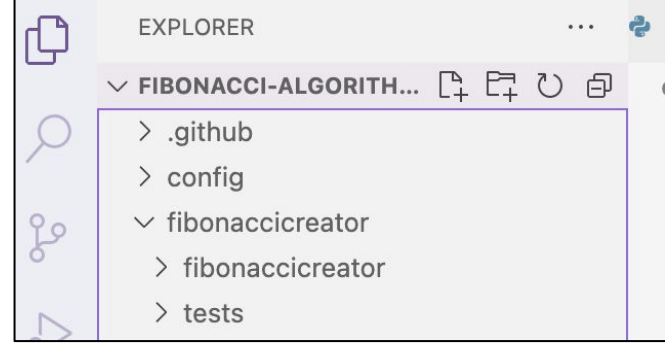


# Chapter 7, 8, 9

Modules, Testing, Exceptions

# File structure

- Larger programs cannot be written in one script
- **Modular programming:** breaking large coding tasks into smaller, manageable subtasks.
- In python, multiple files can be connected using import
- e.g. Current Lab: Fibonacci Engineering Effort



```
# this import and the use of its modules in this program should normally  
# work correctly on both the MacOS and Linux operating systems.
```

```
if os.name != "nt":
```

```
    from resource import getrusage, RUSAGE_SELF
```

```
from fibonaccicreator import fibonacci
```

# Terms

- Module: .py file
  - fibonacci.py
  - main.py
  - test\_fibonacci.py
- symbol: anything that is defined or assigned in the global scope of a module
  - functions
    - `fibonacci_recursivelist` (function in fibonacci.py)
  - classes
  - global variables
    - `FIBONACCI_FUNCTION_BASE` (global variable in main.py)
- namespace: refers to the name of the original module
- fully-qualified name: using a name that has the namespace and the symbol name separated with .
  - `fibonacci.fibonacci_recursivelist`
  - `main.FIBONACCI_FUNCTION_BASE`
- Package: directory
  - any directory that will be used like a package must also contain a file named `__init__.py`
- Library: a package with subpackages

# Imports

- `from MODULE import SYMBOL`
- `import MODULE`
- `import MODULE as ALIAS`
- `import PACKAGE`
- `from PACKAGE import MODULE`
- `from PACKAGE import MODULE.SYMBOL`
- `import PACKAGE.SUBPACKAGE`
- `etc`

# Package vs Module Example

- fibonaccicreator: **package**
- fibonacci: **Module** inside the package
- Package is a fancy term for directory
  - any directory that will be used like a package must also contain a file named `__init__.py`

*# this import and the use of its modules in this program should normally  
# work correctly on both the MacOS and Linux operating systems.*

```
if os.name != "nt":
```

```
    from resource import getrusage, RUSAGE_SELF
```

```
from fibonaccicreator import fibonacci
```

A diagram consisting of two purple arrows. One arrow originates from the word 'package' in the first bullet point and points to the underlined module name 'fibonaccicreator' in the import statement. The second arrow originates from the word 'Module' in the second bullet point and points to the underlined module name 'fibonacci' in the same import statement.

# Colab - importing

- <https://forms.gle/vD1yNqgcKjbPksiv8>

# Chapter 8 - Testing and Debugging

"**Testing** is the process of running a program to try and ascertain whether it works as intended. **Debugging** is the process of trying to fix a program that you already know does not work as intended."

- Testing often refers to testing whether the specifications are met
- i.e. did the function `is_prime` actually return true for primes and false for non-primes?
- **Black-box testing**

# Testing logic

If you want to test something based on specifications

- **examine** the inputs and outputs **as stated in the specs**
  - Docstrings usually contain the specifications
  - Make sure you write good docstrings
- **partition** the input/output space into all meaningful categories
- test simplest thing in each partitioned category
- if the test fails...then
- DEBUG!



# Example

- spec: inputs are x,y
- spec: output true if x is less than y...
- Partitions:
  - x less than y  $\rightarrow$  true
  - y less than x  $\rightarrow$  false
  - x == y  $\rightarrow$  false
- simple tests: x = 1, y = 0 | x = 0, y = 1 | x = 0, y = 0

boundary condition!



```
def is_smaller(x, y):  
    """Assumes x and y are ints  
        Returns True if x is less than y and False  
        otherwise."""
```

# Example

x positive, y positive,  $x < y$   
x positive, y positive,  $y < x$   
x negative, y negative,  $x < y$   
x negative, y negative,  $y < x$   
x negative, y positive  
x positive, y negative  
 $x = 0$ ,  $y = 0$   
 $x = 0$ ,  $y \neq 0$   
 $x \neq 0$ ,  $y = 0$

```
def is_smaller(x, y):  
    """Assumes x and y are ints  
        Returns True if x is less than y and False  
        otherwise."""
```

# Testing continued

- testing can also refer to testing the code - if you have knowledge and access to the implementation
- **glass-box testing**
  - ideal test is: "**path-complete**" meaning all possible branches and stack frame arrangements created while using the code will be tested
  - practical test is: as path-complete as possible

# Glass-box

- spec says non-neg  $\rightarrow$  0 and up
- notice the branches of the implementation
  - if else etc

```
def is_prime(x):  
    """Assumes x is a nonnegative int  
       Returns True if x is prime; False otherwise"""  
    if x <= 2:  
        return False  
    for i in range(2, x):  
        if x%i == 0:  
            return False  
    return True
```

# "Rules of thumb are usually worth following:

- Exercise both branches of all if statements.
- Make sure that each except clause (see Chapter 9) is executed. For each for loop, have test cases in which
  - The loop is not entered (e.g., if the loop is iterating over the elements of a list, make sure that it is tested on the empty list).
  - The body of the loop is executed exactly once.
  - The body of the loop is executed more than once.
- For each while loop
  - Look at the same kinds of cases as when dealing with for loops.
  - Include test cases corresponding to all possible ways of exiting the loop. For example, for a loop starting with
    - while len(L) > 0 and not L[i] == e
  - find cases where the loop exits because len(L) is greater than zero and cases where it exits because L[i] == e.
- For recursive functions, include test cases that cause the function to return with no recursive calls, exactly one recursive call, and more than one recursive call.

# Tools - Pytest - `poetry run task test`

```

└─ tests
  └─ __init__.py
  └─ test_fibonacci.py
24 def test_first_fibonacci_singleton_tuple():
25     """Ensure that the request for first Fibonacci number returns same number in a tuple."""

```

```


egrabner ~/Documents/Teaching/S2024-CMPSC101/fibonacci-algorithms-starter/fibonaccicreator %  poetry run task test

===== test session starts =====
platform darwin -- Python 3.12.2, pytest-7.4.4, pluggy-1.4.0
rootdir: /Users/egrabner/Documents/Teaching/S2024-CMPSC101/fibonacci-algorithms-starter/fibonaccicreator
collected 0 items / 1 error

===== ERRORS =====
_____ ERROR collecting tests/test_fibonacci.py _____
tests/test_fibonacci.py:13: in <module>
    from fibonaccicreator import fibonacci
fibonaccicreator/fibonacci.py:9: in <module>
    def fibonacci_recurisivelist(number: int) -> List[int]:
E   NameError: name 'List' is not defined

===== short test summary info =====
ERROR tests/test_fibonacci.py - NameError: name 'List' is not defined
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! stopping after 1 failures !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! Interrupted: 1 error during collection !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

===== 1 error in 1.36s =====

egrabner ~/Documents/Teaching/S2024-CMPSC101/fibonacci-algorithms-starter/fibonaccicreator % 

```

# After testing fails....

"**Debugging** starts when testing has demonstrated that the program behaves in undesirable ways. Debugging is the process of searching for an explanation of that behavior. "

- be able to explain the test, and what failed
- take note if something was too big, too small, wrong type, too long etc
- look at the code and read it slowly to come up with a theory that would explain the test results

# Build intuition

but also - you can peek into your code using breakpoints or print statements

- if you begin doing code surgery, use git to commit various versions (not push, just add, commit)
- write REAL commit messages that explain the progress or problem still existing
- these commit bookmarks will help you!



- *Look for the usual suspects.* Have you
  - Passed arguments to a function in the wrong order?
  - Misspelled a name, e.g., typed a lowercase letter when you should have typed an uppercase one?
  - Failed to reinitialize a variable?
  - Tested that two-floating point values are equal (==) instead of nearly equal (remember that floating-point arithmetic is not the same as the arithmetic you learned in school)?
  - Tested for value equality (e.g., compared two lists by writing the expression `L1 == L2`) when you meant to test for object equality (e.g., `id(L1) == id(L2)`)?
  - Forgotten that some built-in function has a side effect?
  - Forgotten the `()` that turns a reference to an object of type `function` into a function invocation?
  - Created an unintentional alias?
  - Made any other mistake that is typical for you?

# Bisection approach

- Did the code break before or after this point?
  - decide if it was before or after
  - "adjust" the search space, i.e. move your breakpoints or print statements

## Explain the code/algorithm to someone else

Pretend you are teaching someone what you have done - don't necessarily explain the test failure, but what logic you MEANT to use.

## Colab - testing

# Chapter 9 - Exceptions

- ZeroDivisionError
- TypeError
- ValueError
- IndexError
- NameError
- UnboundLocalError
- <https://docs.python.org/3/library/exceptions.html>


All of these and many more are built in exception.

Python knows these names like they are global variables

```
try
    code block
except (list of exception names):
    code block
else:
    code block
```

# Exceptions prevent crashes

```
try:  
    print(get_ratios([1, 2, 7, 6], [1, 2, 0, 3]))  
    print(get_ratios([], []))  
    print(get_ratios([1, 2], [3]))  
except ValueError as msg:  
    print(msg)
```



something goes  
wrong here

but...this code does NOT crash!!!  
it simply prints the string in msg, and then continues

# Assert

The Python `assert` statement provides programmers with a simple way to confirm that the state of a computation is as expected. An **assert statement** can take one of two forms:

```
assert Boolean expression
```

or

```
assert Boolean expression, argument
```

When an `assert` statement is encountered, the Boolean expression is evaluated. If it evaluates to `True`, execution proceeds on its merry way. If it evaluates to `False`, an `AssertionError` exception is raised.

# Colab - Exceptions