COMP1531

3.1 - More Objects in python

Objects in python

- Contain *attributes* and *methods*
- Attributes are values inside objects
- Methods are functions inside objects
- Methods can read or modify attributes of the object

A simple example

```python
from datetime import date

today = date(2019, 9, 26)

# 'date' is its own type
print(type(today))

# Attributes of 'today'
print(today.year)
print(today.month)
print(today.day)

# Methods of 'today'
print(today.weekday())
print(today.ctime())
```

Creating objects

- *Classes* are blueprints for objects

```python
class Student:
    def __init__(self, zid, name):
        self.zid = zid
        self.name = name
        self.year = 1

    def advance_year(self):
        self.year += 1

    def email_address(self):
        return self.zid + "@unsw.edu.au"

rob = Student("z3254687", "Robert Leonard Clifton-Everest")
hayden = Student("z3418003", "Hayden Smith")
```

Details

- Methods can be *invoked* in different ways
  - rob.advance_year()
  - Student.advance_year(rob)
- The 'self' argument is implicitly assigned the object on which the method is being invoked
- The '__init__()' method is implicitly called when an object is *constructed* from the class

Aside: variations

- Python lets you do the same thing in **lots** of different ways
- We're teaching the simplest and least error-prone ways of doing things
- We'll come back to this later on.

Namespacing

- Each class has its own *namespace*.
- Different classes can have methods and attributes with the same name.

```python
class Course:
    def __init__(self, code, name):
        self.code = code
        self.name = name

    def email_address(self):
        return self.code + "@cse.unsw.edu.au"

comp1531 = Course("cs1531", "Software Engineering Fundamentals")
```

Duck typing

- Giving different classes attributes/methods with the same name can be useful
- "If it walks like a duck and it quacks like a duck, then it must be a duck"
- This function works for both Student and Course

```python
def contact_info(authority):
    heading = f"Contact info for {authority.name}"
    body = f"You can reach {authority.name} via {authority.email_address()}"
    return heading + "\n\n" + body
```

Iterators

- In Python, iterators are objects *containing* a countable number of elements
- For example, we can get an iterator for a list:

```python
animals = ["dog", "cat", "chicken", "sheep"]

animal_iterator = iter(animals)
```

Iterators

- Any object with the methods __iter__() and __next__() is an iterator
- Duck typing ^^^
- Simple example (squares)

```python
class Squares:
    def __init__(self):
        self.i = 0

    def __iter__(self):
        return self

    def __next__(self):
        self.i += 1
        return self.i*self.i
```

For loops

- Python for loops use iterators behind the scenes
- This is valid code:

```python
squares = Squares()

for i in squares: # Loops forever
    print(i)
```

Iterator vs Iterable

- Intuitively:
  - An iterator stores the state of the iteration (i.e. where it's up to).
  - Something is iterable if it can be iterated over.
- Concretely:
  - An iterator has __iter__() and __next()__ methods.
  - Iterables have __iter__() methods
- For example, lists are iterable, but they are not iterators
- For loops only need to be given something *iterable*