Summary

OOP Principles

Classes and Objects

A class is a blueprint for creating objects, and each object created from a class is an instance of that class.

The class specifies what attributes (instance variables) and methods each instance has. When we call an instance's method, it acts on that instance.

Encapsulation

Rather than allow people to change the variables directly, methods are provided to call to change variables. These methods are written by the developers who wrote the class and fully understand it, so know what needs to be changed.

- This allows people to use the code without needing to know the details of how it works (as a black box).
- It also allows the code to be improved without disruption — as long as the code looks the same from the outside, places that the classes have been used won't need to be rewritten.

Inheritance

Classes that are similar to existing classes but need small changes can inherit from those classes. This makes all the already-written methods available, and enables new code to be written to provide extra functionality or override existing methods.

- Less code needs to be written, reducing effort and complexity.
    - So we have fewer bugs.
- Bugs in the original classes only need to be fixed once.

Composition

We can use objects as attributes of other objects, which allows us to model complex objects — e.g. a class LightSystem which has a Light object and a Switch object among its attributes.

This gives the same advantages as Inheritance:

- Less code needs to be written, reducing effort and complexity.
    - So we have fewer bugs.
- Bugs in the original classes only need to be fixed once.

Advantages of OOP

- Allows parallel code development for groups of developers
- Helps write cleaner code with fewer errors
    - Simpler code means less likely to have errors
    - Code is easier to fix when there are errors
    - Fixes are invisible to developers using the classes
- Helps to reuse code
- Allows use of related code organisation and design techniques

By modularising our code, we:

- Allow parallel development within a team of software engineers.
- Simplify the code - rather than one large complex block, we have numerous simpler classes. We say the code is less tightly bound. Because each class is relatively simple, it is more likely to be bug free.
- If a bug exists, we fix it in the responsible class. This class is imported into many programs, so the fix propagates to where the code is used. If all of the code for the button is in one class, and there is a bug, we fix it once and it propagates to every instance of the button we create.
- We can also replace code with better versions; if we develop a better button class, as long as the class name and method headers are the same, we can replace the class and not break any code that uses the class.

    - If we implemented the button from scratch every time, we would have had to replace and test the code in numerous places.

OOP in Python

Python doesn't enforce OOP or its principles. Instead, the freedom to use the language any way you like is emphasised. Encapsulation is implemented based on trust.

Classes in Python

Classes are written like this:

```
class Person:
    def __init__(self, name, job, pay):
        self._name = name
        self._job = job
        self._pay = pay
```

- The `__init__` method is the constructor — it's called when an instance of the class is created.
- We use underscores in the attribute names to indicate that they are private and should not be accessed directly.
- All class methods must include `self` in their definition, though we don't have to pass it when we call the methods.
    - `self` is a pointer to this instance.

They're used like this:

```
cathal = Person('Cathal', 'dev', 55000)
laura = Person('Laura', 'manager', 70000)
print(cathal.name, cathal.pay)
print(laura.name, laura.pay)
```

Methods are written and used just like functions:

```
def givePayRaise(self, percentage):
    if percentage < 0 or percentage > 100:
        print("%i is an illegal percentage" % (percentage))
    else:
        self._pay += self._pay // 100 * percentage

cathal.givePayRaise(10)
```

Packaging Classes

Imports

Other code can be used in your code by importing it:

```
from person import Person
```

Now the code can be used as if written in the importing code:

```
cathal = Person('Cathal', 'dev', 55000)
```

Test Block

Use `if __name__ = '__main__'` to check if the module is imported or run run directly, only call the test block if it is run direcly:

```
def main():
    #test block

if __name__ = '__main__':
    main()
```

- `__name__` will be set to the module's name if it's imported.

Commenting

Comments should be written using docstrings. In this module we're following the Google Python Style Guide.

`__str__` Method

The `__str__` method of an object defines how it's converted to a string when (e.g.) passed to `print()`. It's written like this:

```
class Person:
    def __str__(self):
        description = ("%s %s %d" % (self._name, self._job,
self._pay))
        return description
```

- This must be called `__str__` and must return a string.

Properties (and Property Decorators)

Properties allow attributes to be accessed through normal dot notation, but use getter and setter methods to control the access. They can be defined in two ways.

Property Notation

```
class Person:

    def getPay(self):
        return self._pay

    def setPay(self, pay):
        if pay < 0 or pay > 100000:
            print("%i is an invalid pay value - no value
set" %(pay))
        else:
            self._pay = pay

    pay = property(getPay, setPay)  #Note the order of the
methods in the list.
```

Property Decorator Notation

```
class Person:

    @property
    def pay(self):
        return self._pay

    @pay.setter
    def pay(self, pay):
        if pay < 0 or pay > 100000:
            print("%i is an invalid pay value - no value
set" %(pay))
        else:
            self._pay = pay
```

- In both cases, we can now write code like `cathal.pay = 20000`, and our setter functions will be used.

Advantages

- Familiar notation (dot notation) rather than explicit getter and setter methods — code is less awkward to use.
- Third-party code remains loosely bound to our code.
- If direct access was originally implemented, code can be changed to use getters and setter without requiring changes in the third-party code.

Inheritance in Python

Inheritance is written in Python like this:

- Import the necessary class(es):

```
from employee import Employee
```

- Inherit from the desired class:

```
class Engineer(Employee):
```

- Use the base class constructor:

```
def __init__(self, name, ssn, salary, skill):
    Employee.__init__(self, name, ssn, salary)
    self._skill = skill
```

The Search Heirarchy

When you inherit, you provide a search path for methods and variables.

- The sub-class is searched first.

- If it isn't found there, the super-class is search next.
- This is repeated up the inheritance heirarchy.

Basically, the most specific version is used. We use this when we override methods to provide specialisation.

Overriding Methods

By writing a method in a sub-class that has the same name as a method in the super-class, we can override it.

Composition in Python

We can include objects as attributes of other objects, just like any other inbuilt data type.

Object Persistence in Python

You can persist objects in Python by serialising them and writing them to disk — this is called shelving.

Pickles allow you to convert any object or collection of objects to a string of bytes and write them to a file.

Shelving stores pickled objects as key/value pairs, with all the usual dictionary methods.

Pickling is done automatically when we shelve objects.

Here's how to do it:

```
import shelve
from employee import Employee

john = Employee('SN12345', 'John Doe', 32000)

db = shelve.open('employeedb')
db[john.ssn] = john

cathal = db['SN48689']
cathal.givePayRaise(10)
db[cathal.ssn] = cathal

db.close()
```

- We have to open and close it like a file. By default you can read and write.
- We have to make sure the keys for the dictionary are unique, so here we use the employee's ssn.
- The shelf is synchronised when closed.

OOP Loose Ends

Polymorphism

- Operators (and any methods) can have different meanings depending on the type of the arguments.
    - For example, `i * 5` is different depending on whether `i` is a string or a number.

Duck Typing

Duck Typing is the process of establishing whether or not some object supports a particular behaviour or state.

Operator Overloading

In Python, we can give meaning to the operations built into Python (e.g. `<`, `>`, `==`, `!=`, `+`, `-`) when applied to our objects.

- This can allow our objects to behave more like built-in types,
- But it can be confusing if it's not clear what these operators should do.

We can provide an implementation for the `==` operator as follows:

```
class Student(object):
    def __init__(self, student_id, name):
        self._id = student_id
        self._name = name

    def __eq__(self, other):
        return self._id == other._id
```

With these methods, the method of the object on the left is called, passing the other as an argument.

We can provide similar definitions for `__add__`, `__lt__` (less than), `__ne__` (not equal), and others.

Exceptions

Exceptions are for when we run into runtime errors. These typically happen because of inputs, because we don't have control over what's passed into our program, or because of system errors.

We want to either return the program to a valid state or exit gracefully.

We can catch them with `try/except`, and raise them with `raise`.

```
try:
    if i < 10:
        raise ValueError
except ValueError:
    print('ValueError raised')
except Exception:
    print('Some other error raised')
else:
    print('No exceptions raised')
finally:
    print('This line prints whether exceptions were raised
or not')
```

There's a full list of exceptions in the Python Documentation.

GUIs

GUIs with tkinter

Here's a basic example:

```
from tkinter import *

root = Tk()         # Creates a toplevel container
lab = Label(root, text="Hello CS2513")      # Adds a label
widget
lab.pack()          # Arranges or lays out widgets

root.mainloop()     # Runs the interface and handles events
```

- `from tkinter import *` — this imports the needed tkinter classes (used to be `Tkinter` in Python 2)

- You have to have a top-level container (root in this example).
- Anything you want on the screen has to be passed to a layout manager — in this case `.pack()` is used to put the label on the screen. There's also `.grid()` and `.place()`.
- Once the main loop has been started, control is only given back to our code during callbacks we've written.