MONASH University

Information Technology

FIT1008/2085
Exceptions and Assertions

Prepared by:
Maria Garcia de la Banda and Pierre Le Bodic

Objectives for this lesson

- **To learn about preconditions and postconditions**
  - To be able to determine preconditions and postconditions for our code
- **To learn about exception handling and when to use them**
  - To be able to handle and raise them when appropriate
- **To learn about assertions and when to use them**
  - To be able to write assertions when appropriate

Preconditions and Postconditions

- **Precondition: properties of the data that must be true before a piece of code (function, loop, etc) is executed**
  - If violated (i.e., if not true), the result of the execution is undefined (unknown)
  - **Examples**: the list to be processed is not empty; the input list for a function is already sorted, the elements of a list are comparable (for sorting!)
- **Postcondition: properties of the data or I/O, that results from the code being executed**
  - If violated, there is a bug in the code
  - **Examples**: the returned/printed value is positive; the input list is now sorted
- **Part of "Design by Contract" technique between the caller and the callee**
  - Precondition must be met by caller, postcondition by the callee
- **For more info: see document in Moodle (under Week 4)**

MONASH University

Exceptions

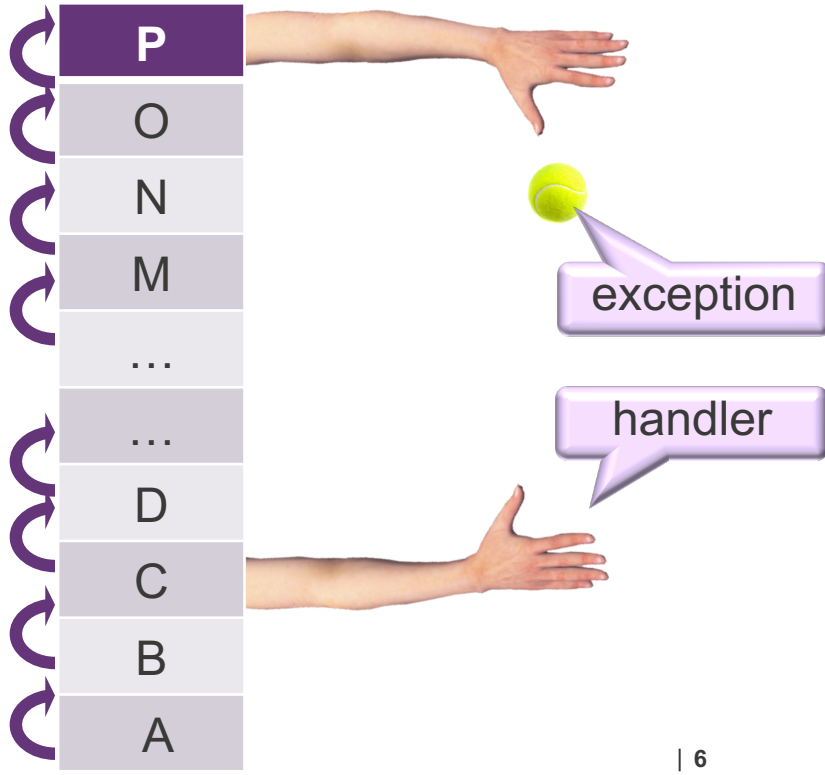
Dealing with errors (not bugs, errors from code users)

- **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 for an exported function/method is not met
- **Called "defensive programming"**
  - An absolute MUST in the real world for robust code (can survive the unexpected)
- **What do we do if an error is detected?**
  - Before: you have printed error messages
  - This might be OK when reading human user input
    - It lets the user know what happened and allows them to provide a correct input
  - 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")**

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

Looks like… the system stack!



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 aborts 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 message indicates an exception that no-one handled

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

Note that this exception was thrown by the code for accessing an element after 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: "))
    except ValueError:
        print("Not a valid number.")
    else:
        print("Thanks! ")
```

Try clause

Except clause

Else clause

`try` and **`except`** are keywords, as **`else`** is

Exception handling in Python (cont)

The aim of the else clause is to keep the try clause as short as possible. This reduces bugs created by different exceptions being raised by code in the try clause

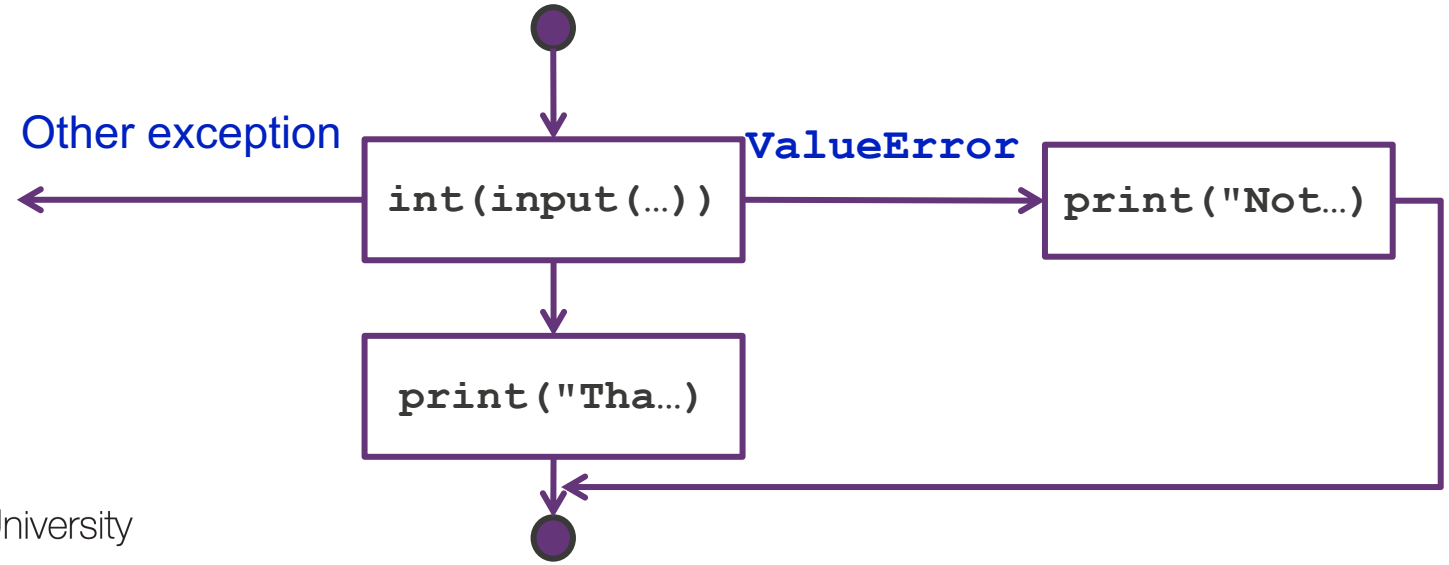
- **How does this work in terms of control flow?**
- **First, execute the try clause**
- **If no exception inside the try: go directly to the else clause**
- **If exception, skip the rest of the try clause and:**
  - If its type matches the exception named after the **`except`**
    - The **`except`** clause is executed
    - Execution continues after the end of the **`else`** clause (in this case, it returns)
  - If its type does not match:
    - Control is given 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: "))
    except ValueError:
        print("Not a valid number.")
    else:
        print("Thanks! ")
```

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: ")
    except ValueError:
        print("Not a valid number.")
    else:
       print("Thanks! ")
```



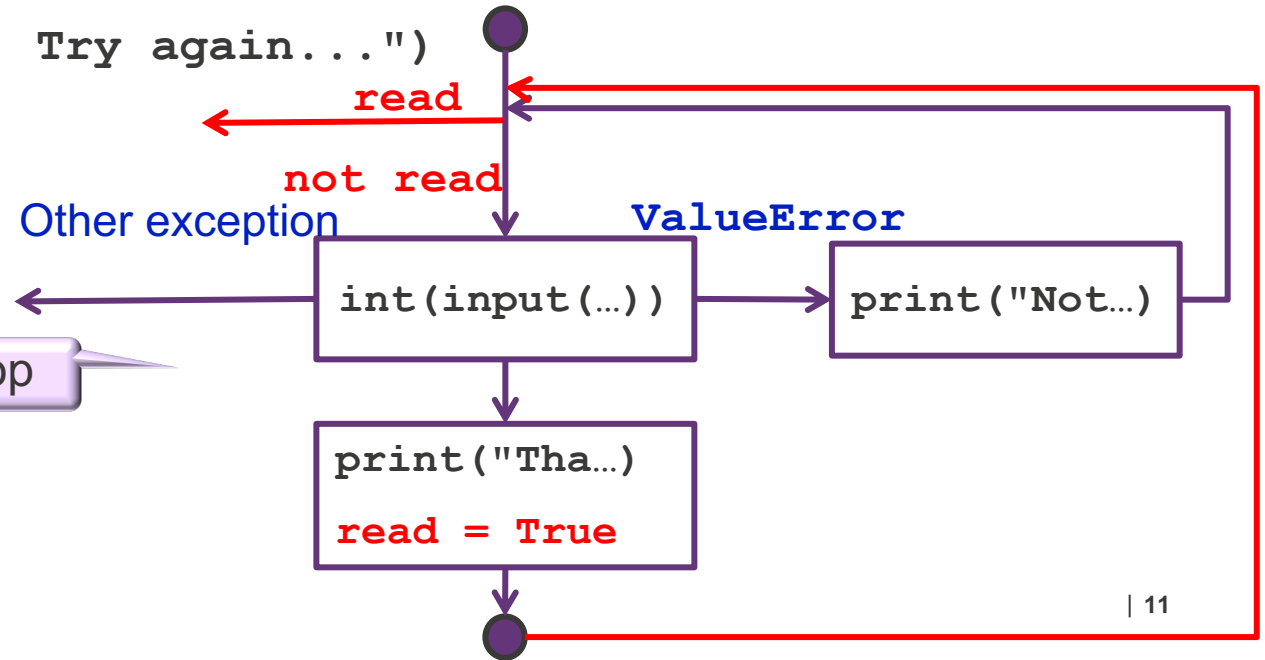
Another exception handling example

- **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: "))
        except ValueError:
            print("Not a valid number. Try again...")
        else:
            print("Thanks! ")
            read = True
```

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

Works exactly the same, but inside a loop



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 (see https://docs.python.org/3/library/exceptions.html)
  - Often called with a string as argument indicating the detailed cause of the error

Common built-in 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.

Handling different exceptions in different ways

- **Several except clauses might be required to handle things differently:**

```
def average(a_list):
    try:
        result = sum(a_list)/len(a_list)
    except ZeroDivisionError:
        return None
    except Exception:
        raise
    else:
        return result
```

first matching except clause is triggered.

- `print(average([1,2,3])` **does not raise an exception; it returns what?**
  - 2.0
- `print(average([]))` **does raise an exception; it returns what?**
  - **None**
- `print(average("hello")` **does raise another exception; it returns what?**
  - Nothing, it raises an exception that is not caught

User-defined Exceptions

- **You can also create your own**

```
>>> class MyError(Exception):
...     pass
...
>>> raise MyError("Example Message")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyError: Example Message
>>>
```

This will not be completely clear until we learn about classes

For now, if you want to define one, just copy the class code with the name you want instead of MyError

- **We will come back to this once we go over classes**

MONASH
University

Assertions

Dealing with bugs (not errors from users, but from us)

- **Useful to check the internal state of a program is as expected**
- **In other words: used as a sanity check to avoid bugs**
- **The format is as follows:** `assert Expression[, Arguments]`
- **The `Expression` is first evaluated:**
  - If true, execution continues normally
  - If false, an exception is raised with the given `Arguments`

The exception raised has the type `AssertionError`

- **Examples:**

```
assert age >= 0, "Age cannot be negative!”
assert type(id) is IntType, "id is not an integer"
```

- **Assertions can be turned off for efficiency (-O option to compiler)**
  - Once the program is correct and in production…

How do assertions work?

- If the assertion succeeds, it does not affect the rest of the program

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

- If it fails, it breaks the flow of the program

```
>>> print("Before Assertion"); assert False; print("After Assertion")
Before Assertion
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
>>>
```

What can we test with assertions?

- **Any logical statement. Examples:**

Without arguments

```
>>> assert 2 < 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
>>> assert 2 < 1, "Learn maths!"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: Learn maths!
>>> x = [2,3]; y = [3,2]; assert x == y, "Are not equivalent"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: Are not equivalent
>>>
```

Assertions versus Exceptions (see later)

- **Exceptions are used for ensuring program robustness:**
  - Any data (read or passed as argument) is within expected values (preconditions)
  - Any kind of abnormal condition
- **Assertions are used for ensuring program correctness:**
  - Internal state is as expected (debugging)
  - Checking invariants (other than in user input) and “cant’ happen situations”
  - Ensuring return values are what they should, and all postconditions hold
- **In general: code cannot recover from bugs (assertions), but can from user errors (exceptions). In practice, not so crystal clear…**
  - For example, sometimes checking the precondition is too expensive (list is sorted)

Checking preconditions

No need to check the types; Python does it when in debug mode for you

- **The function:**

```
def rounding(x: float) -> int:
    """ Computes the integer closest to x.
    :pre: x is >= 0
    """
    return int(x + 0.5)
```

does not check its precondition (x>=0).

- **How should we modify the code in our unit to ensure it does?**
  - If it is an exported function: raise exceptions
  - Otherwise: use assertions (you are your own user, so it is a bug)
  - Preconditions that are expensive to compute (e.g., sorted) will not be checked

Checking preconditions(cont)

- **If rounding is not exported:**

```
def rounding(x: float) -> int:
    """ Computes the integer closest to x.
    :pre: x is >= 0
    """
    assert x >= 0, "Argument must be >= 0"
    return int(x + 0.5)
```

- **If it is:**

```
def rounding(x: float) -> int:
    """ Computes the integer closest to x.
    :pre: x is >= 0
    """
    if x < 0:
        raise Exception(”Argument must be >=0")
    return int(x + 0.5)
```

Summary

- You can now determine the preconditions and postconditions of your code
- You are able to handle and raise exceptions when appropriate
- You know when and how to use assertions