COMPSCI 101
Principles of Programming

Lecture 28 – Docstrings, Doctests

Learning outcomes

- At the end of this lecture, students should be able to
  - use Doctest by including simple tests in function docstrings

docstring

- A docstring is a special kind of string used to provide documentation
  - Appears at the top of every program
  - three double-quotes are used to surround the docstring
  - All programs should include a docstring at the beginning of the program
  - The docstring contains the author and usually a version number
  - As well as the docstring describing the purpose of the program, amost important recommendation is the common sense: be short, clear and concise!

def get_the_fib(which_fib):

```
"""Prints the minutes given hours and minutes
    Author: Adriana Ferraro
"""
def main():
    hours = 5
    minutes = 23
    total_minutes = hours * 60 + minutes
    print(total_minutes)
main()
```

docstring

Errors

- No matter how smart or how careful you are, errors are your constant companion.
- With practice, you will get better at not making errors, and much, much better at finding and correcting them.
- There are three kinds of errors:
  - syntax errors,
  - runtime errors, and
  - logic errors.

Syntax Errors

- These are errors where Python finds something wrong with your program, and you can't execute it.
  - mostly typos - missing punctuation , wrong indentation, case sensitive …
- Syntax errors are the easiest to find and correct. The compiler will tell you where it got into trouble. Usually the error is on the exact line indicated by the compiler, or on the line just before it;

```
def main():
    number = 4
    print(number)
    for i in range(1, number)
        print("hello"
main()
```

No output regarding the number

Missing colon, missing ')'

```
File "Example01.py", line 4
    for i in range(1, number)
                             ^
SyntaxError: invalid syntax
```

Execution / Runtime Errors

- If there are no syntax errors, Python may detect an error while your program is running
- For example: IndexError, Division by 0 etc
- Runtime errors are moderate in difficulty. Python tells you where it discovered that your program went wrong, but you need to trace back from there to figure out where the problem originated.

```
def main():
    number = 0
    print(number)
    print(230 / number)

main()
```

the interpreter tries to give useful information

Output:

```
0
Traceback (most recent call last):
File "Example01.py", line 6, in <module>
    main()
File "Example01.py", line 4, in main
    print(230 / number)
ZeroDivisionError: division by zero
```

Logical Errors

Example01.py

DEMO

- A **logical error**, or **bug**, is when your program compiles and runs, but does the **wrong** thing.
- The Python system, of course, has no idea what your program is **supposed to do**, so it provides **no** additional information to help you find the error.
- Logical errors are often difficult to find and correct.
- Example: We would like to print a string in a reverse order:
  - The expected output is "l a c l g o l"

```
def main():
  word = "logical"
  for i in range(len("word")-1, -1, -1):
     print(word[i], end=" ")

main()
```

Actual Output!

```
i g o l
```

What is wrong?

Types of errors continued

- Logical – harder to find, harder to correct

A

```
x = int(input("x: "))
y = int(input("y: "))

if x > 10:
    if y == x:
        print("Fine")
else:
    print("So what?")
```

```
x: 3
y: 3
```

B

```
x = int(input("x: "))
y = int(input("y: "))

if x > 10:
    if y == x:
        print("Fine")
    else:
        print("So what?")
```

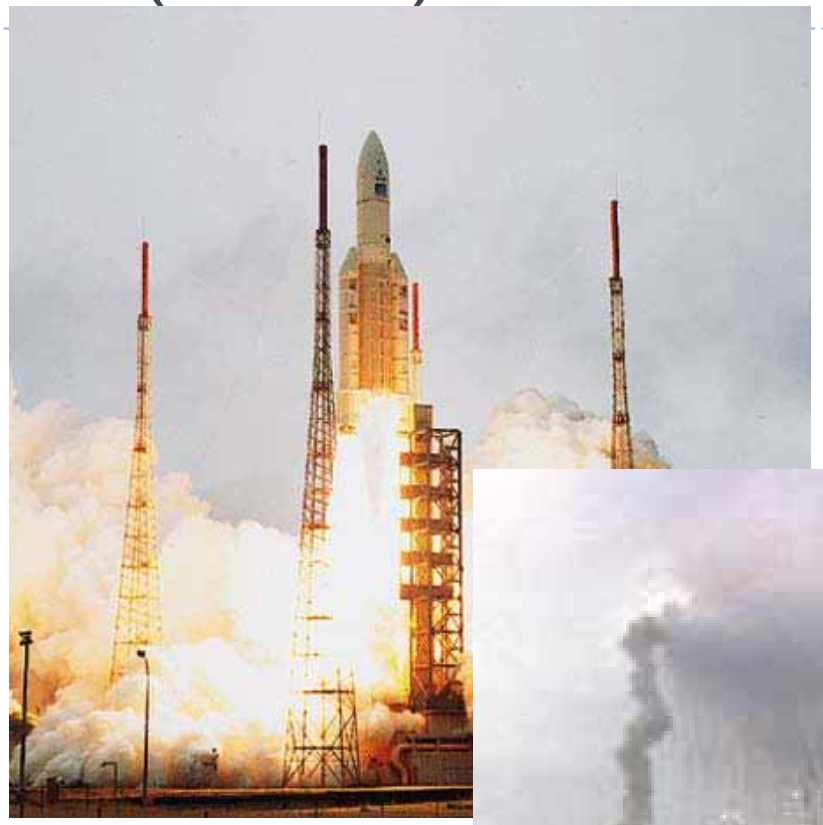
```
x: 3
y: 3
```

- Complete the output for code A and code B above?
- Which was the intention?

Testing is important!

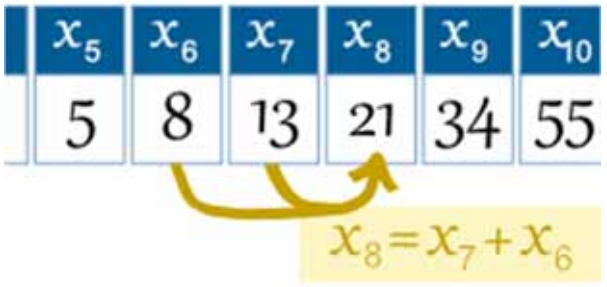
Expensive Fireworks (1996)

- In 1996, code from the Ariane 4 rocket is reused in the Ariane 5, but the new rocket's faster engines trigger a bug in an arithmetic routine inside the flight computer.
- The error is in code to convert 64-bit floating-point numbers to a 16-bit signed integers. The faster engines cause the 64-bit numbers to be larger, triggering an overflow condition that crashes the flight computer.
- As a result, the rocket's primary processor overpowers the rocket's engines and causes the rocket to disintegrate only 40 seconds after launch.



The Fibonacci Sequence

- The Fibonacci Sequence is the series of numbers:
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
  - The next number is found by adding up the two numbers before it.
  - The 2 is found by adding the two numbers before it (1+1)
  - Similarly, the 3 is found by adding the two numbers before it (1+2),
  - And the 5 is (2+3),
  - and so on!
  - Here is a longer list:



n	1	2	3	4	5	6	7	8	9	10	11	12
	1	1	2	3	5	8	13	21	34	55	89	144

First Attempt:

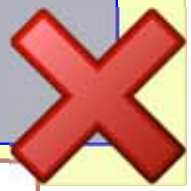
Example02.py

DEMO

- Complete the following function which prints the fibonacci numbers up to but **not including** up_to_number:

```
def print_fibs(up_to_number):
   prev_fib = 1
   next_fib = 1
   while next_fib < up_to_number:
        print(next_fib, end=" ")
        prev_fib, next_fib = next_fib, next_fib + prev_fib


print("up to 20:", end = " ")
print_fibs(20)
print("up to -4:",end = " ")
print_fibs(-4)
...
```

```
up to 20: 1 2 3 5 8 13
up to -4:
up to 0:
up to 1:
up to 2: 1
```



In order to test the correctness of the function, we need to check with different parameters (valid and invalid values)

Missing some values

Example 3

Example02.py

DEMO

- Returns a list of the required number (given by **how_many**) of fibonacci numbers:

```
def get_fibs_list(how_many):
def get_fibs_list(how_many):
   prev_fib = 0
   next_fib = 1
   fib_list = []

   while len(fib_list) < how_many:


        prev_fib, next_fib = next_fib, next_fib + prev_fib
   return fib_list

print("List of first 5 fib numbers:", get_fibs_list(5))
...
```

List of first 5 fib numbers [1, 1, 2, 3, 5]

List of first 0 fib numbers: []

List of first -2 fib numbers: []

Example 4

Example02.py

DEMO

- Returns the nth (given by which_fib) fibonacci number:

```
def get_the_fib(which_fib):
    if which_fib < 1:
        return 0
    prev_fib = 0
    next_fib = 1
    term_number = 0
    while term_number < which_fib:
        prev_fib, next_fib = next_fib, next_fib + prev_fib
        term_number += 1
    return next_fib

    print("Get fib number 6:", get_the_fib(6))
```

Get fib number 6: 13

Get fib number 0: 0

Get fib number -2: 0

Get fib number 4: 5

n	1	2	3	4	5	6	7	8	9	10	11	12
	1	1	2	3	5	8	13	21	34	55	89	144

Using the interactive interpreter

- Note:The interactive interpreter can be used to check and run Python code interactively.

```
>>> def get_result(command, what_to_do, where):
        return command + " " + what_to_do + " in the " + where


>>> get_result("a", "b", "c")
'a b in the c'
>>> get_result("come", "sing", "hall")
'come sing in the hall'
>>> get_result("go", "jump", "pond")
'go jump in the pond
```

Remember – using docstrings

- We used docstrings to state the purpose of the program and to print the module author.
  - This is the program documentation.
  - Remember: be short, clear and concise! Other programmers, who use/improve your module, will be using your docstring as documentation.
  - Docstrings can also be added to our functions. A docstring containing the purpose of the function should be added to the docstring.

```
def get_the_fib(which_fib):
    """Returns the nth (given by which_fib) Fibonacci number.
    """

    prev_fib = 0
    next_fib = 1
    ...
```

Testing using doctest module

Example03.py

DEMO

- Put all your **test cases** into your docstrings

```
def cube(x):
    """
    returns ...
    >>> cube(0)
    0
    >>> cube(1)
    1
    >>> cube(2)
    8
    >>> cube(10)
    1000
    """
    return x * x

import doctest
doctest.testmod()
```

Test cases

```
File "Example02.py", line 7, in __main__.cube
Failed example:
    cube(2)
Expected:
    8
Got:
    4
**********************************************************************
File "Example02.py", line 9, in __main__.cube
Failed example:
    cube(10)
Expected:
    1000
Got:
    100
**********************************************************************
...
***Test Failed*** 2 failures.
```

Test Failed

Test Failed

Doctests – does the testing

- If we want to include doctests in functions, we need to include the following two statements at the end of our code:

```
def get_the_fib(which_fib):
  ...


import doctest
doctest.testmod()
```

These two statements are the last two statements of the program

import doctest – imports the doctest module
doctest.testmod() – starts the testing of the module

Doctests –testmod() does the testing

- A docstring can also contain testing code.
- Any code in our function docstrings which looks like interactive code,
  - i.e., any line in the docstring which starts with the interactive interpreter prompt, ">>>" will be executed and the outcome of the code will be compared with the stated expected outcome.

```
def get_the_fib(which_fib):
    """Returns the nth (given by which_fib) Fibonacci number.
    >>> this code will be executed by testmod()
    this is the expected outcome from executing the previous line of code
    """
    ...

import doctest
doctest.testmod()
```

Running a program which contains doctests

- Note that in the program a main() function can be included or it can be **left out** if you just wish to just run the doctests.

- When you run the doctests (e.g., run the program on the previous slide), there is no output **if the tests cause no problem**, i.e., if the outcome of the tests **is exactly the same** as the outcome stated.

- If the outcome of the test is different, then the test **fails** and the doctest gives useful information.

Testing using the doctest module

- Put all your **test cases** right into your doc strings
- When this program is run, there is no output because all the doctests pass.

```
def cube(x):
    """
    returns ...
    >>> cube(0)
    0
    >>> cube(1)
    1
    >>> cube(2)
    8
    >>> cube(10)
    1000
    """
    return x * x * x

import doctest
doctest.testmod()
```

```
python Example03.py
C:\Python33\Python "Example03.py"
Process started >>>
<<< Process finished. (Exit code 0)
```

No output!

Running with "–v"

- Run your program using -v option, and doctest prints a detailed log of what it's trying, and prints a summary at the end:

```
def cube(x):
    """
    returns ...
    >>> cube(0)
    0
    >>> cube(1)
    1
    >>> cube(2)
    8
    >>> cube(10)
    1000
    """
    return x * x * x

import doctest
doctest.testmod()
```

python Example02.py -v

```
…
Trying:
    cube(2)
Expecting:
    8
ok
Trying:
    cube(10)
Expecting:
    1000
ok
…
4 passed and 0 failed.
Test passed.
```

Common Problem 1

Example05.py

DEMO

- No blank space after the '>>>' prompt sign:

Missing space

```
def my_function(a, b):
    """
    >>>my_function(2, 3)
    6
    """
    return a * b
```

```
Traceback (most recent call last):
  File "Example03.py", line 12, in
<module>
    doctest.testmod()
…
```

Common Problem 2

Example06.py

DEMO

- If the outcome doesn't match exactly (including trailing spaces), the test fails, e.g.,
  - Example: embedded whitespace can also cause tricky problems with tests. This example has a single extra space after the 6.

An extra space

```
def my_function(a, b):
    """
    >>> my_function(2, 3)
    6
    >>> my_function('a', 3)
    'aaa'
    """
    return a * b

import doctest
doctest.testmod()
```

unnoticed in the source file and invisible in the test failure report

```
Failed example:
    my_function(2, 3)
Expected:
    6
Got:
    6
**********************************************************************
1 items had failures:
   1 of   2 in __main__.my_function
***Test Failed*** 1 failures.
```

Common Problem 3

Example07.py

DEMO

- No blank line after the expected outcome – in this case any text on the next line is considered to be part of the output, e.g.,

```
def my_function(a, b):
    """
    >>> my_function(2, 3)
    6
    more comment
    >>> my_function('a', 3)
    'aaa'
    """
    return a * b
...
```

Doctest considers that the line "more comment" is part of the output. Therefore the test fails.

```
Failed example:
    my_function(2, 3)
Expected:
    6
    more comment
Got:
    6
**********************************************************************
1 items had failures:
   1 of   2 in __main__.my_function
***Test Failed*** 1 failures.
```

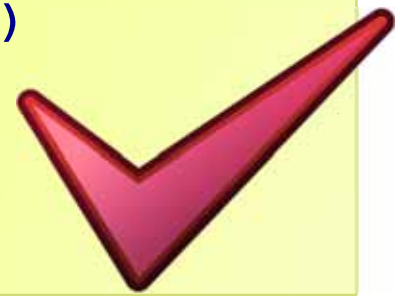
Blank lines are used to delimit tests.

- In real world applications, output usually includes whitespace such as blank lines, tabs, and extra spacing to make it more readable.
- Blank lines, in particular, cause issues with doctest because they are used to delimit tests.

```
Process started >>>
<<< Process finished. (Exit code 0)
```

```
def my_function(a, b):
    """
    >>> my_function(2, 3)
    6

    >>> my_function('a', 3)
    'aaa'
      """
    return a * b
...
```

delimit tests



Common Problem! - 4

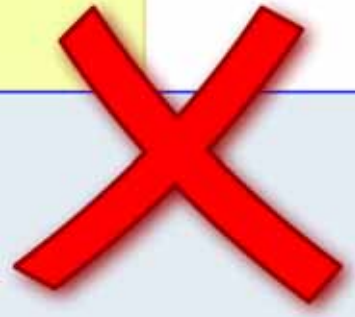
Example08.py

DEMO

- Write a function which takes a list of input lines, and prints them double-spaced with **blank lines** between.

```
def double_space(lines):
    """Prints a list of lines double-spaced.

    >>> double_space(['Line one.', 'l
    Line one.

    Line two.

    """
    for l in lines:
        print(l)
        print()
    return

import doctest
doctest.testmod()
```

interprets the blank line after Line one. in the docstring as the end of the sample output



```
Expected:
    Line one.
    Line two.
Got:
    Line one.
    <BLANKLINE>
    Line two.
    <BLANKLINE>
**********************************************
*****************************
1 items had failures:
   1 of   1 in __main__.double_space
***Test Failed*** 1 failures.
```

Solution

- Using <BLANKLINE>

```
def double_space(lines):
  """Prints a list of lines double-spaced.
  >>> double_space(['Line one.', 'Line two.'])
  Line one.
  <BLANKLINE>
  Line two.
  <BLANKLINE>
    """
  for l in lines:
    print(l)
    print()
  return

import doctest
doctest.testmod()
```

doctests – exercise 1

Exercise01.py

```
def get_the_fib(which_fib):
    """Returns the nth Fibonacci number.

    >>> get_the_fib(8)
    21
    >>> get_the_fib(5)
    5
    """
    if which_fib < 1:
        return 0
    prev_fib = 0
    next_fib = 1
    fib_number = 0
    while fib_number < which_fib:
        prev_fib, next_fib = next_fib, next_fib + prev_fib
        fib_number += 1
    return next_fib

import doctest
doctest.testmod()
```

Do the two doctests pass or fail?

doctests – exercise 2

Exercise02.py

```
def get_fibs_list(how_many):
    """Returns a list of Fibonacci numbers.
    The parameter is the number of terms in the list.



    """
    prev_fib = 0
    next_fib = 1
    fib_list = []


    while len(fib_list) < how_many:
        fib_list.append(next_fib)
        prev_fib, next_fib = next_fib, next_fib + prev_fib
    return fib_list

import doctest
doctest.testmod()
```

Write two (useful and different) doctests for the get_fibs_list() function.

Exercise03.py

Converting Celsius - Fahrenheit

- Often, before writing the code, we know what outcomes we are expecting. These expected outcomes can be added to the function being developed using doctests.

```
def c_to_f(celsius):
        """Returns the parameter degrees
        converted to fahrenheit.

        >>> c_to_f(0)
        32.0
        >>> c_to_f(37.8)

        >>> c_to_f(-32)

        """

import doctest
doctest.testmod()
```

Celsius	Fahrenheit
-35.000	-31.000
-30.000	-22.000
-25.000	-13.000
-20.000	-4.0000
-15.000	5.0000
-10.000	14.000
-5.0000	23.000
0.0000	32.000
5.0000	41.000
10.000	50.000
15.000	59.000
20.000	68.000
25.000	77.000
30.000	86.000
35.000	95.000
40.000	104.00
45.000	113.00
50.000	122.00
55.000	131.00

Summary

- In a Python program:
  - docstrings can be associated with modules and with functions
  - simple tests can can be added to the docstring of a function. These tests are automatically carried out.