

# A gamified introduction to Python Programming

## Lecture 10

# The art of Testing and Debugging

Matthieu DE MARI – Singapore University of Technology and Design



SINGAPORE UNIVERSITY OF  
TECHNOLOGY AND DESIGN



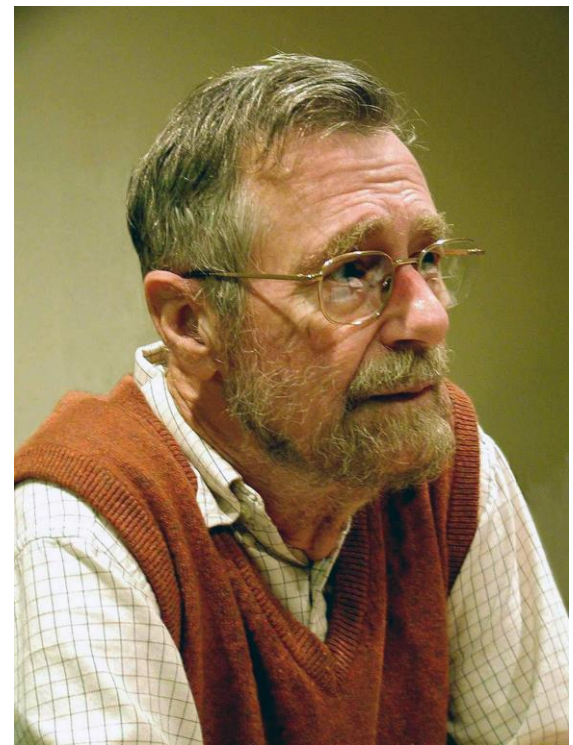
## Outline (Chapter 10)

- What is **testing** in programming?
- What is a **unit test**?
- What is the **assert** keyword? How is it related to **error messages**?
- What is therefore the **right philosophy for programming and testing**?
- What is an **integration test**?
- What is **debugging**?
- What are **good practices in debugging**?

# Testing: a definition?

“Program testing can be used to show the presence of bugs, but never to show their absence!”

– Edsger W. Dijkstra

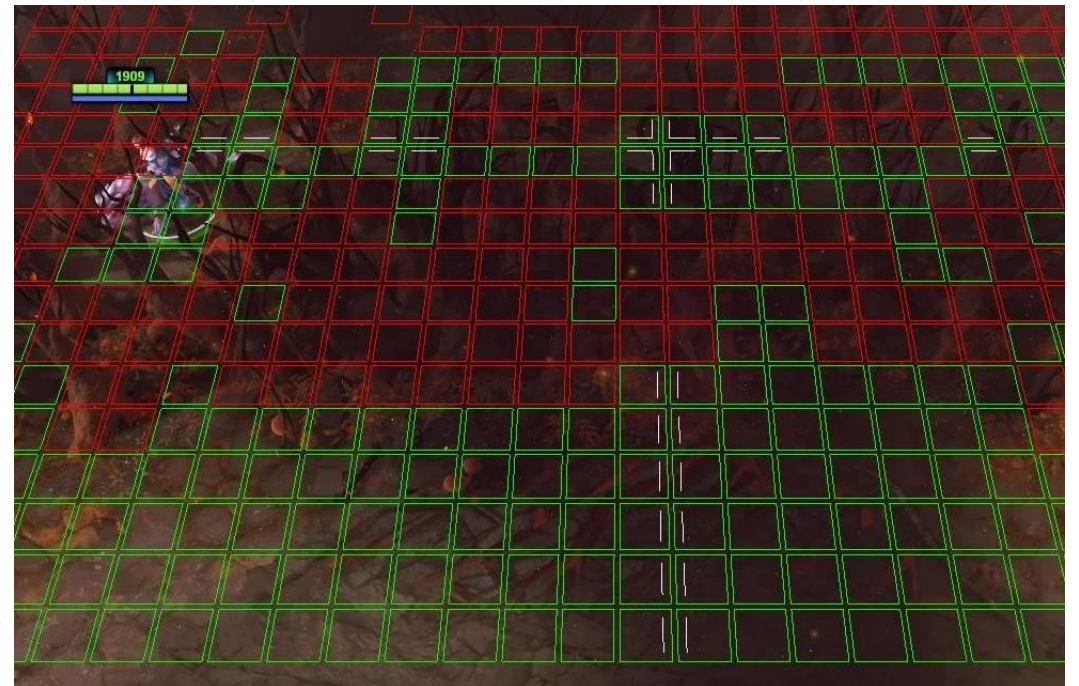
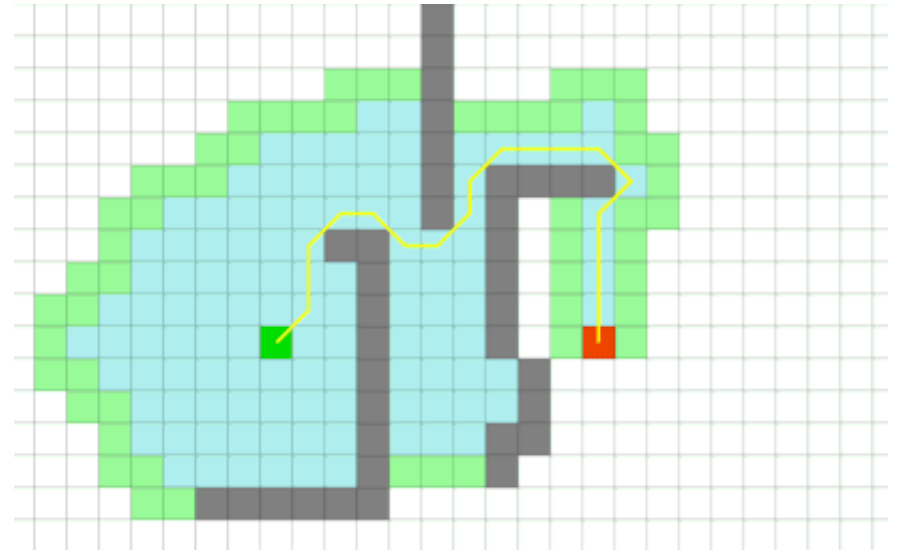


# Testing: a definition?

“Program testing can be used to show the presence of bugs, but never to show their absence!”

– Edsger W. Dijkstra

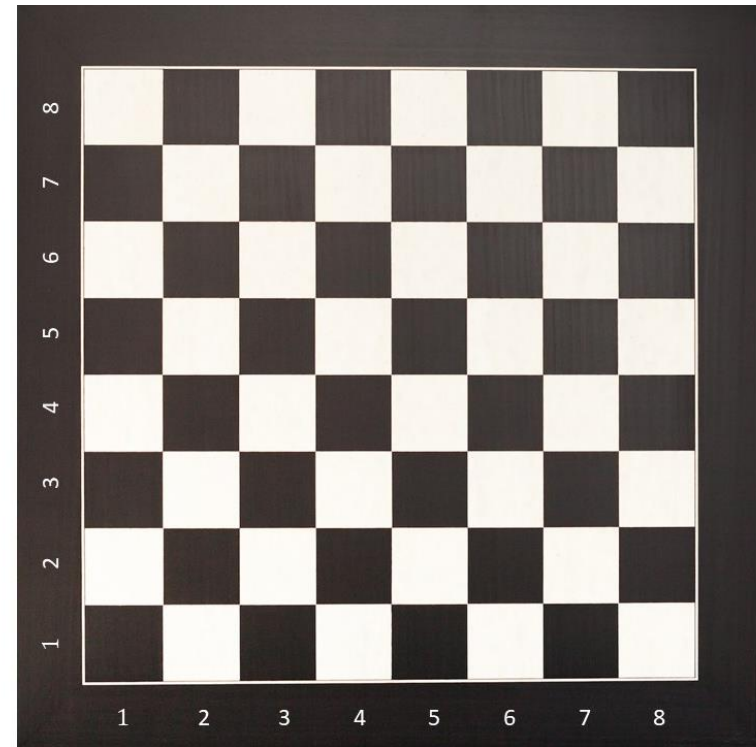
**FYI:** Dijkstra is famous for his algorithm, used for path-finding in many applications (GPS, video games, etc.)



# Testing: a definition?

**Program testing** is the process of executing a program, with the objective of finding errors.

- **Program testing** cannot show the absence of errors. It can only show if errors are present.
- **Test driven development:** It is often preferable to identify and write the tests before coding the program. This is something I have done for you on early activities, your job now!



## Expected results

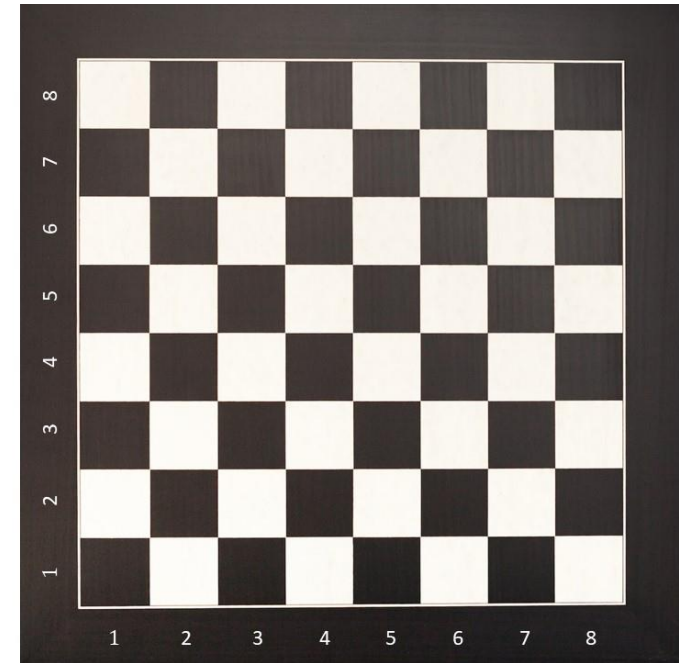
A few test cases for you to try:

- The square at coordinates (4,5) is white.
- The square at coordinates (2,2) is black.
- The square at coordinates (4,6) is black.
- The square at coordinates (3,5) is black.



# Limits of testing

- **Issue #1:** Program testing cannot show the absence of errors. It can only show if errors are present.
- **Issue #2:** In the ideal world, all possible paths through a programs should be tested.
- But it is often impossible.
  - E.g., I can test all squares of an 8x8 grid, because there is a finite number of them.
  - But I cannot test all possible integers, lists of integers, etc.



## Expected solutions

If your function has been correctly designed, the following test cases should work.

```
1  ### Some test cases
2  # This should return True
3  print(is_triangle_rectangular(a = 3, b = 4, c = 5))
4
5  # This should return False
6  print(is_triangle_rectangular(a = 4, b = 4, c = 6))
7
8  # This should return False
9  print(is_triangle_rectangular(a = 3, b = 5, c = 4))
10
11 # This should return True
12 print(is_triangle_rectangular(a = 0, b = 0, c = 0))
```

# Unit testing

**Definition (Unit testing):** **Unit tests** are test cases, whose objective is to verify that a **single** function is able to operate as expected, on a specifically identified use case.

- Often a good idea to start simple, and progressively look for more complex/special cases.
- Let us demonstrate on the next slides.

```
1 def function(my_list):  
2     # Remove max and min from list  
3     min_val = min(my_list)  
4     max_val = max(my_list)  
5     while(min_val in my_list):  
6         my_list.remove(min_val)  
7     while(max_val in my_list):  
8         my_list.remove(max_val)  
9     return my_list
```

```
1 # Test case 1: a normal, good looking list,  
2 # with a single max value and a single min value  
3 my_list = [1,2,3,4,5]  
4 print(function(my_list))
```

[2, 3, 4]



```
1  # Test case 2: a normal, good looking list,  
2  # with multiple occurrences of the max and min values  
3  my_list = [1,1,2,3,4,5,5,5]  
4  print(function(my_list))
```

[2, 3, 4]

```
1  # Test case 3: a list with only min and max values  
2  my_list = [1,1,5,5,5]  
3  print(function(my_list))
```

[]

```
1  # Test case 4: a list with only min and max values,  
2  # and the min/max values are identical  
3  my_list = [5,5,5]  
4  print(function(my_list))
```

[]

```
1 # Test case 5: an empty list
2 my_list = []
3 print(function(my_list))
```

-----

**ValueError**

Traceback (most recent call last)

<ipython-input-9-a5d51a1c5688> in <module>

```
1 # Test case 5: an empty list
2 my_list = []
----> 3 print(function(my_list))
```

<ipython-input-3-02cd6a8e44a3> in function(my\_list)

```
1 def function(my_list):
2     # Remove max and min from list
----> 3     min_val = min(my_list)
4     max_val = max(my_list)
5     while(min_val in my_list):
```

**ValueError:** min() arg is an empty sequence

# Unit testing

**Definition (Unit testing):** **Unit tests** are test cases, whose objective is to verify that a **single** function is able to operate as expected, on a specifically identified use case.

- Often a good idea to start simple, and progressively look for more complex/special cases.
- Let us demonstrate on the next slides.
- Whenever a test case is identified as not working, **amend** the function to cover for this special case if needed.

```
1 def function_v2(my_list):
2
3     # Warning: Need to cover for empty list case
4     if len(my_list) == 0:
5         return []
6     else:
7         # Remove max and min from list
8         min_val = min(my_list)
9         max_val = max(my_list)
10        while (min_val in my_list):
11            my_list.remove(min_val)
12        while (max_val in my_list):
13            my_list.remove(max_val)
14        return my_list
```

```
1 # Test case 5: an empty list
2 my_list = []
3 print(function_v2(my_list))
```

[]

# Unit testing

**Definition (Unit testing):** **Unit tests** are test cases, whose objective is to verify that a **single** function is able to operate as expected, on a specifically identified use case.

- Often a good idea to start simple, and progressively look for more complex/special cases.
- Let us demonstrate on the next slides.
- Whenever a test case is identified as not working, **amend** the function to cover for this special case if needed.
- Unit testing is a **cat-and-mouse game!**

# Unit testing is essential...

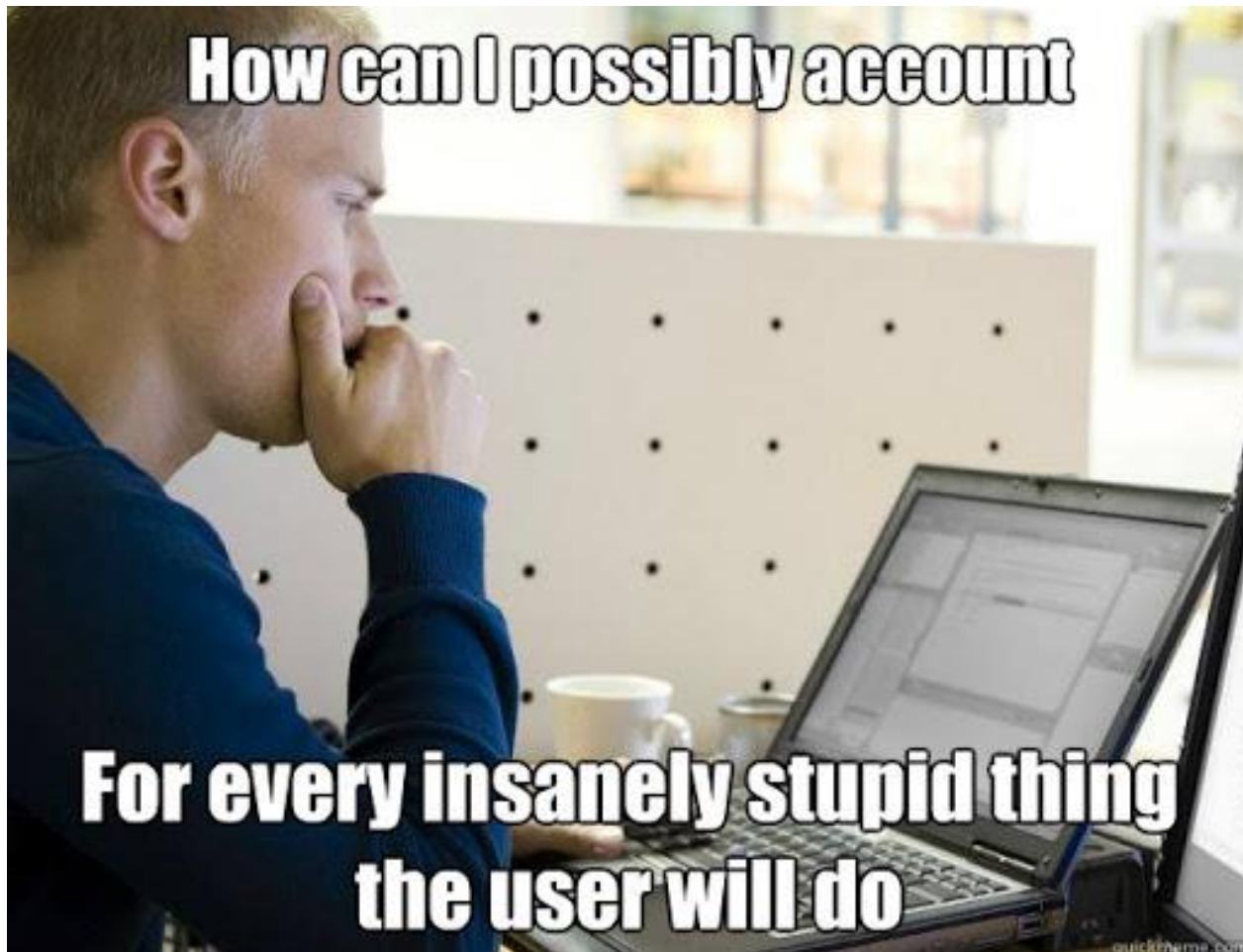
- But ultimately, unit testing is never enough.
- It is a never-ending task, as it will never be able to cover for all possible ways to break our function.

```
1  # Test case 6: a non-empty list
2  # with non-numerical values
3  my_list = ["Hello", "What", "is", "up?"]
4  # Function works without errors,
5  # but is it really the expected behavior?
6  print(function_v2(my_list))
```

```
['What', 'is']
```



# Finding the right balance?



# Assertion testing

**In some (stupid) cases, the function should simply refuse to operate.**

- It would sometimes be preferable if our function could check that it is indeed being used in an “acceptable” or “normal” setting.
- When not, it should raise error messages to inform the user that he/she is misusing the function, if so.
- That is something easily done with the **assert** keyword.

# The **assert** Keyword

**Definition (the **assert** keyword):** The **assert** keyword is used for assertion testing. It is the most basic error control structure, whose objective is to verify that a function/program is being used in its intended purpose/configuration.

- It receives a Boolean and a message in string format.
- If the Boolean is **True**, **nothing happens**.
- If **False**, the program **crashes on purpose** and the **message is displayed as an error**.

# The **assert** Keyword

```
1 bool1 = True
2 message = "Nothing will be displayed."
3 assert bool1, message
```

```
1 bool1 = False
2 message = "Assertion test failed, interrupted program, error message here."
3 assert bool1, message
```

-----  
**AssertionError**

Traceback (most recent call last)

<ipython-input-4-2370335b2e43> in <module>

1 bool1 = False

2 message = "Assertion test failed, interrupted program, error message here."

----> 3 assert bool1, message

**AssertionError:** Assertion test failed, interrupted program, error message here.

# Assertions

```
1 from numpy import cos
2 # Works as expected
3 print(cos(0))
4 # The cosine function was not
5 # designed to operate on text!
6 print(cos("Hello"))
```

1.0

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-1-9b7a6eeabc31> in <module>
      4 # The cosine function was not
      5 # designed to operate on text!
----> 6 print(cos("Hello"))
```

**TypeError:** ufunc 'cos' not supported for the input types, and the inputs could not be safely coerced to any supported types according to the casting rule ''safe''

# The **assert** Keyword

**Definition (the **assert** keyword):** The **assert** keyword is used for assertion testing. It is the most basic error control structure, whose objective is to verify that a function/program is being used in its intended purpose/configuration.

Typically, **assert** can be used to check

- That a user entered digits only, when prompted with input(),
- Mathematical functions are used with numerical types variables,
- Etc.



# Asserting on types

## Definition (type checking):

Another interesting function is the **isinstance()** one, which is used for **type checking**.

- It receives two arguments.
- The first one is a variable,
- The second a type (int, float, str, list, etc.)
- It returns **True**, if the variable is of said type and **False** otherwise.

```
1 x = 10
2 # Returns True, because x is an int type
3 print(isinstance(x, int))
4 # Returns False, because x is an int type
5 print(isinstance(x, str))
```

True

False

# An application example: our sqrt function

```
1 def my_sqrt(x):  
2     return x**0.5
```

```
1 # Test case 1: stricly positive number, perfect square  
2 x = 4  
3 print(my_sqrt(x))
```

2.0

```
1 # Test case 2: stricly positive number, not a perfect square  
2 x = 3  
3 print(my_sqrt(x))
```

1.7320508075688772

```
1 # Test case 3: zero  
2 x = 0  
3 print(my_sqrt(x))
```

0.0

# An application example: our sqrt function

```
1  # Test case 4: strictly negative number  
2  x = -10  
3  print(my_sqrt(x))
```

```
(1.9363366072701937e-16+3.1622776601683795j)
```

# An application example: our sqrt function

```
1  # Test case 4: strictly negative number
2  x = -10
3  print(my_sqrt(x))
```

(1.9363366072701937e-16+3.1622776601683795j)

```
1  def my_sqrt_v2(x):
2      error_message = "Warning: square roots with neg. values not supported."
3      assert x >= 0, error_message
4      return x**0.5
```

# An application example: our sqrt function

```
1 # Test case 4: strictly negative number
2 x = -10
3 print(my_sqrt_v2(x))
```

```
-----
AssertionError                                Traceback (most recent call last)
<ipython-input-18-7af65051bf94> in <module>
      1 # Test case 4: strictly negative number
      2 x = -10
----> 3 print(my_sqrt_v2(x))

<ipython-input-17-b087dc1361f2> in my_sqrt_v2(x)
      1 def my_sqrt_v2(x):
      2     error_message = "Warning: square roots with neg. values not supported."
----> 3     assert x >= 0, error_message
      4     return x**0.5
```

**AssertionError:** Warning: square roots with neg. values not supported.

# An application example: our sqrt function

```
1 # Test case 5: passing a non-numerical variable
2 x = "Hello"
3 print(my_sqrt_v2(x))
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-19-da2de9f8d1ba> in <module>
      1 # Test case 5: passing a non-numerical variable
      2 x = "Hello"
----> 3 print(my_sqrt_v2(x))

<ipython-input-17-b087dc1361f2> in my_sqrt_v2(x)
      1 def my_sqrt_v2(x):
      2     error_message = "Warning: square roots with neg. values not supported."
----> 3     assert x >= 0, error_message
      4     return x**0.5
```

**TypeError:** '>=' not supported between instances of 'str' and 'int'



```
1 def my_sqrt_v3(x):
2     error_message = "Warning: x must be a strictly positive number."
3     assert isinstance(x, int) or isinstance(x, float), error_message
4     assert x >= 0, error_message
5     return x**0.5
```

```
1 # Test case 5: passing a non-numerical variable
2 x = "Hello"
3 print(my_sqrt_v3(x))
```

```
-----
AssertionError                                Traceback (most recent call last)
<ipython-input-24-1686192c086b> in <module>
      1 # Test case 5: passing a non-numerical variable
      2 x = "Hello"
----> 3 print(my_sqrt_v3(x))

<ipython-input-23-66afafabbe16> in my_sqrt_v3(x)
      1 def my_sqrt_v3(x):
      2     error_message = "Warning: x must be a strictly positive number."
----> 3     assert isinstance(x, int) or isinstance(x, float), error_message
      4     assert x >= 0, error_message
      5     return x**0.5
```

**AssertionError:** Warning: x must be a strictly positive number.

# What makes a good function?

## **Definition (what makes a good function):**

We can define a good function as

1. A function that is able to operate on any of its “normal” test cases (strictly positive numbers for sqrt)
2. A function that is able to cover for special cases (empty lists, division by zero, etc.)
3. A function that raises explicative errors, if it is misused by the user (wrong variable type, not implemented feature, etc.)
4. (A function that produces results as fast as possible)

# What makes a good function?

## **Definition (what makes a good function):**

We can define a good function as

1. A function that is able to operate on any of its “normal” test cases (strictly positive numbers for sqrt)
2. A function that is able to cover for special cases (empty lists, division by zero, etc.)
3. A function that raises explicative errors, if it is misused by the user (wrong variable type, not implemented feature, etc.)
4. ~~(A function that produces results as fast as possible)~~

→ Start small, progressively build up the function with more features!

# From unit testing to integration testing

It is considered good practice to perform unit tests on **ALL** the functions you design, to control that they operate as expected.

- It is also essential to perform **integration testing**.

**Definition (integration testing):** **Integration testing** focuses on testing that several subfunctions are able to operate correctly when assembled together in a larger program.

- **Especially important when “patching” a function:** slightly modifying a single function should not make the rest of the program (game, app, etc.) malfunction!

# From unit testing to integration testing

It is considered good practice to perform unit tests on ALL the functions you design, to control that they operate as expected.

- It is also essential to perform integration testing.

**Definition (integration testing):** integration testing focuses on testing that several subfunctions are able to operate correctly when assembled together in a larger program.

- **Especially important when “patching” a function:** slightly modifying a single function should not make the rest of the program (game, app, etc.) malfunction!

**MORE ON THIS TO BE COVERED IN ADVANCED CLASSES IN SUTD!**

# Error messages

## **Definition (what is an error message anyway?):**

Error messages are exceptions, which were caught by the Python compiler program, when it attempted to execute your code.

Typically, they are assertions, which did not pass, so that the program could execute normally.

- They usually consist of an **approximate location of where the error occurred in your code**,
- And a **standardized error message**, attempting to explain the type of error encountered.



# Error messages

```
1  # Add 1 to all numbers in list_numbers
2  list_numbers = ["0", "1", "2", "3"]
3  add_1_list = []
4  for number in list_numbers:
5      val = number + 1
6      add_1_list.append(val)
7  print(add_1_list)
```

-----

**TypeError**

Traceback (most recent call last)

**<ipython-input-8-31df80fbb176>** in **<module>**

```
3  add_1_list = []
4  for number in list_numbers:
----> 5      val = number + 1
6      add_1_list.append(val)
7  print(add_1_list)
```

**TypeError:** can only concatenate str (not "int") to str

# Error messages

```
1 # Add 1 to all numbers in list_numbers
2 list_numbers = ["0", "1", "2", "3"]
3 add_1_list = []
4 for number in list_numbers:
5     val = number + 1
6     add_1_list.append(val)
7 print(add_1_list)
```

**TypeError**

Traceback (most recent call last)

<ipython-input-8-31df80fbb176> in <module>

3 add\_1\_list = []

4 for number in list\_numbers:

----> 5 val = number + 1

6 add\_1\_list.append(val)

7 print(add\_1\_list)

**TypeError:** can only concatenate str (not "int") to str

# Typical error types

- **Learn more:** <https://www.tutorialsteacher.com/python/error-types-in-python>

Exception	Description
AssertionError	Raised when the assert statement fails.
AttributeError	Raised on the attribute assignment or reference fails.
EOFError	Raised when the input() function hits the end-of-file condition.
FloatingPointError	Raised when a floating point operation fails.
GeneratorExit	Raised when a generator's close() method is called.
ImportError	Raised when the imported module is not found.
IndexError	Raised when the index of a sequence is out of range.
KeyError	Raised when a key is not found in a dictionary.
KeyboardInterrupt	Raised when the user hits the interrupt key (Ctrl+c or delete).
MemoryError	Raised when an operation runs out of memory.

# Debugging

**Definition (Debugging):** **Debugging** is the **art/process** of detecting and removing existing and potential errors (also called as 'bugs') in a code that can cause it to behave unexpectedly or crash.

To **debug** a program, the developer (you) has to

1. start with a problematic piece of code,
2. isolate the source of the problem,
3. and then propose a fix for it.

# Debugging: Isolating a problem

## How to isolate the problem

- Python will try its best to provide a location of where the error occurred in its error message.
- Unfortunately, it is not always good info.
- It is often a good idea to try and pinpoint the location of the error, with some prints, to control which parts of the program work fine and which do not.
- Feel free to comment/uncomment parts of the code to see if it affects the apparition of said error as well!

# Debugging: 1&2. isolating a problem

```
1 # Add 1 to all numbers in list_numbers
2 list_numbers = ["0", "1", "2", "3"]
3 add_1_list = []
4 for number in list_numbers:
5     print("Ok")
6     val = number + 1
7     print("Not ok")
8     add_1_list.append(val)
9 print(add_1_list)
```

Ok

**TypeError**

Traceback (most recent call last)

<ipython-input-9-c336059098f9> in <module>

```
4 for number in list_numbers:
5     print("Ok")
----> 6     val = number + 1
7     print("Not ok")
8     add_1_list.append(val)
```

**TypeError:** can only concatenate str (not "int") to str

# Debugging: 3. Fixing the problem

- The error message indicates that Python was not able to sum a number (int), with a block of text (string).

**TypeError:** can only concatenate str (not "int") to str

- Why did this occur?
- And how do I fix it?

# Debugging: 3. Fixing the problem

```
1  # Add 1 to all numbers in list_numbers
2  list_numbers = ["0", "1", "2", "3"]
3  add_1_list = []
4  for number in list_numbers:
5      print("Ok")
6      val = number + 1
7      print("Not ok")
8      add_1_list.append(val)
9  print(add_1_list)
```

Contains strings of  
digits, not number  
type!

Ok

-----  
**TypeError**

Traceback (most recent call last)

<ipython-input-9-c336059098f9> in <module>

```
4  for number in list_numbers:
5      print("Ok")
----> 6      val = number + 1
7      print("Not ok")
8      add_1_list.append(val)
```

**TypeError:** can only concatenate str (not "int") to str



## Debugging: 3. Fixing the problem

- **Proposed fix (one of many):** convert to **int/float** before summing, convert back to **str** at the end.


```
1  # Add 1 to all numbers in list_numbers
2  list_numbers = ["0", "1", "2", "3"]
3  add_1_list = []
4  for number in list_numbers:
5      val = int(number) + 1
6      add_1_list.append(str(val))
7  print(add_1_list)
```

```
['1', '2', '3', '4']
```



# Let us practice some debugging!

Open the “Debugging practice” notebook for some  
typical examples of bugs and debugging!



## Conclusion (Chapter 10)

- What is **testing** in programming?
- What is a **unit test**?
- What is the **assert** keyword? How is it related to **error messages**?
- What is therefore the **right philosophy for programming and testing**?
- What is an **integration test**?
- What is **debugging**?
- What are **good practices in debugging**?