THE UNIVERSITY OF SYDNEY

# INFO1110 / COMP9001 Week 9 Tutorial

**Classes and Unit test**

## Classes

We have seen different ways of structuring data, from concrete types such as int and float to more abstract types such as lists, tuples and dictionaries but we have the ability to build our own types through classes.

### What is a class

A class allows the programmer to create a `type` which can bundle both data and functionality. The `str` class contains an internal list of characters as well as functions to operate on its own data.

### `__init__(self, ...)` constructor and `self`

Part of constructing and initialising instances of classes requires implementing the __init__ method. This method is invoked when an object of the type is initialised. You can define properties to be passed to the object through the constructor arguments.

It is also important to note the use of the `self` variable used in the constructor. The self variable is a binding to the instance of the variable and when writing instance methods, we will need to specify and use the self variable.

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

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

To create an instance of a class:

```
p = Person('Jim Moriarty', 28)
#To use the name method
p.get_name()
#However, we can refer to the variable itself
p.name
```

## Instance properties, getters and setters

We can extend our class from the previous example by adding an occupation property in the constructor. When creating a class we need to consider how we should expose the data.

- If instance properties should be read after initialisation then you should add a getter method. Denoted with a `get_` prefix. Example: `get_`$ame$,

```
class Person:

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

  def get_name(self):
    return self.name

  def get_age(self):
    return self.age

  def get_occupation(self):
          return self.occupation

  def set_occupation(self, job):
          self.occupation = job
```

## Private attributes

Other programming languages have the notion of encapsulation (or how data is exposed). However python does not have such a mechanism. Simple case of "private attributes do not exist" but typically to demonstrate to other developers that someone shouldn't use or manipulate a variable is denoted by __ prefix. Example:

```
class Person:
  def __init__(self, name, age):
    self.age = age
    self.name = name
    self.__times_name_called = 0

  def get_name(self):
    self.__times_name_called += 1
    return self.name
```

# Question 1: Components and variables

Consider the following class definition:

```
class Book:
        def __init__(self, title, author, year, url):
                self.title = title
                self.author = author
                self.year = year
                self.url = url
```

- What do each of the components/keywords of the this class mean?
- How can we access the data in the class?
- What type is `self`?
- What does it mean to create a new `Book` in our code?

# Question 2: Attributes

- Have we already used classes prior to this tutorial? What type and what data do you think they have stored?
- What are the advantages to creating classes instead of using what is already there?
- With the given type, describe what kind of attributes we could give it
  - Polygon
  - Song (music)
  - Album
  - Employee
  - Table (furniture)
  - Company

For each example, consider the following questions:

- What data, incudling data type, should be stored in each object?
- How should the data be accessed?
- Should someone be able to read/write the data?

# Question 3: Player and Highscore

You are given a class called Player. Each player has a highscore they have achieved from the game. After all player highscores are entered.

- You will need to define the properties associated with a player
- You will need to define a **class method** that will take a list of players and return which player has the highest score.

Player Class Scaffold:

```
class Player:

  def __init__(self, name):
    pass

  def highscore(self):
    pass

  def highest_score(players):
    pass
```

Usage:

```
player_list = [] #list of players
hp = Player.highest_score(plist)
print("Highest Score: {} with {} points".format(hp.name, hp.score))
```

# Question 4: Box and Items

You are given two classes, one is a box which can contain N items and the other is an item which has a name and weight.

The box instances needs to be able to report how much they weigh via the weight() method.

Scaffold:

```
class Box:
  def __init__(self, n):
    pass

  def add_item(self, item):
    pass

  def weight(self):
    pass



  #Add method about contents of box

class Item:
  def __init__(self, name, weight):
    pass

  def get_weight(self):
    pass

  def get_name(self):
    pass
```

Usage:

```
b = Box(5) #Number of items it can hold
b.add_item(Item('Gameboy', 2))
b.add_item(Item('Book', 10))
b.add_item(Item('DVD', 1))
b.add_item(Item('Bluray', 1))
b.add_item(Item('Game1', 1))
b.add_item(Item('Game2', 1)) #Cannot fit

print("The box weighs: {}kg".format(b.weight))
```

# Question 5: Pet

Create a class for the Pet object. Your class should contain instance variables (fields), appropriate get/set methods and at least one constructor. Do not worry about the implementation of the rest of the class.

A `Pet` object contains the following:

- a name
- an array of nicknames
- age
- species (animal type)
- whether or not the pet is house trained

# Question 6: Methods for pet

Implement the followign methods for the Pet class created in the previous question.

- Implement an `add_nickname` method that adds a new nickname to the pet (but only if the pet doesn't already have that nickname)
- Implement an `has_nickname` method that checks if the pet has a given nickname.

## Using the `import` and `from` keyword

Typically a good practice is to write classes in separate files and `import` them when needed. This allows for better organisation for project and gives you flexibility of use.

The from keyword allows you to specify the file and then import specific variables, functions and types that can be used in that file.

We can import another file into our current file like so:

Format:

```
import <file>
```

Example:

```
import person
```

Or if we want a specific variable or type from a file.

```
from <file> import <component1>, [<component2>], ...
```

Example:

```
from person import Person
```

The later examples allows the usage of very specific components of module to be used within your own code. They will also be namespaced to your module (you do not need to use person.Person to use the Person class).

## Question 7: Testing your pet class

We will be introducing a unit testing framework called pyunit. Last week you wrote test cases for your simple calc program. Now we are going to transform those unit test cases to py unit test cases. Since we have just covered classes we will be creating a class that will inherit from. Like so:

Format:

```
import pet
import unittest
class PetTest(unittest.TestCase):
```

After creating this class we will create methods that that start with `test_` (these methods will be ran by test runner, other methods are ignored).

```
import pet
import unittest
class PetTest(unittest.TestCase):

  def test_has_nickname(self):
    # Your code here

  def test_add_nickname(self):
    # Your code here

  ...
```

To execute a pyunit test case, we will run it similarly to last week's pdb.

```
python -m unittest <your test file>
```

```
python -m unittest pettest.py
```

## The `@property` decorator (Extension Reading and Discussion)

There is typically a case where want to provide a read-only variable or in this case property. We use the `@property` decorator to be functionally similar to a variable but refer to functions which operate on a private variable.

In this example we will rewrite the person class to use only properties and private variables:

```
class Person:
  def __init__(self, name, age, eyes):
    self._name = name
    self._age = age
    if eyes <= self.max_eyes:
      self._eyes = eyes
    else:
      self._eyes = 2

  @property
  def name(self):
    return self._name

  @name.setter
  def name(self, value):
    self._name = value

  @property
  def age(self):

  @age.setter
  def age(self, v):
    self._age = v

  @property
  def max_eyes(self):
    return 2
```

Where appropriate, change your classes from previous questions to utilise the `@property` decorator.