ECOR 1042
Data Management

Modules and Unit Testing

Recap Learning Outcomes Previous Lecture

- Be able to apply iterative, incremental development when working on a small-scale software project
- Be able to write code that processes data read from a text file
- Know the meaning of these words and phrases
  - Iterative development
  - Incremental development
  - Text file
  - `open` and `close` (built-in functions)
  - `split` and `strip` methods (type `str`)

Carleton University

References

- *Practical Programming*, 3rd ed.
  - Chapter 6, *A Modular Approach to Program Organization*

Lecture Objectives

- Review how to use Python's built-in modules
- Describe how to create our own modules
- Review unit-testing
- Describe how to create test programs for modules

Learning Outcomes

- Be able to:
  - create new modules
  - write programs that import modules and call the module's functions
  - write unit tests for modules

Modules

Why Modules?

- Why use modules?
  - This is useful to make the program organized and less cluttered
  - Allow re-use of code as a ”Library”
  - Divide the program into separate functionalities
- Once you run the project, all the files will basically be combined into one program

- ***Note:*** In ECOR1042 you will have all the files in the same folder, however for real applications, the program’s files will be in different folders

Review: Modules

- A *module* is a file containing a collection of (usually) closely-related functions and variables
- In ECOR 1041, you learned that Python's `math` module provides many math functions, plus variables for a few well-known values; e.g., **π**, *e*
- To use the functions and variables that are defined in a module we must first *import* it; e.g.,

```
>>> import math
```

Review: Modules

- You learned that the `math` module has a function named `sqrt`

```
>>> import math
>>> help(math.sqrt)
Help on built-in function sqrt in module
math:
sqrt(x, /)
  Return the square root of x.
```


Carleton University

Review: Modules (calling `sqrt`)

```
>>> sqrt(25)
builtins.NameError: name 'sqrt' is not
defined
>>> math.sqrt(25)
5.0
>>> math.sqrt(-25)
builtins.ValueError: math domain error
```

Use the dot operator to specify that we are calling the `sqrt` function from the `math` module

Review: Modules

- We can also use a `from import` statement to import specific functions and variables from a module; e.g.,

```
>>> from math import sqrt
```

- We can then call `sqrt` without using the dot operator

```
>>> sqrt(25)
5.0
```

Developing Modules

- Suppose we are developing a module that contains functions that process polynomials
- We will name the module `poly` (filename is poly.py)
- The module will have a function named `quad_roots` that calculates the roots of a quadratic equation (a second-order polynomial equation in a single variable *x*):

$$ax^2 + bx + c = 0$$

Carleton University

`quad_roots` Header & Docstring

```
def quad_roots(a: float, b: float, c: float) -> \
               tuple[float, float]:
    """Return the two roots of the quadratic equation
    ax^2 + bx +c = 0.

    >>> quad_roots(1, -6, 8)
    (4.0, 2.0)
    >>> quad_roots(2, 4, 2)
    (-1.0, -1.0)
    >>> quad_roots(2, 1, 2)
    (None, None)
    >>> quad_roots(0, 1, 2)
    (-math.inf, math.inf)
    """
```

Calculating Roots

- The roots of the quadratic equation are given by the *quadratic formula*:

$$x = \frac{b \pm \sqrt{b^2 - 4ac}}{2a}$$

`quad_roots` Body

```
if abs(a) < 0.0001:
    # Coefficient of x^2 term is 0.
    return (-math.inf, math.inf)

disc = b ** 2 - 4 * a * c
if disc < 0:
    # Equation has no real roots.
    return (None, None)

sqrt_disc = math.sqrt(disc) #calculate this just once!
return((-b + sqrt_disc) / (2 * a),
       (-b - sqrt_disc) / (2 * a))
```

Using `quad_roots`

- Here is a simple interactive app that imports `poly` and calls `quad_roots`

```
import poly

a = float(input('Enter coefficient a of ax^2 + bx + c'))
b = float(input('Enter coefficient b of ax^2 + bx + c'))
c = float(input('Enter coefficient c of ax^2 + bx + c'))
root1, root2 = poly.quad_roots(a, b, c)
print('The roots are:', root1, root2)
```

Use the dot operator to specify that we are calling the `quad_roots` function from the `poly` module

Selecting Which Code is Run on `import`

- Suppose we add a script to module `poly` that "exercises" (or tests) `quad_roots` (i.e. calls the function and prints the tuple it returns, four times)

```
def quad_roots(a: float, b: float, c: float) -> \
               tuple[float, float]:
    # Function body not shown

print(quad_roots(1, -6, 8))
print(quad_roots(2, 4, 2))
print(quad_roots(2, 1, 2))
print(quad_roots(0, 1, 2))
```

Selecting Which Code is Run on `import`

- Python can execute a module *directly* (e.g., by opening it in Wing 101 and clicking the Run button)
  - When `poly` is run directly, the module's script is executed
- Python executes modules *indirectly* as it imports them
  - In the simple interactive app, when `import poly` is executed, the `print` calls in `poly` are executed
- We probably do not want this!

Selecting Which Code is Run on `import`

- Sometimes we want a module to have code that should be executed only when the module is run directly and not when the module is imported
- Every module has a string variable called `__name__` that is created by Python, and which can be used to determine if a module is run directly or indirectly

Selecting Which Code is Run on `import`

- When a module is run directly, `__name__` is automatically assigned the string `'__main__'`
- When a module is imported, its `__name__` variable is automatically assigned a string containing the module's name; e.g., `'poly'`
- A module can determine if it is being run directly or has been imported by checking the value assigned to its `__name__` variable

Selecting Which Code is Run on `import`

- In `poly`, put the `print` calls in an `if __name__ == '__main__'` statement:

```
def quad_roots(a: float, b: float, c: float) -> \
               tuple[float, float]:
    # Function body not shown

if __name__ == '__main__':
    print(quad_roots(1, -6, 8))
    print(quad_roots(2, 4, 2))
    print(quad_roots(2, 1, 2))
    print(quad_roots(0, 1, 2))
```


Carleton University

Selecting Which Code is Run on `import`

- When `poly` is imported, the module is executed, `__name__ == '__main__'` evaluates to `False`, so the code in the `if` statement's block will not be executed

Modules and Unit Testing

Review: Unit Testing

- Unit testing tests a single unit in isolation
- In Python, a unit is typically a file containing one or more closely-related functions; e.g., a module
- Manual testing using the shell is ok for quick confidence tests, but is not the best way to thoroughly test a unit
- Automated testing is recommended

Review: Automating Testing

- Develop a test program for each unit
- The program has one or more test cases for each function in the unit
- The program performs regression testing (tests all functions, including ones that previously passed all tests)
- The program (not the person running the tests) compares actual outcomes with expected results, determines if tests passed/failed

Review: Automating Testing

- In ECOR 1041, you learned how to write simple test functions that use `assert` to compare expected outcomes to actual results
- In this lecture, we will review this concepts to create unit tests for modules

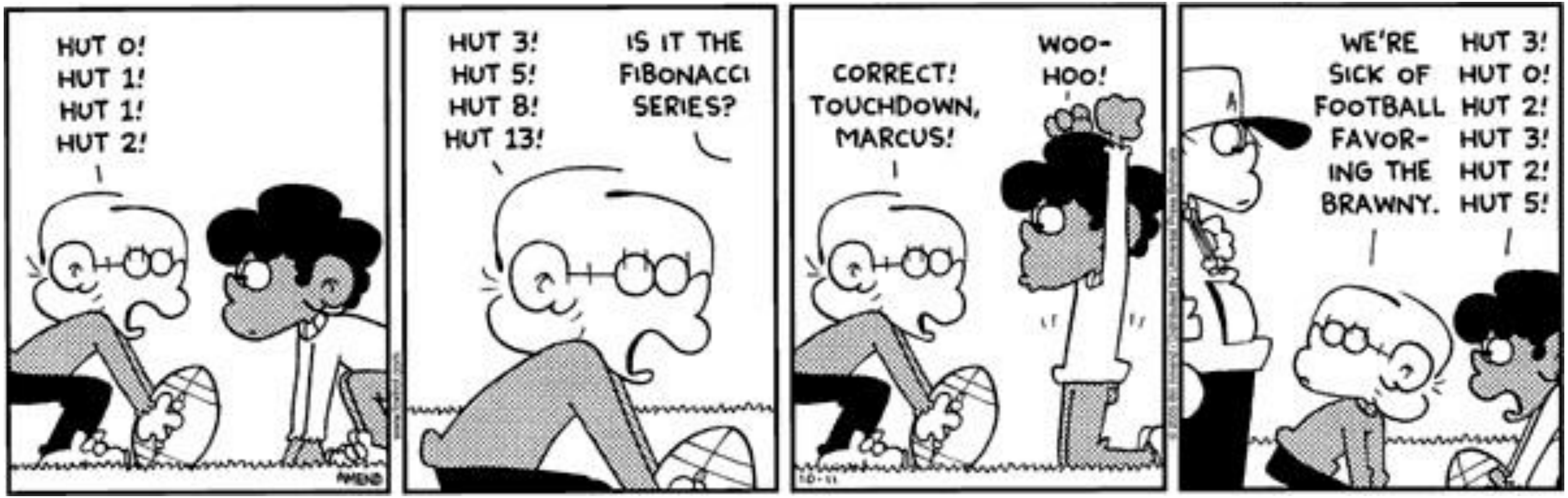
`assert` Statements

- `assert` is a Python keyword
- An `assert` statement has the form:

  `assert` *`expression`*`,` *`message`*

- If *`expression`* is `True`, execution continues quietly (no message is displayed)
- If *`expression`* is `False`, execution halts (by default) and an `AssertionError` is displayed
- If a *`message`* is included, the message is displayed

Case Study: Fibonacci Numbers



Carleton University

Case Study: Fibonacci Numbers

- Fibonacci numbers are the sequence of integers 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …
- Definition:
  - $F_1 = 1$
  - $F_2 = 1$
  - $F_n = F_{n-1} + F_{n-2}$
  - Sometimes, F_0 is included in the sequence ($F_0 = 0$)

`fibonacci`: Header and Docstring

```
def fibonacci(n: int) -> int:
    """Return the Fibonacci number F_n for positive values
    of n, where F_1 = 1, F_2 = 1, and F_n = F_n-1 + F_n-2,
    n > 2.

    >>> fibonacci(1)
    1
    >>> fibonacci(2)
    1
    >>> fibonacci(3)
    2
    >>> fibonacci(5)
    5
    """
```

Put the function definition in a module named `sequences` (file sequences.py)

`fibonacci`: Function Body with a Bug

```
def fibonacci(n: int) -> int:
    """Docstring omitted."""
    a = 1
    b = 1
    for i in range(1, n + 1):   # Bug!
                                # 2nd argument should be n.
        a, b = b, a + b
    return a
```

Testing `fibonacci`

- When designing test cases, we need to consider *test coverage*
  - Have we identified all boundary cases?
  - Do we have enough test cases to ensure that every line of code in the function has been executed at least once?

Test Cases

- `fibonacci(1)` is a boundary case
  - docstring states that *n* must be positive, so 1 is the smallest integer for which `fibonacci` must return a correct value
- We need a test that verifies that `fibonacci(1)` returns 1

Carleton University

Test Cases

- `fibonacci(2)` is a special case: it cannot be calculated from the formula $F_n = F_{n-1} + F_{n-2}$
- We need a test that verifies that `fibonacci(2)` returns 1
- Testing `fibonacci(1)` and `fibonacci(2)` is not sufficient: what if the function always returns 1?

Test Cases

- `fibonacci(3)` is a boundary case
- 3 is the smallest integer for which the result
  - is calculated using the formula $F_n = F_{n-1} + F_{n-2}$,
  - is a value other than 1
- We need a test that verifies that `fibonacci(3)` returns 2

Test Cases

- Notice that $F_n = n - 1$ for $n = 2, 3$ and 4
- Testing `fibonacci(1)`, `fibonacci(2)`, `fibonacci(3)` and `fibonacci(4)` is not sufficient
  - What if the function returns 1 when $n = 1$ and returns $n - 1$ for all $n > 1$?
- We need a test that verifies that `fibonacci(5)` returns 5
- Could instead test `fibonacci(6)`, `fibonacci(7)`, etc.

Outcomes vs Expected Results

- The docstring examples show the expected result of each test

```
"""
...
>>> fibonacci(1)
1
>>> fibonacci(2)
1
>>> fibonacci(3)
2
>>> fibonacci(5)
5
"""
```

Outcomes vs Expected Results

- The outcomes are obtained by calling `fibonacci` for each test case

```
>>> fibonacci(1)
1
>>> fibonacci(2)
2                   # Should be 1
>>> fibonacci(3)
3                   # Should be 2
>>> fibonacci(5)
8                   # Should be 5
```

`fibonacci` Test Program

- The test program is in a different module than the module that contains `fibonacci`
- Name the file `test_sequences.py`
- The test program uses `assert` and imports `fibonacci` from module `sequences`
- The test cases are implemented inside the function `test_fibonacci`, and if all tests pass, the function returns the number of tests that have been executed. If one test fails, the program will terminate.

`fibonacci` Test Program

```
from sequences import fibonacci

def test_fibonacci() -> int:
    tests = 0
    assert fibonacci(1) == 1, 'Wrong value for fibonacci(1)'
    tests += 1
    assert fibonacci(2) == 1, 'Wrong value for fibonacci(2)'
    tests += 1
    assert fibonacci(3) == 2, 'Wrong value for fibonacci(3)'
    tests += 1
    assert fibonacci(5) == 5, 'Wrong value for fibonacci(5)'
    tests += 1
    return tests

if __name__ == '__main__':
    print('All test pass. ',test_fibonacci(), 'tests executed')
```

`fibonacci` Test Program: Exercise

- Put sequences.py and test_sequences.py in the same folder
- Run test_sequences.py, review the output from the test program
- How can we use the output to determine the flaw in the code?
- Fix the bug in `fibonacci()`, run the test program, review the output

Extra Practice for Home

Extra-Practice Exercises

- In sequences.py, define function `factorial(n)`, which calculates n!
  - Define `test_factorial` in test_sequences.py and add a call to `test_factorial` to the test program
- Find the definition of the Perrin sequence, define function `perrin(n)` in sequences.py, add a test function to test_sequences.py

Recap Learning Outcomes

- Be able to:
  - create new modules
  - write programs that import modules and call the module's functions
  - write unit tests for modules

ECOR 1042
Data Management

Modules and Unit Testing