` class
- It just contains a single method which is used to initialise it

Classes and Objects

- The class information tells the Python system how to make an instance of the class
  - This is called an *object*
- We have told the system how to create a `player` instance
- However, we have not actually created any `player` objects yet

Creating an object

```
p = player('Fred', 10)
print(p.name)
```

- This creates an instance of the `player` class and then prints out the name held in it
- Note that we pass in the name and the score when we create the instance
- This is passed into the `__init__` method

The __init__ method

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

- The `__init__` method is called the *constructor* for the class
- It is called automatically when we make a new instance

self in the __init__ method

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

- The __init__ method needs a reference to the object that is being created
- Python sets this reference and passes it into the method as the first parameter

Creating object attributes

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

- The `__init__` method creates `name` and `score` attributes which are held in the object
- The input values are copied into these attributes

Fetching data from an object

```
p = player('Jim', 99)
print(p.name)
p.name = 'Fred'
```

- You can get data out of attributes in an object by referring to them by name
  - They are effectively just like regular Python variables, they just live in an object

Fetching data from an object

```
p = player('Jim', 99)
print(p.name)
p.name = 'Fred'
```

- This code would print the name 'Jim' and then change the name of the player to 'Fred'
- Code can work with and change any attributes in an object

Special Object Methods

- Python will find the `__init__` method and use it when an object is to be created
- You can add your own methods to the objects as well
- We could add a `print_details` method which prints out the contents of the player class

Using attributes in class methods

```
def print_details(self):
  print(name, " scored ", score)
```

- You might think you can just use the attribute names directly in the `print_details` method
- However this will not work

This is not a happy ending

```
Traceback (most recent call last):
  File "C:\Users\Rob\SkyDrive\Wrestling with Python\Season 2\Week 04 Classes\cricket_class.py", line 6, in <module>
    p.print_details()
  File "C:\Users\Rob\SkyDrive\Wrestling with Python\Season 2\Week 04 Classes\cricket_class.py", line 3, in print_details
    print(name, " scored ", score)
NameError: global name 'name' is not defined
```

- When the `print_details` method is called it fails with the above error

This is not a happy ending

```
Traceback (most recent call last):
  File "C:\Users\Rob\SkyDrive\Wrestling with Python\Season 2\Week 04 Classes\cricket_class.py", line 6, in <module>
    p.print_details()
  File "C:\Users\Rob\SkyDrive\Wrestling with Python\Season 2\Week 04 Classes\cricket_class.py", line 3, in print_details
    print(name, " scored ", score)
NameError: global name 'name' is not defined
```

- We get the error because Python can't find the name attribute in the class

Attributes, methods and references

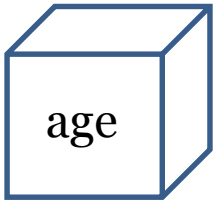
- To understand why we get the error, and how to fix it, we have to learn a bit about how Python finds objects
- Python uses *references* to locate and use the objects that a program is working with

References

- It is important that you understand this
- You need to know how references work to understand Python programs

Simple Variables

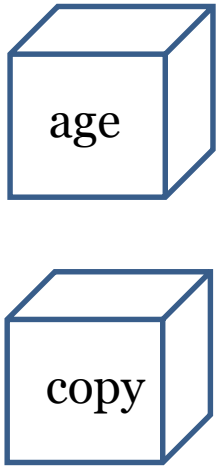
```
age = 99
```



- This creates a variable called age
  - You can think of it as a box with a name on it
- When you assign a value to the variable it puts something in the box
- When you use the variable the value is fetched out of the box

Assigning Simple Variables

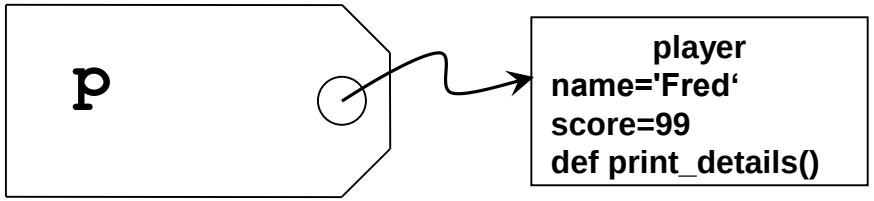
```
age = 99
copy = age
```



- If we assign one variable to another we get another box which contains the same data – both these boxes have 99 in them

Creating class instances

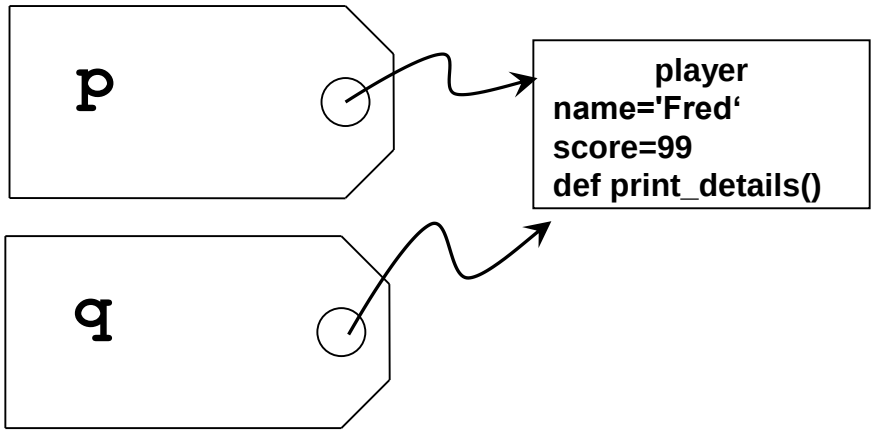
```
p = player('Fred',10)
```



- When we create a class instance what we actually create is an object and a reference
- You can think of a reference as a *tag*

Reference Assignment

```
p = player('Fred',10)
q = p
```



- If you assign a reference to another you get a tag which is tied to the same object
- A copy is **not** made of the object

References and Confusion

- Most of the time you can treat simple variables and references as the same thing
- But you need to be aware of their differences as getting them muddled up can lead to confusing behaviour
  - Changes to one thing might cause changes to something else – because they may be tags tied to the same object in memory

Back to print_details

```
def print_details(self):
  print(name, " scored ", score)
```

- The reason this doesn't work is that Python has no way of finding the name and score attributes of a particular object unless we tell it the object we want it to use

Back to print_details

```
def print_details(self):
  print(name, " scored ", score)
```

- We saw the `self` parameter when we wrote the `__init__` method
- We use it again in the print_details method

Using the self reference

```
def print_details(self):
  print(self.name, " scored ", self.score)
```

- We can use self in our methods to get hold of attributes
- You can think of it as a "reference to myself"
- It lets the method know which particular player the method is running within

Understanding self

- If this seems confusing (and it is) consider it from Python's point of view
- We want `print_details` to print out the name and score of the player object it is running inside
- To do this the method needs to know which object this is
- The `self` reference provides this information

Self and other parameters

```
def print_details(self, name_required):
    if name_required :
        print(self.name, " scored ", self.score)
    else:
        print(self.score)
```

- A method in an object can have multiple parameters
- This version of `print_details` is given a flag to control whether the name is printed

Self and other parameters

```
p.print_details(True)
```

- When we call the method we don't have to add the value of self to the call
- The Python system takes care of this automatically

Local variables in methods

```
def print_details(self, name_required):
    total = 0
    if name_required :
        print(self.name, " scored ", self.score)
    else:
        print(self.score)
```

- You can declare local variables inside methods
  - These can be used for things like working totals
- They are not part of the class and are destroyed when the method call ends

Creating a player object

PRACTICAL BREAK 1

Creating mutant objects

```
p = player('Fred',10)
p.age = 21
```

- Python will create new attributes in objects when you give them values
  - Just like it will create new variables the first time you use them
- The statement above creates a "mutant" object based on player that has an extra age attribute

Mutant Objects are Bad

- This is very dangerous
- It might mean that your programs can't rely on all the objects of a particular type holding the same data
- You should make sure that you create all the attributes when you initialise the object and don't add any on a piecemeal basis later

Lists of objects

```
players = []
p = player('Fred',10)
players.append(p)
print(players[0].name)
```

- We can add object references to lists, so that we can store complex collections of data

Summary

- Classes bring together methods and attributes to hold collections of information
- A class can have an `__init__` method that is used to set initial values to the attributes
- Methods in classes are passed a 'self' reference which is used by the method to access attributes in the object
- Classes are managed by references, which are tags linked to objects in memory

Storing a number of player objects

PRACTICAL BREAK 2