Lecture 28 – Docstrings, Doctests
Learning outcomes

At the end of this lecture, students should be able to

• use Doctests by including simple tests inside function docstrings
```python
def main():
a_list1 = [4, 3]
a_list2 = [1, 3, 4]
function_17(a_list1, a_list2)
print("a_list1:", a_list1)
print("a_list2:", a_list2)

def function_17(list1, list2):
    list3 = list2
    for i in range(len(list1)):
        list3.append(list1[i])
        list2.append(list1[i])
    list1 = list3
    print("list3:", list3)
    print("list3:", list3)

main()
```

**list3:** [1, 3, 4, 4, 4, 3, 3]

**a_list1:** [4, 3]

**a_list2:** [1, 3, 4, 4, 4, 3, 3]
Types of errors

**syntax** (mostly typos - missing punctuation (missing colons, missing brackets), wrong indentation, case sensitive) – the interpreter tells you where.

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

**execution** (KeyError, IndexError, Division by 0)

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

```python
File "ExecutionError.py", line 2, in main
    print(230 / number)
ZeroDivisionError: division by zero
```
Logical errors – harder to find, harder to correct

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

What is the output?

Was this the intention?
Logical errors – harder to find, harder to correct

A

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

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

B

```python
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 (at both prompts the user enters 3)?

Which was the intention?
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 series – complete the function

The Fibonacci numbers:

<table>
<thead>
<tr>
<th>n</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>nth Fibonacci:</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>55</td>
<td>89</td>
<td>...</td>
</tr>
</tbody>
</table>

Complete the following function which prints the fibonacci numbers up to but not including the parameter, up_to_number:

```python
# Prints the Fibonacci numbers up to the parameter number
def print_fibs(up_to_number):
    prev_fib = 0
    next_fib = 1
    for _ in range(up_to_number):
        fib = prev_fib + next_fib
        prev_fib, next_fib = next_fib, fib
        if _ < up_to_number:
            print(fib, end=' ') 

def main():
    print("Fibs up to 50:", end = " ")
    print_fibs(50)

main()
```

Fibs up to 50: 1 1 2 3 5 8 13 21 34
The Fibonacci series—a third function

Returns the nth (given by \texttt{which\_fib}) fibonacci number:

\begin{verbatim}
def get_the_fib(which_fib):
    prev_fib = 0
    next_fib = 1
    count = 1
    while number < which_fib:
        ex_next_fib = next_fib
        next_fib = next_fib + prev_fib
        prev_fib = ex_next_fib
        number = number + 1
    return next_fib

def main():
    print("Fib number 30:", get_the_fib(30))

main()
\end{verbatim}

\texttt{Fib number 30: 832040}
The Fibonacci series – testing function (slide 8)

```python
#Prints the Fibonacci numbers up to the parameter number

def print_fibs(up_to_number):
    prev_fib = 0
    next_fib = 1
    while next_fib < up_to_number:
        print(next_fib, end=" ")
        prev_fib, next_fib = next_fib, next_fib + prev_fib
    print()

def main():
    print("Fibs up to 50:", end = " ")
    print_fibs(50)
    print("Fibs up to 2:", end = " ")
    print_fibs(2)

main()
```

Fibs up to 50: 1 1 2 3 5 8 13 21 34

Fibs up to 2: 1 1
The Fibonacci series – testing function (slide 9)

```python
def get_fibs_list(how_many):
    # Returns a list of Fibonacci numbers
    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

def main():
    print("List of first 10 fib numbers:", get_fibs_list(10))
    print("List of first 5 fib numbers:", get_fibs_list(5))

main()
```

List of first 10 fib numbers: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
List of first 5 fib numbers: [1, 1, 2, 3, 5]
The Fibonacci series – testing function (slide 10)

def get_the_fib(which_fib): #Returns the nth Fibonacci number
    prev_fib = 0
    next_fib = 1
    count = 1

    while count < which_fib:
        prev_fib, next_fib = next_fib, next_fib + prev_fib
        count = count + 1
    return next_fib

def main():
    print("Fib number 5:", get_the_fib(5))
    print("Fib number 8:", get_the_fib(8))
    print("Fib number 30:", get_the_fib(30))
    print("Fib number 1:", get_the_fib(1))
    print("Fib number 2:", get_the_fib(2))

main()
Remember – using the interactive interpreter

The interactive interpreter can be used to check and run Python code interactively (See the previous lecture).

```python
>>> 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'
```

When you are in 'interactive interpreter' mode, the prompt is `>>>`. 
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, an important recommendation is: **be short, be clear, be concise**!

The general format of a Python program is:

```python
"""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()
```
Attach a docstring to a function

We have used docstrings to state the purpose of the program and to print the module author. This is the program documentation. Other programmers, who use/improve your module, will be using your docstring as documentation.

Docstrings can also be added to functions. The docstring should contain the purpose of the function.

```python
def get_the_fib(which_fib):
    """Returns the nth (given by which_fib) Fibonacci number."""
    prev_fib = 0
    next_fib = 1
    count = 1
    if which_fib < 1:
        return 0
    while count < which_fib:
        prev_fib, next_fib = next_fib, next_fib + prev_fib
        count = count + 1
    return next_fib
```

Docstring attached to a function
Doctests – import doctest, testmod() 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:

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

def main()
    fib22 = get_the_fib(22)
    print(fib22)

import doctest
doctest.testmod()
```

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

Python provides a very easy way to automate the testing of functions using a system called doctest. The docstring can 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.

```python
import doctest
doctest.testmod()
```

```
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 when the previous line of code is executed
    """
    ...
```

"""this code will be executed by testmod()"
"""this is the expected outcome when the previous line of code is executed"""
```python
def print_fibs(up_to_number):
    """Prints the Fibonacci numbers up to the parameter number""

    >>> print_fibs(50)
    1 1 2 3 5 8 13 21 34
    >>> print_fibs(2)
    1 1
    """

    prev_fib = 0
    next_fib = 1

    while next_fib < up_to_number:
        print(next_fib, end=" ")
        prev_fib, next_fib = next_fib, next_fib + prev_fib

import doctest
doctest.testmod()
```
Running a program which contains doctests

Note that in a program the main() function can be included or, it can be left out if you 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 in the doctest.

If the outcome of the test is different, then the test fails and the doctest gives useful information.
Running a program which contains doctests

When the program from Slide 19 is run, there is no output because both the doctests pass.

```
print_fibs(50)
```

prints (note: in the output below there is a single space after 34):

```
1 1 2 3 5 8 13 21 34
```

and,

```
print_fibs(2)
```

prints (note: in the output below there is a single space after 1):

```
1 1
```

Note that the outcomes have to be exactly as expected - spacing and content.
If the outcome doesn't match exactly (including trailing spaces), the test fails, e.g.,

```python
def print_fibs(up_to_number):
    """Prints the Fib numbers."
    >>> print_fibs(50)
    1 1 2 3 5 8 13 21
    >>> print_fibs(2)
    1 """
    ...

import doctest
doctest.testmod()
```

What if a doctest fails?

File "MyFibDoctest.py", line 7, in __main__.print_fibs
Failed example:
    print_fibs(50)
Expected:
    1 1 2 3 5 8 13 21
Got:
    1 1 2 3 5 8 13 21 34

File "MyFibDoctest.py", line 9, in __main__.print_fibs
Failed example:
    print_fibs(2)
Expected:
    1 1
Got:
    1
Other problems causing a doctest to fail

1. Problem - 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.

```python
def print_fibs(up_to_number):
    """Prints the Fib numbers.
    >>> print_fibs(50)
    1 1 2 3 5 8 13 21 34
    blah! Blah! Blah!
    " "
    ...
```

Doctest considers that the line "blah! Blah! Blah!" is part of the output. Therefore the test fails.

2. Problem - No blank space after the '>>>' prompt sign:

```python
def print_fibs(up_to_number):
    """Prints the Fib numbers.
    >>> print_fibs(50)
    1 1 2 3 5 8 13 21 34
    " "
    ...
```

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

    >>> get_the_fib(8)
    21
    """

    prev_fib = 1
    next_fib = 1
    count = 1

    while count < which_fib:
        prev_fib, next_fib = next_fib, next_fib + prev_fib
        count = count + 1

    return next_fib

import doctest
doctest.testmod()
Doctest feedback from the previous slide

File "MyFibDoctest.py", line 12, in __main__.get_the_fib
Failed example:
    get_the_fib(8)
Expected:
    21
Got:
    34
```python
def get_the_fib(which_fib):
    '''Returns the nth Fibonacci number.

    >>> get_the_fib(8)
    21
    '''
    prev_fib = 0
    next_fib = 1
    count = 1
    while count < which_fib:
        prev_fib, next_fib = next_fib, next_fib + prev_fib
        count = count + 1

    return next_fib
```

The slide 24 function should have been defined as shown here. Now the doctest passes.
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()
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 by using doctests.

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

    >>> c_to_f(0)
    32.0
    >>> c_to_f(37.8)
    100.0
    >>> c_to_f(-32)
    0.0

import doctest
doctest.testmod()
```

<table>
<thead>
<tr>
<th>Celsius</th>
<th>Fahrenheit</th>
</tr>
</thead>
<tbody>
<tr>
<td>-35.000</td>
<td>-31.000</td>
</tr>
<tr>
<td>-30.000</td>
<td>-22.000</td>
</tr>
<tr>
<td>-25.000</td>
<td>-13.000</td>
</tr>
<tr>
<td>-20.000</td>
<td>-4.0000</td>
</tr>
<tr>
<td>-15.000</td>
<td>5.0000</td>
</tr>
<tr>
<td>-10.000</td>
<td>14.000</td>
</tr>
<tr>
<td>-5.0000</td>
<td>23.000</td>
</tr>
<tr>
<td>0.0000</td>
<td>32.000</td>
</tr>
<tr>
<td>5.0000</td>
<td>41.000</td>
</tr>
<tr>
<td>10.000</td>
<td>50.000</td>
</tr>
<tr>
<td>15.000</td>
<td>59.000</td>
</tr>
<tr>
<td>20.000</td>
<td>68.000</td>
</tr>
<tr>
<td>25.000</td>
<td>77.000</td>
</tr>
<tr>
<td>30.000</td>
<td>86.000</td>
</tr>
<tr>
<td>35.000</td>
<td>95.000</td>
</tr>
<tr>
<td>40.000</td>
<td>104.00</td>
</tr>
<tr>
<td>45.000</td>
<td>113.00</td>
</tr>
<tr>
<td>50.000</td>
<td>122.00</td>
</tr>
<tr>
<td>55.000</td>
<td>131.00</td>
</tr>
</tbody>
</table>
def c_to_f(celsius):
    """Returns the parameter degrees converted to fahrenheit."

    >>> c_to_f(0)
    32.0
    >>> c_to_f(37.8)
    100.0
    >>> c_to_f(-32)
    0.0

    fahrenheit = celsius * 9.0 / 5 + 32
    return round(fahrenheit, 1)

import doctest
doctest.testmod()
In a Python program:

- docstrings can be associated with modules and also with functions
- simple tests can be added to the docstring of a function. These tests are automatically carried out when the program is run even if the program does not contain a main() function.
Examples of Python features used in this lecture

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

    >>> c_to_f(0)
    32.0
    >>> c_to_f(37.8)
    100.0
    >>> c_to_f(-32)
    0.0

    """

    fahrenheit = celsius * 9.0 / 5 + 32
    return round(fahrenheit, 1)

import doctest
doctest.testmod()
```