Announcements - In-lab project assessment

* In-lab project assessment along **this week** (i.e., week 11)
* You will be interviewed by a tutor during the lab. Opportunity to:
  - Defend/show understanding of your work
  - Receive preliminary feedback on your work
* If absent without approval of the conveners, your mark for the project will be zero
* Lab 10, which runs during Weeks 11 and 12, will be on practicing final exam exercises and programming problems

Announcements - Final exam format

Final exam worths 60% of your final mark

Exercises (**24%**):

* Improving code quality (6%)
* Testing (6%)
* Debugging (6%)
* Time complexity (6%)

Programming problems (**36%**):

* Problem 1 (9%)
* Problem 2 (9%)
* Problem 3 (9%)
* Problem 4 (9%)

See Lab 10 (weeks 11 and 12) for examples of practice exam exercises and programming problems

Australian National University

COMP1730/COMP6730
Programming for Scientists

Modules and command-line parsing

Lecture outline

* Python modules
* Command-line interface and parsing

Modules

Modules (Motivation)

- We have been using modules all semester (e.g., `robot`, `math`, `numpy`, etc.)
- A module is a collection of interrelated data and functions (also *classes*; see tomorrow's guest lecture)
- Functions in a module can be reused in many different programs (think, e.g., about `numpy` or `math` modules)
- If you have several functions that can be handy in many different programs, put them in a module
- Modules are specially relevant as a means to organize different functionality in **large software projects**
- In Python, writing modules is very easy: just collect the functions you want in a source file, and that becomes a module

Modules (example on making our own module)

Formulas for computing with interest rates:

- $A = A_0 \left(1 + \frac{p}{360*100}\right)^n$
- $A_0 = A \left(1 + \frac{p}{360*100}\right)^{-n}$
- $n = \frac{\log\left(\frac{A}{A_0}\right)}{\log\left(1 + \frac{p}{360*100}\right)}$
- $p = 360 * 100 \left(\left(\frac{A}{A_0}\right)^{\frac{1}{n}} - 1\right)$

A_0: initial amount; A: final amount;
p: annual interest rate (%); n: Number of days

Modules (example on making our own module)

```python
import math

def final_amount(A0, p, n):
    """docstring goes here"""
    return A0*(1.0 + p/(360.0*100.0))**n

def initial_amount(A, p, n):
    """docstring goes here"""
    return A*(1.0 + p/(360.0*100.0))**(-n)

def num_days(A0, A, p):
    """docstring goes here"""
    return math.log(A/A0)/math.log(1.0 + p/(360.0*100.0))

def annual_rate(A0, A, n):
    """docstring goes here"""
    return 360.0*100.0*((A/A0)**(1.0/n) - 1.0)
```

Modules (example on making our own module)

- Collect the 4 functions in a source file `interest.py`
- Now `interest.py` becomes a module named `interest`

Example of usage: how many years does it take to double an initial amount at 5% interest rate?

```
from interest import num_days
A0 = 100.0
p = 5.0
n = num_days(A0, 2*A0, p)
years = n/365.0
print(f"Your initial amount will double in {years:.2f} years")
```

Modules (names and namespaces)

* When the Python shell runs in “script mode”, the file it’s executing becomes the “main module”.
  – Its name becomes `'__main__'`
  – Its namespace is the global namespace
* The first time a module is imported, that module is loaded (executed); it may later be re-loaded
* Every loaded module creates a separate (permanent) namespace

Modules (the `import` statement)

* When executing `import` *`modname`*, the Python interpreter:
  - checks if *`modname`* is already loaded;
  - if not (or if reloading), it:
    - finds the module file (normally *`modname`*`.py`)
    - executes the file in a new namespace;
    - and stores the module object (roughly, namespace) in the system dictionary of loaded modules;
  - and then associates *modname* with the module object in the current namespace.

Modules (checking for module names)

* The global variable `__name__` in every module namespace stores the module name
* `sys.modules` is a dictionary of all loaded modules
* `dir(`*`module`*`)` returns a list of names defined in *`module`*'s namespace
* `dir()` lists the current (global) namespace

Modules (example)

```
>>> __name__
'__main__'
>>> import sys
>>> len(sys.modules)
...
>>> sys.modules['math'].__name__
'math'
>>> dir()
[ ..., 'sys' ]
>>> import math
>>> dir()
[ ..., 'sys', 'math' ]
```

Adding tests to modules (example)

- Modules can have an `if __name__ == 'main'` statement at the end containing e.g., tests or code demonstrating the module
- This block is NOT executed when the file is imported from another module but ONLY when the file is run as a script

```
... # Module's function definition statements

def test_all_functions():
   A = 2.2133983053266699; A0 = 2.0; p = 5; n = 730
   A_computed = final_amount(A0, p, n)
   A0_computed = initial_amount(A, p, n)
   n_computed = num_days(A0, A, p)
   p_computed = annual_rate(A0, A, n)
   tol=1E-12
   success = abs(A_computed,-A)<tol) and ... abs(n_computed-n)<tol
   assert success, "interest module tests failed!"

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

The command-line interface

The command-line interface

* A command-line (“terminal” or “shell”) is a text I/O interface to the computer’s operating system (OS), as apposed to GUIs, which are based on graphical elements (windows, buttons, etc.)
* The shell is an **interpreter** for a command-based programming language

(Image from wikipedia)

* The syntax (and to some extent the concepts) of command programming languages are quite different among OSs, but there also fairly common aspects among them
* Command-line interfaces are very common in scientific pipelines (e.g., when running scientific programs on supercomputers)

Passing arguments from the command-line

- A Python program can be run from the command-line, e.g.:

  `$ python my_program.py`

  where `python` is the Python interpreter
- We can pass arguments to the program from the command-line:

  `$ python my_program.py arg1 "arg two" 3.1416`
- `sys.argv` is a list of **strings** where `sys.argv[0]` is the name of the Python program and `sys.argv[1:]` are the arguments
- An alternative way to connect the program with the outside world

Example

- Program that evaluates the mathematical formula (1D motion):

$$y(t) = y_0 + v_0 t + \frac{1}{2} a t^2$$

- Input: y_0 (initial pos), v_0 (initial vel), a (acceleration), t (time)
- Output: $y(t)$ (position at time t)

```
import sys
y0 = float(sys.argv[1])
v0 = float(sys.argv[2])
a  = float(sys.argv[3])
t  = float(sys.argv[4])
yt = y0 + v0*t + 0.5*a*t*t
print(f"Position of object at time={t} (s) is {yt} (m)")
```

- Evaluate for $y_0 = 10$ m, $v_0 = 2$ m/s, $a = -9.81$ m/s^2, $t = 0.5$ s:

  `$ python position.py 10 2 9.81 0.5`

- Do you anticipate any potential issues with the program above?

Command-line arguments with options

- Many programs, especially on Unix-type systems, take a set of command-line arguments of the form `--option value`, such as, for example:

```
$ python position.py --y0 10 --v0 2 --a -9.81 --t 0.5
$ python position.py --t 1.0
```

- The second invocation relies on default values for the other parameters: we only provide those values that we want to change
- Such option-value pairs facilitate the user to understand and remember what the different options mean
- The `argparse` module (next slide) provides **very convenient way** to ”parse” command-line arguments with options (e.g., it handles all possible errors with meaningful error messages)

Parsing option-value pairs with `argparse` module

```
import argparse
parser = argparse.ArgumentParser()

# Define command-line arguments
parser.add_argument('--v0', '--initial_velocity', type=float,
                    default=0.0, help='initial velocity')
parser.add_argument('--y0', '--initial_position', type=float,
                    default=0.0, help='initial position')
parser.add_argument('--a', '--acceleration', type=float,
                    default=1.0, help='acceleration')
parser.add_argument('--t', '--time', type=float,
                    default=1.0, help='time')

# Read the command line and interpret the arguments
args = parser.parse_args()

# Extract values and evaluate formula
y0 = args.y0; v0 = args.v0; a = args.a; t = args.t
yt = y0 + v0*t + 0.5*a*t**2
```

Documentation:

https://docs.python.org/3/library/argparse.html