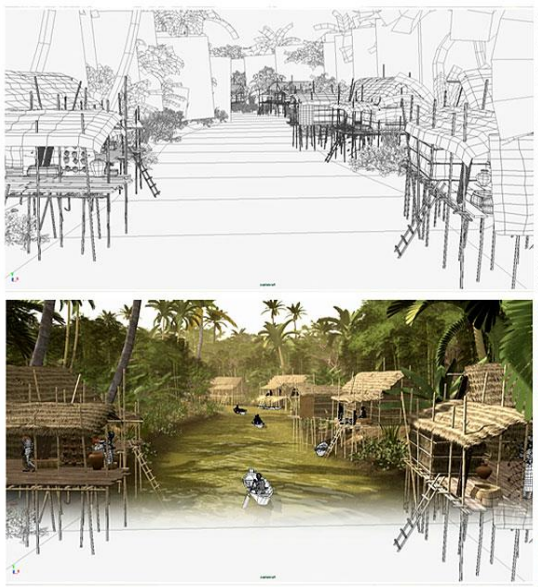
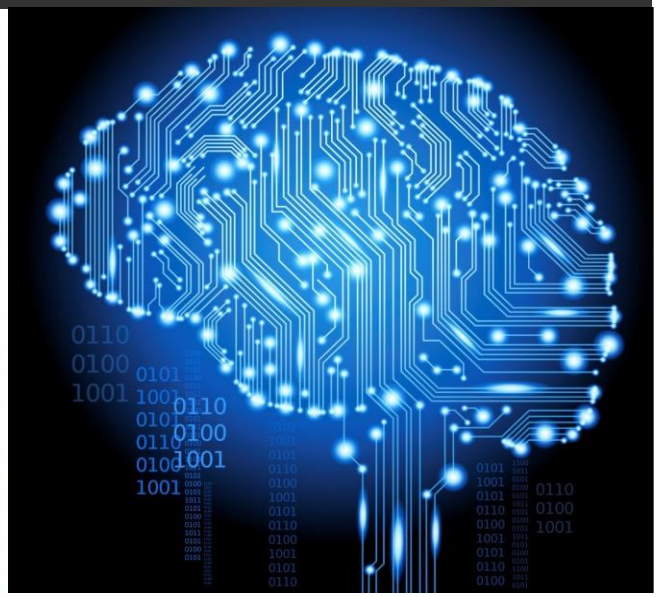
MONASH University

Information Technology

FIT1008 & 2085 Lecture 12 Exceptions, Assertions, Unit tests

Prepared by: M. Garcia de la Banda, Pierre Le Bodic





Where are we at?

- **We are now familiar with Python basics**
- **Have learnt how to implement in Python:**
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
- **Have learnt about time complexity**
- **Have started to become accustomed to think about:**
  - The use of invariants for improving our code
  - The properties of our algorithms (e.g., stable? incremental?)
  - Their Big O complexity

Objectives for this lecture

- **To learn about exception handling**
  - Be able to handle and raise them
- **To learn about assertions**
  - Inferring assertions from preconditions and postconditions
- **To learn about unit testing**
  - When should we write them? Always!
  - How to write them?

Exceptions

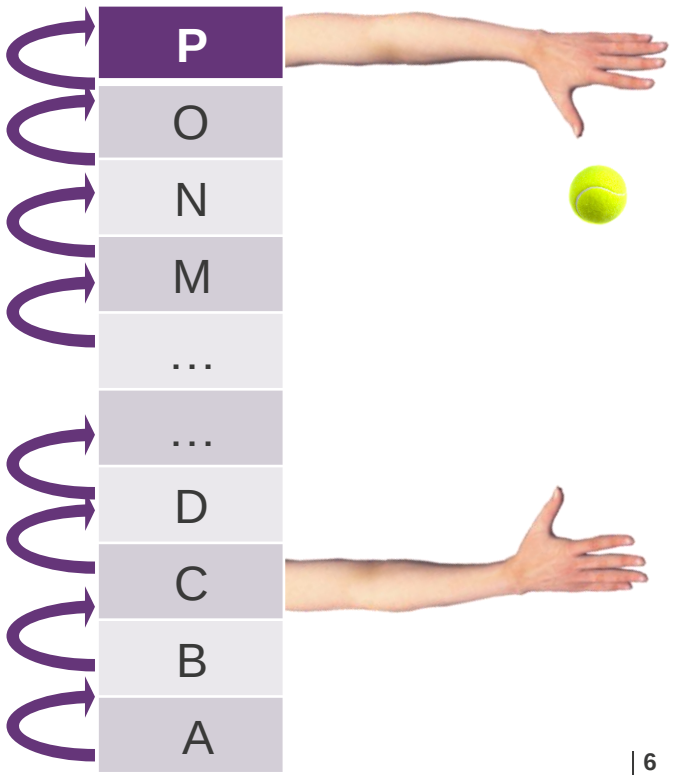
Dealing with errors (not bugs, errors)

- **There are two main situations where we need to deal with errors:**
  - When reading from input (i.e., from a file, the screen, etc)
  - When a precondition is not met
    - Called "defensive programming" (a MUST in the real world)
- **What do we do if an error is detected?**
  - Before: you have printed error messages
  - This might be OK when reading user input
    - It lets the user know what happened
  - BUT, what about the code that called the function?
  - How does it get to know something went wrong?
- **Modern languages use exception-handling**
  - Also called "catch/throw"

Note: assertions are implemented using exceptions, but assertions are not executed in optimised mode

Exception Handling

- **Exception: run-time event that breaks the normal flow of execution**
- **Exception handler: block of code that can recover from the event**
- **Exception handling: mechanism to transfer control to a handler**
- **You can see this as a big building:**
  - Each level represents a block of code
  - Block A calls B, which calls C, etc
  - Assume block P detects an error
  - P "throws" (or raises) an exception
    - Throws a "ball"
  - O might want to "catch it" (handle it)
  - If O doesn't, then N; if not M, if not …
    - Assume C does



Exception handling (cont)

- **Function C might be able to resolve the issue and continue**
- **Or, it might try and not be able:**
  - It will raise (throw) an exception itself
  - B or A might be able to "catch it"
- **If not, execution will be aborted and Python will say something like**

```
>>> x = [1,2,3]
>>> x[7]=0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
>>>
```

This indicates an exception that no-one has handled

Two parts in the last line: the type of error (**IndexError**), and what is about (after the ":")

Note, this exception is checking a precondition: the index must be between 0 and len-1

Exception handling in Python

- **Consider function `int(string)`:**
  - Returns and integer if the input string represents an integer
  - Otherwise, it raises (throws) exception **`ValueError`**
- **How to handle exceptions? Lets see with an example**
- **Consider a loop to read an integer provided by the user:**

```
def read_a_number():
    try:
        x = int(input("Please enter a number: "))
        print("Thanks! ")
    except ValueError:
        print("Not a valid number.")
```

Try clause

Except clause

`try` and **`except`** are keywords

Exception handling in Python (cont)

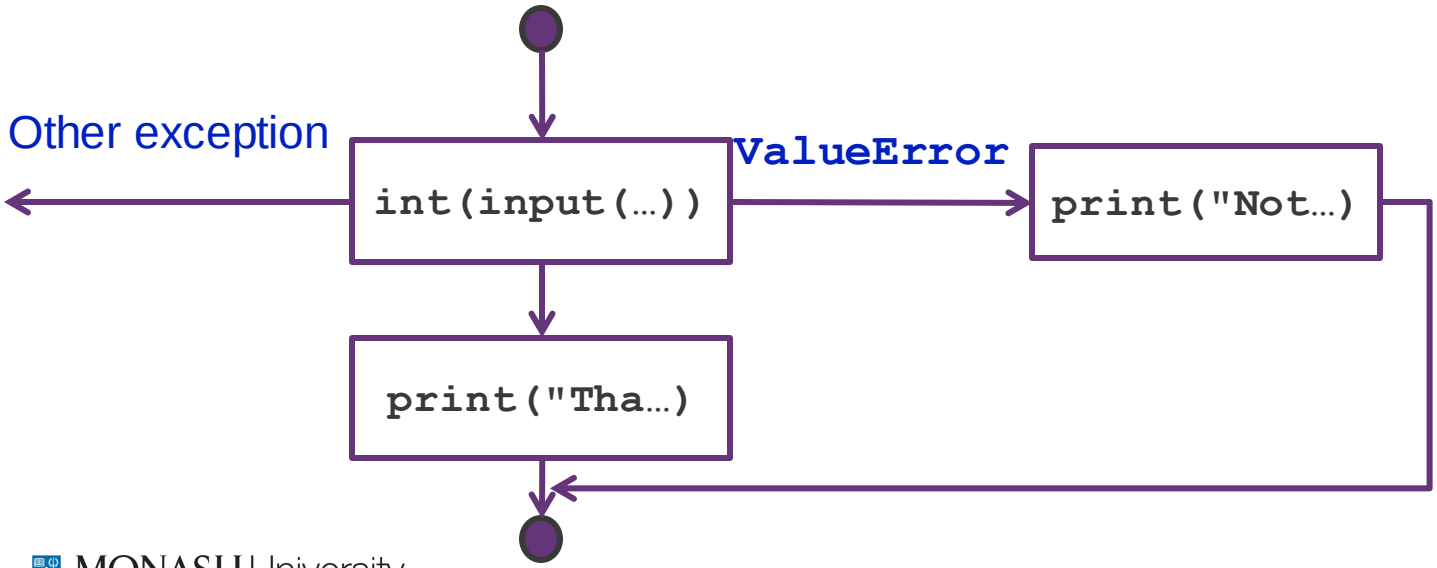
- **How does this work in terms of control flow?**
- **First, execute the try clause**
- **If no exception inside the try: skip the except clause and continue**
- **If exception, skip the rest of the try clause and:**
  - If its type matches the exception named after the **`except`**
    - The **`except`** clause is executed
    - Execution continues after the **`try`** statement.
  - If its type does not match:
    - Control is is passed on to outer **`try`** statements
    - If no handler is found, it is an unhandled exception and execution stops with a message

```
def read_a_number():
    try:
        x = int(input("Please enter a number: "))
        print("Thanks! ")
    except ValueError:
        print("Not a valid number.")
```

Exception handling in Python (cont)

- **How does our example work in terms of control flow?**

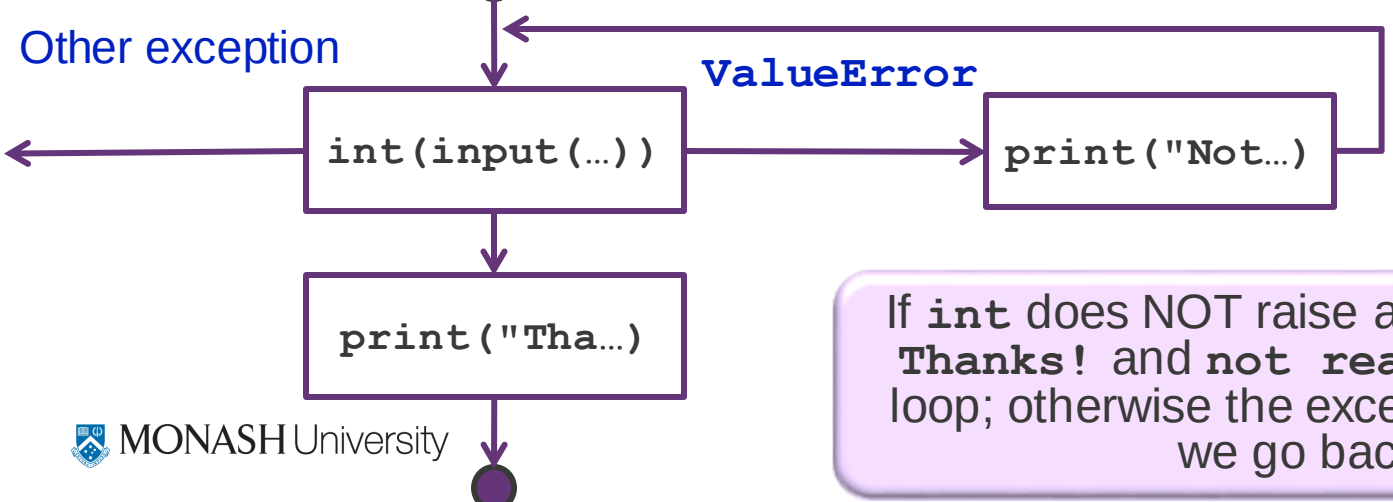
```
def read_a_number():
    try:
        x = int(input("Please enter a number: ")
        print("Thanks! ")
    except ValueError:
        print("Not a valid number.")
```



Another exception handling example

- **And how does this example work in terms of control flow?**

```
def read_a_number():
    read = False
    while not read:
        try:
            x = int(input("Please enter a number: "))
            print("Thanks! ")
            read = True
        except ValueError:
            print("Not a valid number. Try again...")
```



If `int` does NOT raise an exception, we print `Thanks!` and `not read` gets us out of the loop; otherwise the exception is handled and we go back again

Raising exceptions

- **We can raise our own exceptions:**

```
def get_height():
    h = int(input("Please enter your height(cms): "))
    if h < 0:
        raise ValueError("User gave invalid height")
    return h
```

- **The `raise` keyword gets us out of normal execution:**
  - To its caller and so on, until it finds a handler for `ValueError`
- **`ValueError` is a built-in exception type**
  - There are many others
- **You can also create your own**
  - We will see how once you learn about classes

Common exception types

Exception	Explanation
`KeyboardInterrupt`	Raised when Ctrl-C is hit
`OverflowError`	Raised when floating point gets too large
`ZeroDivisor`	Raised when there is a divide by 0
`IOError`	Raised when I/O operation fails
`IndexError`	Raised when index is outside the valid range
`NameError`	Raised when attempting to evaluate an unassigned variable
`TypeError`	Raised when an operation is applied to an object of the wrong type
`ValueError`	Raised when operation or function has an argument with an incorrect value.

User-defined Exceptions

```
class MyError(Exception):
    pass
```

```
raise MyError("Example message")
```

```
---------------------------------------------------------------------------
MyError                                   Traceback (most recent call last)
<ipython-input-17-8b7e2e51c5c2> in <module>()
----> 1 raise MyError("Example message")

MyError: Example message
```

Assertions

How do assertions work?

This assertion succeeds, hence it does not affect the flow of the program:

```
print("Before Assertion")
assert True
print("After Assertion")
```

```
Before Assertion
After Assertion
```

This assertion fails, it breaks the flow of the program:

```
print("Before Assertion")
assert False
print("After Assertion")
```

```
Before Assertion

---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-13-64567645104e> in <module>()
      1 print("Before Assertion")
----> 2 assert False
      3 print("After Assertion")

AssertionError:
```

What can we test with assertions?

Assertions can test any logical statement:

```
assert 2 < 1
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-15-c82711d5fe4d> in <module>()
----> 1 assert 2 < 1

AssertionError:
```

```
assert "derp".find("a") is not -1
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-23-09397f6700d9> in <module>()
----> 1 assert "derp".find("a") is not -1

AssertionError:
```

Assertions with error messages

Assertions can output a message to help debugging:

```
assert 2 < 1, "Learn Maths!"
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-24-156fb4004c86> in <module>()
----> 1 assert 2 < 1, "Learn Maths!"

AssertionError: Learn Maths!
```

Checking preconditions with assertions

Suppose we have the function below. Its pre-and postconditions are documented, but not checked or enforced.

```
def rounding(x):
    """Takes a non-negative real number as input,
    rounds it to the closest integer and returns it"""
    y = (int)(x+.5)
    return y
```

How would we add assertions to the function *rounding()* above, to check that the preconditions and postconditions are met, i.e. that the input is what is expected?

Checking preconditions with assertions

```
def rounding(x):
    """Takes a non-negative real number as input,
    rounds it to the closest integer and returns it"""
    assert x >= 0, "the input x should be non-negative"
    y = (int)(x+.5)
    return y
```

```
rounding(-1)
```

```
---------------------------------------------------------------------
------
AssertionError                            Traceback (most recent call
last)
<ipython-input-32-d682e6a55509> in <module>()
----> 1 rounding(-1)

<ipython-input-30-d31a8af58301> in rounding(x)
      2     """Takes a non-negative real number as input,
      3     rounds it to the closest integer and returns it"""
----> 4     assert x >= 0, "the input x should be non-negative"
      5     y = (int)(x+.5)
      6     return y

AssertionError: the input x should be non-negative
```

Checking postconditions with assertions

How would we modify the example above to check that what the correctness of what is returned?

```
def rounding(x):
    """Takes a non-negative real number as input,
    rounds it to the closest integer and returns it"""
    assert x >= 0, "the input x should be non-negative"
    y = (int)(x+.5)
    assert y >= 0, "error computing y: y should be non-negative"
    assert abs(x-y) <= 0.5, "error computing y: \
    y should be the closest integer to x"
    return y
```

Unit Testing

A systematic method to detect errors

- **Unit Testing**
  - Test each unit of code separately. (typically: 1 unit = 1 function)
- **Why?**
  - Increase confidence in code working as expected
  - Make “refactoring” easier... coding is an ongoing process
  - Found a bug? write a unit test for it... it will never appear again
- **How?**
  - In this unit, we will use Python's module *unittest*. See: https://docs.python.org/3/library/unittest.html

Unit testing example

```
import unittest
import math

class TestRounding(unittest.TestCase):
    def test_finite_rounding(self):
        self.assertEqual(rounding(0), 0)
        self.assertEqual(rounding(5), 5)
        self.assertEqual(rounding(5.1), 5)
        self.assertEqual(rounding(5.4), 5)
        self.assertEqual(rounding(5.5), 6)
        self.assertEqual(rounding(5.7), 6)
```

```
testtorun = TestRounding()
suite = unittest.TestLoader().loadTestsFromModule(testtorun)
unittest.TextTestRunner().run(suite)
```

```
.
----------------------------------------------------------------------
-
Ran 1 test in 0.005s

OK
```

```
<unittest.runner.TextTestResult run=1 errors=0 failures=0>
```

Unit testing example: we forgot a case!

Note that the *rounding()* function we wrote can raise an Exception if the input is infinite!

```
import math
rounding(math.inf)
```

```
---------------------------------------------------------------------
------
OverflowError                                Traceback (most recent call
last)
<ipython-input-56-ca5786ed7d40> in <module>()
      1 import math
----> 2 rounding(math.inf)

<ipython-input-43-55df3e52f1b9> in rounding(x)
      3     rounds it to the closest integer and returns it"""
      4     assert x >= 0, "the input x should be non-negative"
----> 5     y = (int)(x+.5)
      6     assert y >= 0, "error computing y: y should be non-negati
ve"
      7     assert abs(x-y) <= 0.5, "error computing y:     y should
be the closest integer to x"

OverflowError: cannot convert float infinity to integer
```

Unit testing example: we forgot a case!

This should at the very least be documented in the function:

```
def rounding(x):
    """Takes a finite non-negative real number as input,
    rounds it to the closest integer and returns it.
    If the input is infinite, raises an OverflowError."""
    assert x >= 0, "the input x should be non-negative"
    y = (int)(x+.5)
    assert y >= 0, "error computing y: y should be non-negative"
    assert abs(x-y) <= 0.5, "error computing y: \
    y should be the closest integer to x"
    return y
```

Unit testing example: we forgot a case!

And this should also be added as a test to the existing unit tests:

```
def test_infinite_rounding(self):
    with self.assertRaises(OverflowError):
        rounding(math.inf)

TestRounding.test_infinite_rounding = test_infinite_rounding
```

```
suite = unittest.TestLoader().loadTestsFromModule(testtorun)
unittest.TextTestRunner().run(suite)
```

```
..
----------------------------------------------------------------------
-
Ran 2 tests in 0.008s

OK
```

```
<unittest.runner.TextTestResult run=2 errors=0 failures=0>
```