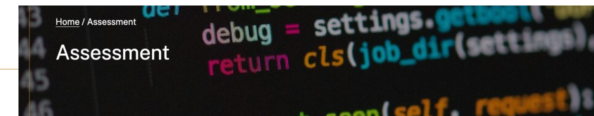
Announcements

- Friday is a public holiday and ALL LABS ON FRIDAY 29th MARCH HAVE BEEN MOVED TO A MAKE-UP TIME
  - Please remember to check your MyTimetable schedule and attend your make-up lab.
  - If you have problems with your allocated time, please use MyTimetable to move to a different lab. Please don't email the course address - we will just ask you to use myTimetable
- Homework 4 is due at the end of this week
- Homework 5 is released in the first week back next term (week 7).
- The assignment will be released in week 8 and is due at the end of week 10
- The Drop-In session this week will be held in N113 CSIT Building on **Thursday** 1-2pm



On this page

- Assessment scheme
- Submissions
- Variation for COMP6730 (master) students
- Final mark
- Marks appeal and correction
- Late submissions and extensions
- Deferred examination
- Special consideration
- Education Access Plan
- Cheating

Assessment scheme

Course assessment will be based on the following individual components (the link will appear at the latest on the release date):

Component/Link	Weight	Release date	Due date/Exam date
Homework 1	3%	26/02/2024 (Wk 2)	**03/03/2024, 23:55PM**
Homework 2	3%	04/03/2024 (Wk 3)	**10/03/2024, 23:55PM**
Homework 3	3%	18/03/2024 (Wk 5)	**24/03/2024, 23:55PM**
Homework 4	3%	25/03/2024 (Wk 6)	**14/04/2024, 23:55PM**
Homework 5	3%	15/04/2024 (Wk 7)	**21/04/2024, 23:55PM**
Project Assignment	35%	22/04/2024 (Wk 8)	**10/05/2024, 23:55PM (Fri Wk 10)**
In-lab project assessment	Mandatory discussions with a tutor in weeks 11 & 12 following the due date. **If absent, your project mark will be zero.**		
Final Exam	50%	N/A	TBA

- All these assessment items are **individual**; no group work is permitted. You must **read and accept** the Academic Integrity Rule 2021. Breaching the Rule will be noted on your ANU records and may have further consequences.
- **Homework and project's deadlines are hard**. No submissions after deadline are allowed without prior approved extension.

Persistence (*Think Python*, Ch14)

- When your program is executed, it has no memory of any previous time it may have been run. And nothing in memory will survive after the program exits.
- **Persistence** is the concept of retaining this information or memory *between* program execution instances
- This is commonly done by storing input and output files on disk
- Also, in databases (which are the subject of semester-long courses by themselves)
- And with python, can use `pickle` to create dumps of program memory that can be reread at another time
- But, importantly, reading files into your program **provides access to data**

Pickle files

- Sometimes it is desirable to store the state of a variable or an object to re-load in a later program run
  - Python does this with Pickle (or with Shelve)
- Pickle creates a string representation of an object, which can stored in a file or database – and later turned back into the original object:

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

- `pickle.dump()` strings can be re-loaded with `pickle.loads()`:

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> t2
[1, 2, 3]
```

Pickle example

- Writing data objects to a file and reloading later:

```
import pickle
import os

signup_names = list()

if os.path.exists('names.txt'):
    names_in = open('names.txt', 'rb')
    signup_names = pickle.load(names_in)
    names_in.close()

.
.
.
```

Modules

COMP1730/6730

Chapter 11 : Lubanovic, *Introducing Python*, 2nd Edition (2019)

But only section: *Modules and the import statement*

Modules (*Introducing Python* Ch 11)

- Every python file (`*.py`) can be imported as a module
- To get modules, use the `import` statement
  - This includes python programs you have written yourself
- Place your module python files in the current working directory

- Why do this?
  - Code reuse
  - Namespace partitioning
  - Avoidance of cut-and-paste code proliferation

Simple example

- Say that this is a function frequently appears in all your programs:

Program: counter.py:

```
def count_letters(word):
    '''Counts the letters in a word and returns the tally as a dictionary
    of counts keyed by each letter present'''
    letter_counts = dict()
    for letter in word:
        try:
            letter_counts[letter] += 1
        except KeyError:
            letter_counts[letter] = 1 # initialize dict key

    return letter_counts


counts = count_letters('example')

for letter in counts:
    print('Letter: ' + letter + "  Count: " + str(counts[letter]))
```

`import` as a module

- You could always cut-and-paste this function into your programs (please don't)
- `import` this function as a module

```
import counter

example_counts = counter.count_letters('example')

for letter in example_counts:
    print('Letter: ' + letter + "  Count: " + str(counts[letter]))
```

- Output:

```
In [26]: runfile('/Users/dan/untitled3.py', wdir='/Users/dan')
Letter: e  Count: 2
Letter: x  Count: 1
Letter: a  Count: 1
Letter: m  Count: 1
Letter: p  Count: 1
Letter: l  Count: 1
Letter: e  Count: 2
Letter: x  Count: 1
Letter: a  Count: 1
Letter: m  Count: 1
Letter: p  Count: 1
Letter: l  Count: 1
```

Note: Importing a *.py file causes it to be run on import. So, in this example. the counting code runs twice…

Tidier example

- Make a *.py of your function definition:

Program: counter.py:

```
def count_letters(word):
    '''Counts the letters in a word and returns the tally as a dictionary
    of counts keyed by each letter present'''
    letter_counts = dict()
    for letter in word:
        try:
            letter_counts[letter] += 1
        except KeyError:
            letter_counts[letter] = 1 # initialize dict key

    return letter_counts
```

`import` modules

- `import` file of counter function definitions:

```
import counter

example_counts = counter.count_letters('example')

for letter in example_counts:
    print('Letter: ' + letter + "  Count: " + str(counts[letter]))
```

- Output:

```
In [28]: runfile('/Users/dan/untitled3.py', wdir='/Users/dan')
Reloaded modules: counter
Letter: e  Count: 2
Letter: x  Count: 1
Letter: a  Count: 1
Letter: m  Count: 1
Letter: p  Count: 1
Letter: l  Count: 1

In [29]:
```

Much better!

`from` keyword

- You can use the from keyword just what you need from a module
- Execute any code in <module_name> script (may not be what you want):

```
from counter import count_letters

example_counts = count_letters('example')

for letter in example_counts:
    print('Letter: ' + letter + "  Count: " + str(counts[letter]))
```

as keyword short-hand

- You can use the as keyword for short-hand:

```
from counter import count_letters as cl

example_counts = cl('example')

for letter in example_counts:
    print('Letter: ' + letter + "  Count: " + str(counts[letter]))
```

Module search path

- Files (*.py) that you import as modules need to be in your **search path**
  - **The current directory from which your program is executed is searched first**
- If python can't find your file in your current directory or in the search path, it will produce and error
- Have a look at what is in your **search path:**

```
>>> import sys
>>> for place in sys.path:
...     print(place)
```

Lubanovic (2019) *Introducing Python*, Ch. 11

- Temporarily append to your search path with:

```
>>> import sys
>>> sys.path.append('d:\\modules\\')
```

Good things about modules (use them!)

- Separates namespace between a program and imported functions
- Very easy, low overhead way to re-use your code
  - The alternative is to cut-and-paste your functions from one program to the next (or worse, cut-and-pasting blocks of code)
    - This is how genes evolve in genomes – copy and diverge. Even evolution knows that this is how to introduce errors!
- Encourages code development as functions
- When code is implemented as functions, then it can also be easily tested
- We all have files of our favorite custom functions
  - It is as easy as placing this in the same directory where your program

Bad things about modules (annoyances, really)

- Modules can be filled with anything
  - Remember that the code in main is executed when a file is imported
- Calling a function from a module executes code, but it has no real concept of 'instance'
  - Functions code blocks can contain internal data and use this during a function call, but you aren't able to easily obtain this and/or modify it
- A module doesn't have the extra features of a Class
  - And a lot of the time this doesn't matter (much)

Exercises

- None

Reading

- Lubanovic, *Introducing Python* Ch 11 (section: *Modules and the import statement*)

Classes

COMP1730/6730

Chapter 10 : Lubanovic, *Introducing Python*, 2nd Edition (2019)
But only sections: *What are objects?, Simple Objects, In self defense, Attribute access, Data classes*

Python is object-oriented

- You may hear repeatedly that everything in python is an object
  - The handling of lot of these objects is hidden by python syntax
- What is an object?
  - An object is a data structure that contains:
    - a value (or multiple values – sometimes called attributes)
    - code (functions – called methods in classes)
  - An object is defined by a **class**
- Why should you care?
  - In python, a lot of the time, you don't really need to care
  - Objects and the classes that define them can be a powerful way to organize code
  - The code libraries that make python particularly useful are implemented as classes
  - **instantiate** classes as objects when you use them
- Code that is large and/or complex is best implemented with classes

Classes (*Introducing Python* Ch10)

- You have seen modules and `import`ing useful code from these
- At an introductory level, `class`es are just an extra formalism that makes things neater and more elegant.
- Object oriented coding is built on the `class`, but beyond the scope of this course
- But – knowing what a `class` is, and maybe how to write some simple ones for yourself, will make it <u>easier to understand what external software libraries</u> are all about.

Class vs module?

- When to use which?
  - Do you just need to import a function? Use modules
  - Do you need functions to operate in the context of some data? Use a class
- Instances: modules only allow a singleton. Classes can have multiple instances, that can all hold different attributes
- A class:
  - Can be instantiated with specific parameters
  - Is easily instantiated as many times as necessary
  - Contains methods that run on the particular instance of that class
  - Supports inheritance, polymorphism and all the object oriented stuff
- Modules become annoying if you:
  - Need to pass lots of data arguments to the function
  - Need to call many functions on the same parameter data

`class` definition syntax

- `class` keyword and a class name followed by a code block of a single line is the bare minimum (this does nothing, of course):

```
>>> class Cat():
...     pass
```

- Then you **instantiate** an object of this class with:

```
>>> a_cat = Cat()
>>> another_cat = Cat()
```

Lubanovic (2019) *Introducing Python*, Ch. 10

- These are distinct objects, from each instantiation. They have separate memory addresses

A class with one attribute - initialisation

- A class is often initialized at creation (instantiation). This is done by a special function named __init__ :

```
>> class TeenyClass():
...     def __init__(self, name):
...         self.name = name
...
>>> teeny = TeenyClass('itsy')
>>> teeny.name
'itsy'
```

Lubanovic (2019) *Introducing Python*, Ch. 10

- The __init__ function is optional, but may be defined to internally assign class attributes from parameter values (and many other things).

Anatomy of a class definition

- Say that your research interests required you to be able to associate quotations with their source:

```
>>> class Quote():
...     def __init__(self, person, words):
...         self.person = person
...         self.words = words
...     def who(self):
...         return self.person
...     def says(self):
...         return self.words + '.'
```

Initialisation method

Parameters

Class Attributes

Class Methods

self

- You could the use instances of this class to access this information in your programs:

```
>>> hunter = Quote('Elmer Fudd', "I'm hunting wabbits")
>>> print(hunter.who(), 'says:', hunter.says())
Elmer Fudd says: I'm hunting wabbits.
```

Lubanovic (2019) *Introducing Python*, Ch. 10

self

- What is self?
  - It isn't strictly a python keyword
  - This is a concept broader than python
- Some experimenting:

```
>>> class WhatIsSelf():
...     def __init__(self, name):
...         self.name = name
...         self.self_id = id(self)

>>> wis = WhatIsSelf('A Name')
>>> type(wis)
<class '__main__.WhatIsSelf'>
>>> id(wis)
140614751849392
>>> wis.self_id
140614751849392
```

self

- When an object is referring to itself, the `self` variable name is commonly used. Don't use `self` as a variable name elsewhere in your code.
- The `self` variable is always the first (silent) argument in all function calls (including __init__)
  - but it is automatic and implicit
  - You don't need to ever specify
  - It is included in method parameters and will therefore be a local variable to your function/method

```
>>> class Quote():
...     def __init__(self, person, words):
...         self.person = person
...         self.words = words
...     def who(self):
...         return self.person
...     def says(self):
...         return self.words + '.'
```

self

self

Attributes

- Attributes are the data or values that an object of a class holds
  - Can be parameters copied at initialisation
  - Can be default values
  - Can be derived values computed with class methods
- There is an open door to access attributes in python
  - All object attribute values can be accessed using the dot notation
  - Assumes programmers have discipline (or know what they are doing)

Parameters

```
>>> class Quote():
...     def __init__(self, person, words):
...         self.person = person
...         self.words = words
...     def who(self):
...         return self.person
...     def says(self):
...         return self.words + '.'
```

Class Attributes

The __init__ method

- __init__ is called when a class is instantiated
- Initialises class data and can perform checks
- Can call class methods at initialization

Parameters

```
>>> class Quote():
...     def __init__(self, person, words):
...         self.person = person
...         self.words = words
...     def who(self):
...         return self.person
...     def says(self):
...         return self.words + '.'
```

Class Attributes

Class methods

- Along with the data a class object contains, the benefit is also that class methods can be called to do tasks with this information

```
>>> class Quote():
...     def __init__(self, person, words):
...         self.person = person
...         self.words = words
...     def who(self):
...         return self.person
...     def says(self):
...         return self.words + '.'
```

Class Methods

An example class: PatientMutations

- The best reason to implement a class is to hide some complexity, to allow your client code to be simpler and easier to read and write
- This example is a class that imports data (personal genome information from a patient) and allows some filtering/retrieval of mutations
- patient_mutations class metadata: `patient_id`
- Genetic variation data table: `gene_name`, `coordinate`, `mutant`, `homozygous`, `essential_gene`, `damage_score`
- This data allows filtering of genetic variation data to find the disease-causing mutation
- With lots of patients with the same disease, we commonly look for common mutations in the same gene

Patient mutations data files:

Comma-separated values (CSV) file format:

`gene_name, chromosome, coord, ref_nucl, var_nucl, homozygous?, essential_category, damage_score`

mutations_193864.csv

```
TNFRSF4,1,1213738,G,A,True,2,0.74
PDE6B,3,46579986,C,T,True,2,0.94
TDGF1,4,660603,T,A,True,2,0.85
NDUFA13,19,19526194,T,C,True,2,1.00
PHEX,X,22247940,G,A,True,5,0.94
```

mutations_658192.csv

```
TNFRSF4,1,1213738,G,A,True,2,0.74
ACVR,12,157774144,A,G,True,3,0.67
HINT1,5,131165096,C,G,False,4,0.62
BCKDHB,6,80273147,A,G,False,7,0.56
SLC4A1,17,44253151,G,A,True,1,0.81
```

mutations_239872.csv

```
TNFRSF4,1,1213738,G,A,True,2,0.74
PDE6B,3,46579986,C,T,True,2,0.94
AMPD3,11,10500245,C,T,True,2,0.90
MOCOS,18,36195283,G,C,True,1,0.99
PHEX,X,22247940,G,A,True,5,0.94
```

mutations_283745.csv

```
TNFRSF4,1,1213738,G,A,True,2,0.74
ACVR,12,157774144,A,G,True,3,0.67
BCKDHB,6,80273147,A,G,False,7,0.56
SMARCA2,9,2060867,C,T,True,3,0.88
KRT2,12,52651601,T,G,False,8,0.23
```

mutations_947631.csv

```
TNFRSF4,1,1213738,G,A,True,2,0.74
TDGF1,4,660603,T,A,True,2,0.85
EGR2,10,62813491,C,A,True,2,0.98
MOCOS,18,36195283,G,C,True,1,0.99
NDUFA13,19,19526194,T,C,True,2,1.00
```

PatientMutations in Genomics.py

- This illustrates the features of a class all together:
  - The class definition
  - A Docstring
  - __init__ method
  - Class attributes
  - Class methods

```
import csv

class PatientMutations():
    ''' An example class that imports personal genome mutation data from a
    file and allows retrieval/filtering of this by three criteria '''

    def __init__(self, patient_id, mutations_filename):
        self.patient_id = patient_id  # metadata to link to patient
        self.input_file = mutations_filename
        self.mutations_data = [] # initialise mutation data as a list here
        self.read_mutations_file() # method call to read from input file

    def read_mutations_file(self):
        input_file = open(self.input_file, 'r')
        mutations_csv = csv.reader(input_file)

        for mutation in mutations_csv:
            mutation.append(self.patient_id)
            # populate internal mutations data from input file
            self.mutations_data.append(mutation)

        input_file.close()


    def candidate_disease_mutations(self,
                                    essential_cutoff=2,
                                    require_homozygous=True,
                                    damage_score_cutoff=0.7):
        disease_mutations = []

        for mutation in self.mutations_data:
            # copies of mutation information for readability below
            is_homozygous = bool(mutation[5])
            essential_score = int(mutation[6])
            damage_score = float(mutation[7])

            if (is_homozygous
                and essential_score <= essential_cutoff
                and damage_score >= damage_score_cutoff):
                disease_mutations.append(mutation)

        return disease_mutations
```

scan_mutations.py

```python
from Genomics import PatientMutations

patient1 = PatientMutations(193862, 'mutations_193864.csv')
patient2 = PatientMutations(283745, 'mutations_283745.csv')
patient3 = PatientMutations(947631, 'mutations_947631.csv')
patient4 = PatientMutations(239872, 'mutations_239872.csv')
patient5 = PatientMutations(658192, 'mutations_658192.csv')

patients = [patient1, patient2, patient3, patient4, patient5]
mutation_tally = dict()

for patient in patients:
    candidate_mutations = patient.candidate_disease_mutations()

    for candidate_mutation in candidate_mutations:
        gene_name = candidate_mutation[0]
        if gene_name in mutation_tally:
            mutation_tally[gene_name] += 1
        else:
            mutation_tally[gene_name] = 1

for mutant_gene in mutation_tally:
    print(mutant_gene + ': ' + str(mutation_
```

```
In [51]: runfile('/Users/dan/scan_mutations.py', wdir='/Users/dan')
TNFRSF4: 5
PDE6B: 2
TDGF1: 2
NDUFA13: 2
EGR2: 1
MOCOS: 2
AMPD3: 1
SLC4A1: 1
```

Classes written by other people

- This is really what libraries are (mainly) composed of. You have all used one already:
  - robot.py contains several classes
  - You imported the robot module:

```
import robot
```

  - Then some magic detected if the robot was plugged in (or the simulation is used):

```
In [1]: robot.init()
```

  - Initialising the robot (without a robot) instantiated the SimRobot class from robot.py. If you have the robot, the RPCRobot class is instantiated.
  - With either of these objects, you could use class methods to:
    - lift_gripper()
    - move_right()
    - etc

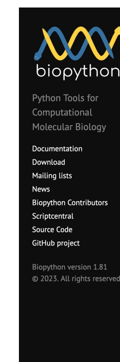
```
In [2]: robot.drive_right()
In [3]: robot.lift_up()
In [4]: robot.gripper_to_open()
In [5]: robot.lift_down()
```

A look at a Robot class:

- This is the RPCRobot class that can be found in robot.py from the labs
- Class `RPCRobot`
- Global attribute defaults
- `__init__` method
- Methods:
  - `lift_up`
  - `lift_down`
  - `drive_right`
  - Etc…

```python
class RPCRobot:
    '''Robot class interfacing with ev3 via RPYC.'''
    DEFAULT_DRIVE_RIGHT = 575
    DEFAULT_DRIVE_LEFT = 600
    DEFAULT_LIFT_UP = 220
    DEFAULT_LIFT_DOWN = 200

    def __init__(self, ip_address = "192.168.0.1"):
        import rpyc

        self.rpcconn = rpyc.classic.connect(ip_address)
        self.ev3 = self.rpcconn.modules.ev3dev.ev3
        self.battery = self.ev3.PowerSupply()
        self.drive = self.ev3.LargeMotor('outB')
        self.lift = self.ev3.LargeMotor('outD')
        self.gripper = self.ev3.MediumMotor('outC')
        self.sensor = self.ev3.ColorSensor()
        self.proxor = self.ev3.InfraredSensor()

    def print_state(self):
        print("drive at " + str(self.drive.position))
        print("lift at " + str(self.lift.position))
        print("gripper at " + str(self.gripper.position))
        print("sensor read: " + str(self.sensor.value()))
        print("proxor read: " + str(self.proxor.value()))
        print("battery: " + str(self.battery.measured_volts) + "V, "
              + str(self.battery.measured_amps) + "A")

    # moving up doesn't require braking
    def lift_up(self, distance=DEFAULT_LIFT_UP):
        print("lift at " + str(self.lift.position)
              + ", speed " + str(self.lift.speed))
        self.lift.run_to_rel_pos(position_sp = -distance, duty_cycle_sp = -25)
        time.sleep(0.5)
        while abs(self.lift.speed) > 0:
            print("lift at " + str(self.lift.position)
                  + ", speed " + str(self.lift.speed))
            time.sleep(0.25)
        print("(end) lift at " + str(self.lift.position)
              + ", speed " + str(self.lift.speed))

    # moving down requires braking, and even then has to be commanded ~10 short
    def lift_down(self, distance=DEFAULT_LIFT_DOWN):
        print("lift at " + str(self.lift.position)
              + ", speed " + str(self.lift.speed))
        self.lift.run_to_rel_pos(position_sp = distance,
                                 duty_cycle_sp = 25,
                                 stop_command='brake')
        time.sleep(0.5)
        while abs(self.lift.speed) > 0:
            print("lift at " + str(self.lift.position)
                  + ", speed " + str(self.lift.speed))
            time.sleep(0.25)
        print("(end) lift at " + str(self.lift.position)
              + ", speed " + str(self.lift.speed))

    def drive_right(self, distance = DEFAULT_DRIVE_RIGHT):
        print("drive at " + str(self.drive.position)
              + ", speed " + str(self.drive.speed))
        self.drive.run_to_rel_pos(position_sp = distance,
                                  duty_cycle_sp = 50,
                                  stop_command='hold')
        time.sleep(0.5)
```

Classes written by other people



Edit this page on GitHub

Biopython

See also our News feed and Twitter.

Introduction

Biopython is a set of freely available tools for biological computation written in Python by an international team of developers.

It is a distributed collaborative effort to develop Python libraries and applications which address the needs of current and future work in bioinformatics. The source code is made available under the Biopython License, which is extremely liberal and compatible with almost every license in the world.

We are a member project of the Open Bioinformatics Foundation (OBF), who take care of our domain name and hosting for our mailing list etc. The OBF used to host our development repository, issue tracker and website but these are now on GitHub.

This page will help you download and install Biopython, and start using the libraries and tools.

Get Started	Get help	Contribute
Download Biopython	Tutorial (PDF)	What's being worked on
Main README	Documentation on this wiki	Developing on Github
	Cookbook (working examples)	Google Summer of Code
	Discuss and ask questions	Report bugs

The latest release is Biopython 1.81, released on 12 February 2023.

Things you can ignore: underscore '__' names

- There is A LOT of syntax that is very specific to python classes.
  - You can write good classes with only a minimum of this
- If interested, see Lubanovic (2019) *Introducing Python* Ch 10: *Magic Methods*
- For example:

Table 10-3. Other, miscellaneous magic methods

Method	Description
`__str__(` *self* `)`	`str(` *self* `)`
`__repr__(` *self* `)`	`repr(` *self* `)`
`__len__(` *self* `)`	`len(` *self* `)`

- There are default methods for these, but you can implement your own too

Things you can ignore:

- Object-oriented coding is beyond the scope of this course
- Introducing classes is a starting point into object-oriented coding
- Further topics that WILL NOT be in the exam:
  - Magic methods
  - Inheritance
  - Polymorphism
  - Operator overloading
- But you are encouraged to read more about them, if you are interested!

Exercises

- Exercises 10.4, 10.5 & 10.6 , *Introducing Python* Ch. 10
  - If you have time

Reading

- Lubanovic (2019) *Introducing Python* Ch 10 (but only sub-sections: *What are objects?, Simple Objects, In self defense, Attribute access, Data classes)*