Running time ratio

Algorithm	50	100	500	1,000	5,000	50,000
Selection	1.34	2.13	8.04	13.31	59.07	537.42
Insertion	1.08	2.02	6.15	11.35	51.86	474.29
Merge	7.58	7.64	6.93	6.87	6.35	6.27
Quicksort (pivot - last item)	1.02	1.63	6.09	11.51	52.02	475.34
Quicksort Hybrid (n < 22 do insertion)	0.55	0.56	0.60	0.66	0.62	0.66

Source: "Algorithms Unlocked" by Thomas Cormen

Prepared by Julian García, based on material by
David Albrecht and María García de la Banda

Lecture 14
Assertions, Exceptions, Testing

a.k.a, when things go wrong

Computer Science

MONASH University
Information Technology

Dealing with errors

- There are two main situations where we need to deal with errors:
  - When a **precondition** is **not met.** Called "defensive programming" (a MUST in the real world)
  - When **reading** from **input** (i.e., from a file, the screen, etc)

- What do we do if an error is detected?
  - In **FIT1040**: you have **printed error messages**
  - This *might be* OK when interacting with a human.
  - BUT, **what about** the **code** that called the function? How does it get to know something went wrong?

- Modern languages use **exception-handling**.

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

Blocks of code.

Block A calls Block B.
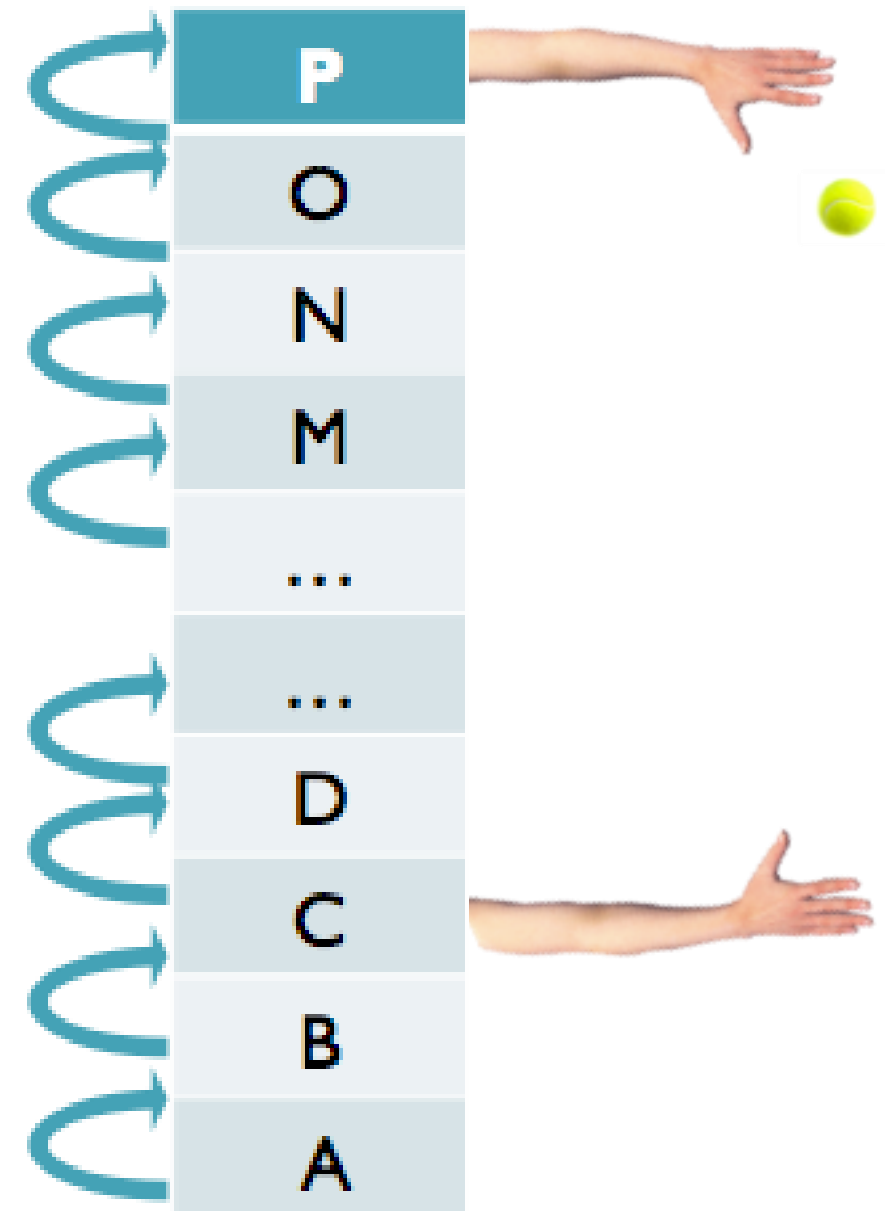Block B calls Block C.
Block C calls Block D….

P detects an error

P throws (raises) an exception

O might want to catch it.

If O doesn't, then N might, if not M …

Assume C catches the error…

P
O
N
M
…
…
D
C
B
A

Exception handling

- **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

Example Exception

```
>>> x = [1,2,3]
>>> x[7]=0
```

Example Exception

```
>>> 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
>>>
```

An **Exception** that no-one has handled.

name

message

Exception handling in Python

- Consider a loop to read an integer provided by the user:

```python
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?

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

- 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 statemen**t.
  - If its **type does not match**:
    - **The next except clause is checked**
    - **Or if none remain, jump to outer try statements**
    - If **no handler is found**, it is an unhandled exception and execution stops with a message

Exception handling in Python (cont)

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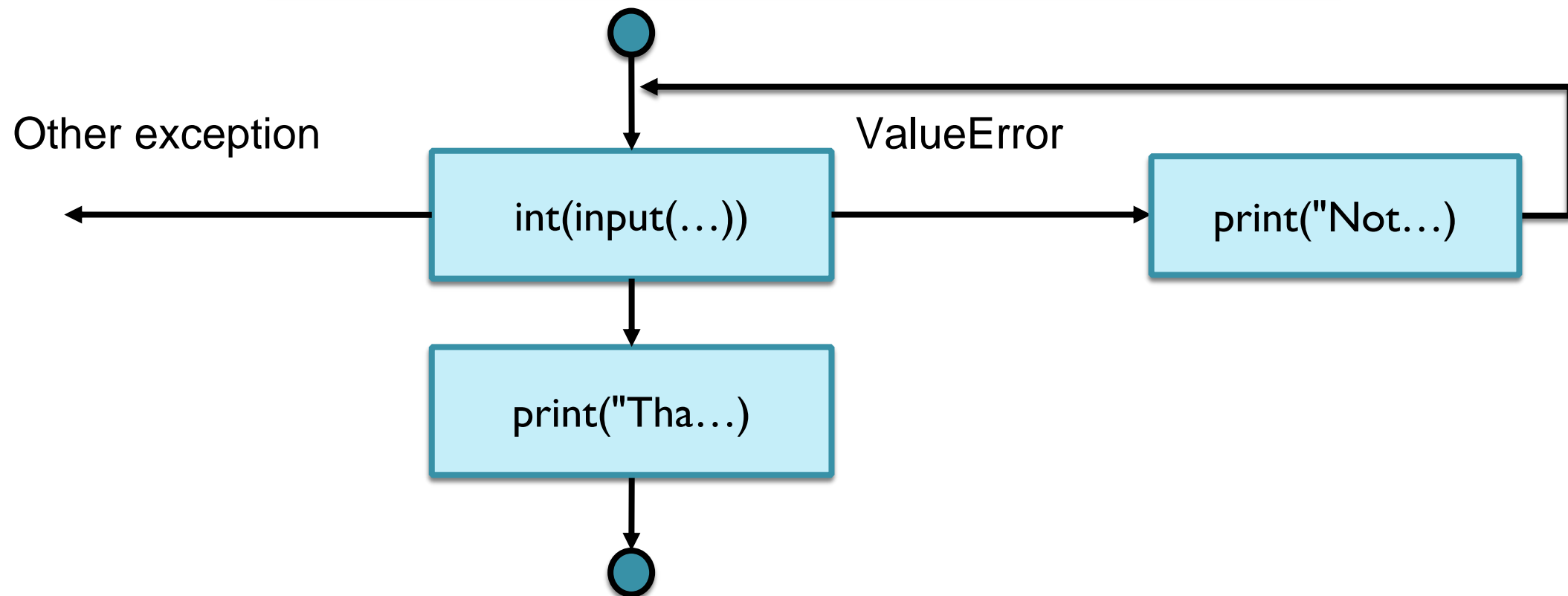
Other exception

ValueError

int(input(…))

print("Not…)

print("Thanks! ")
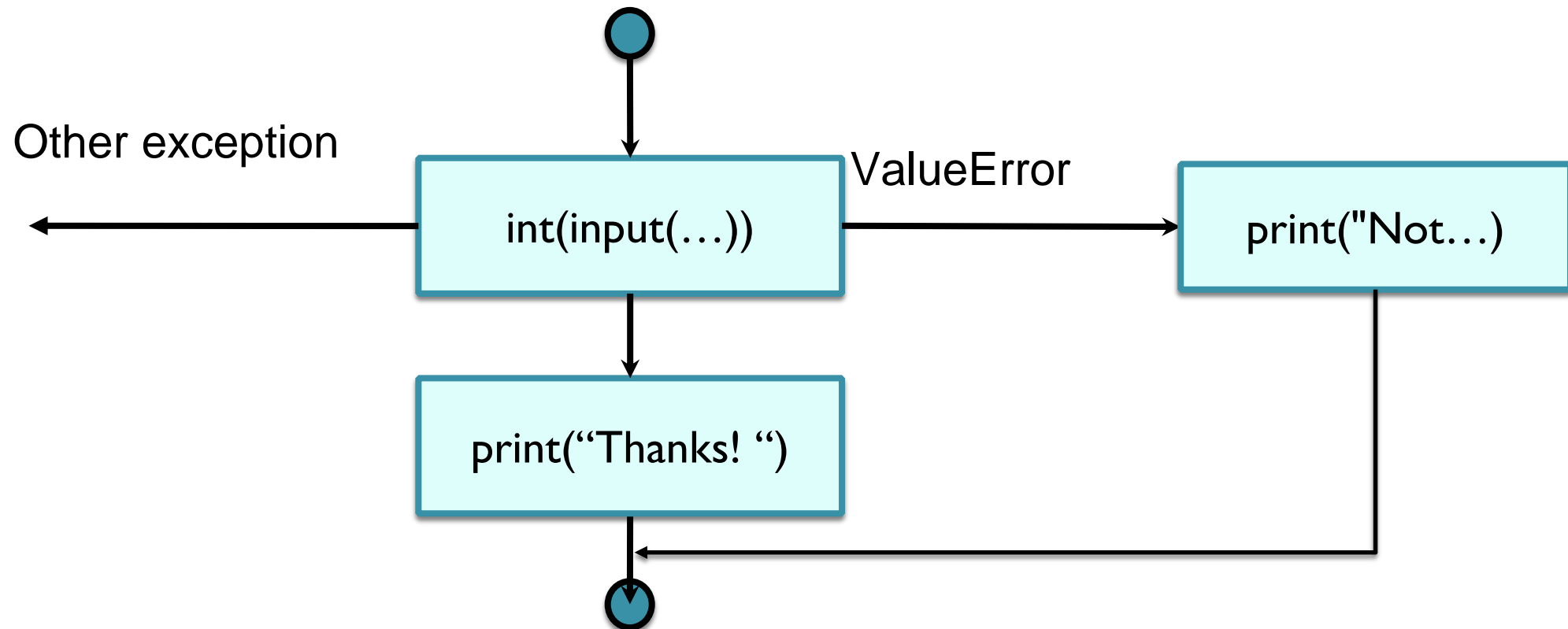
Another example

```python
while True:
    try:
        x = int(input("Please enter a number: "))
        print("Thanks! ")
        break
    except ValueError:
        print("Not a valid number. Try again...")
```

Other exception

ValueError

int(input(…))

print("Not…)

print("Tha…)

Raising exceptions

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

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

- The `raise` keyword gets us out of normal execution. Caller takes control, 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.

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.

Multiple Exceptions

Can you (error) handle this?

```python
def badThingsHappen(nameOfFile):
    file = open(nameOfFile)
    output=""
    chars=0
    CountLines=0
    for line in file:
        output+=line
        chars+=len(line)
        CountLines+=1
    charsPerLine = chars/CountLines
    print(str(charsPerLine)+ statement)
    return output


print(badThingsHappen("filename.txt"))
```

User defined Exception

```
class MyError(Exception):

    pass
```

```
>>> raise MyError('test message')
Traceback (most recent call last):
    File ``<pyshell#247>'', line 1 in <module>
        raise MyError('test message')
MyError: test message
```

what can go wrong?

```python
def power(x, n):
    value = 1
    if n > 0:
        value = power(x, n//2)
        if n % 2 == 0:
            value = value*value
        else:
            value = value*value*x
    return value
```

```python
def power(x, n):
    if n < 0:
        raise ValueError("n should be a positive Integer")
    value = 1
    if n > 0:
        value = power(x, n//2)
        if n % 2 == 0:
            value = value*value
        else:
            value = value*value*x
    return value
```

```
Traceback (most recent call last):
  File "/Users/juliangarcia/Dropbox/monash/FIT1008/FIT1008-2016
    print(power(2.5, -1))
  File "/Users/juliangarcia/Dropbox/monash/FIT1008/FIT1008-2016
    raise ValueError("n should be a positive Integer")
ValueError: n should be a positive Integer

Process finished with exit code 1
```

```python
def power(x, n):
    assert n > 0, "n should be a positive Integer"
    value = 1
    if n > 0:
        value = power(x, n//2)
        if n % 2 == 0:
            value = value*value
        else:
            value = value*value*x
    return value
```

```
/Users/juliangarcia/anaconda/bin/python /Users/juliangarcia
Traceback (most recent call last):
  File "/Users/juliangarcia/Dropbox/monash/FIT1008/FIT1008-
    print(power(2.5, -1))
  File "/Users/juliangarcia/Dropbox/monash/FIT1008/FIT1008-
    assert n > 0, "n should be a positive Integer"
AssertionError: n should be a positive Integer
```

Assert statement

Useful for checking preconditions.

```python
def power(x, n):
    assert n > 0, "n should be a positive Integer"
    value = 1
    if n > 0:
        value = power(x, n//2)
        if n % 2 == 0:
            value = value*value
        else:
            value = value*value*x
    return value
```

Assert statement

Useful for checking preconditions.

asserts can be
turned off

Useful message

```
assert n > 0, "n should be a positive Integer"
value = 1
if n > 0:
    value = power(x, n//2)
```

Logical expression that should be **True**

Assert is a keyword.

If the logical expression evaluates to **False**
an exception is Raised
`AssertionError: Size should be positive`

assert | əˈsəːt |

verb [reporting verb]
state a fact or belief confidently and forcefully: [with clause] : *the company asserts that the cuts will not affect development* | [with obj.] : *he asserted his innocence.*

Precondition: condition or predicate that <u>must always be true</u> just prior to the execution of some section of code.

the effect of the code becomes undefined if the precondition does not hold… it may or may not carry out its intended work.

CONTRACT

From now on… we will require assertions to handle all declared preconditions in lab.

A systematic approach to "revealing" errors….
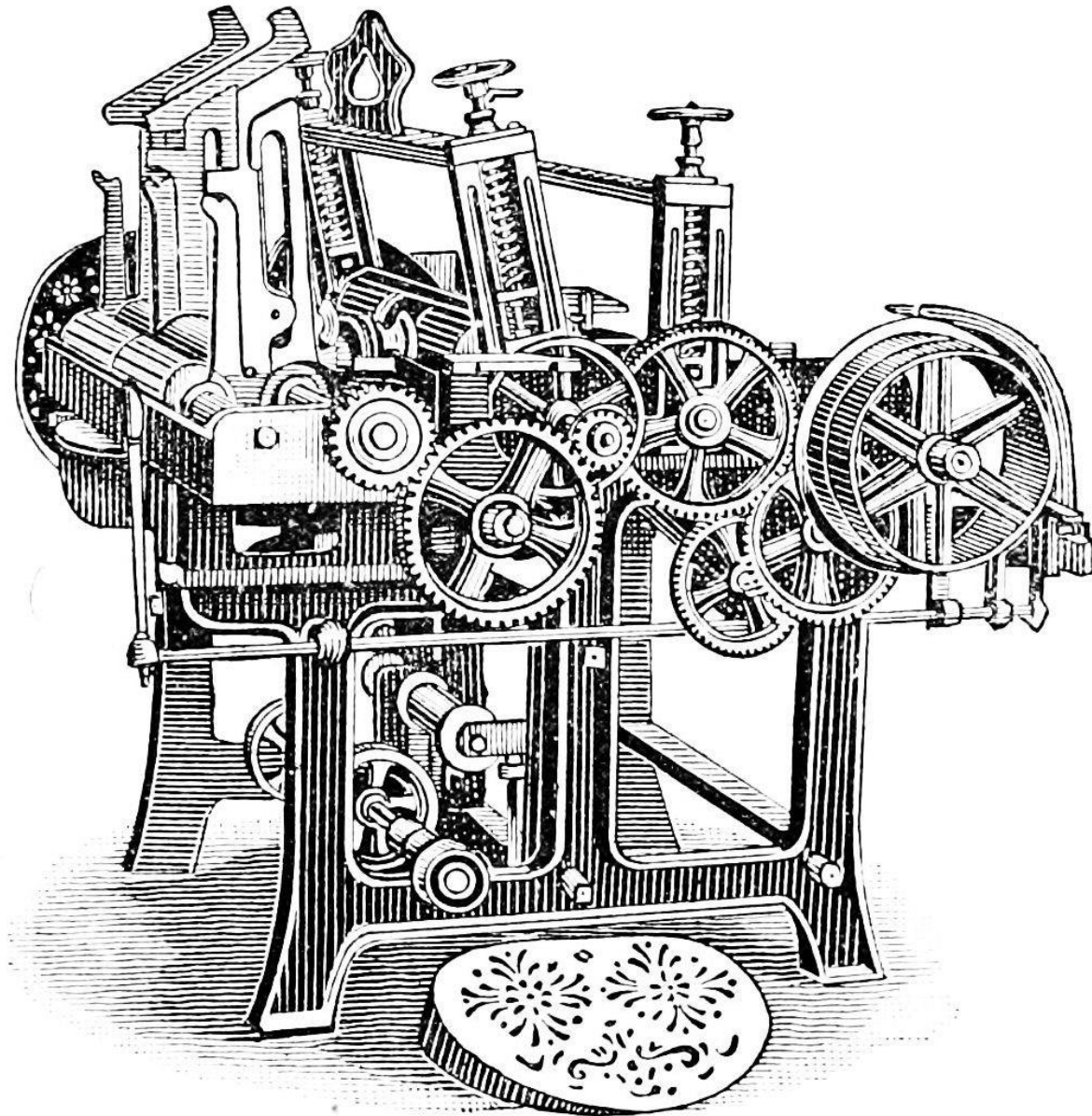
Testing the machine

Testing each component

A systematic approach to "revealing" 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 FIT1008)

1. For each module, create a separate file: *test_power.py*
2. For each unit to be tested, create a function: def *test_power()*

    i) Set up testing conditions   ii) Call method

    iii) Verify output (assert)   iv) Clean up (if necessary)

```python
def power(x, n):
    assert n >= 0, "n should be a positive Integer"
    value = 1
    if n > 0:
        value = power(x, n//2)
        if n % 2 == 0:
            value = value*value
        else:
            value = value*value*x
    return value
```
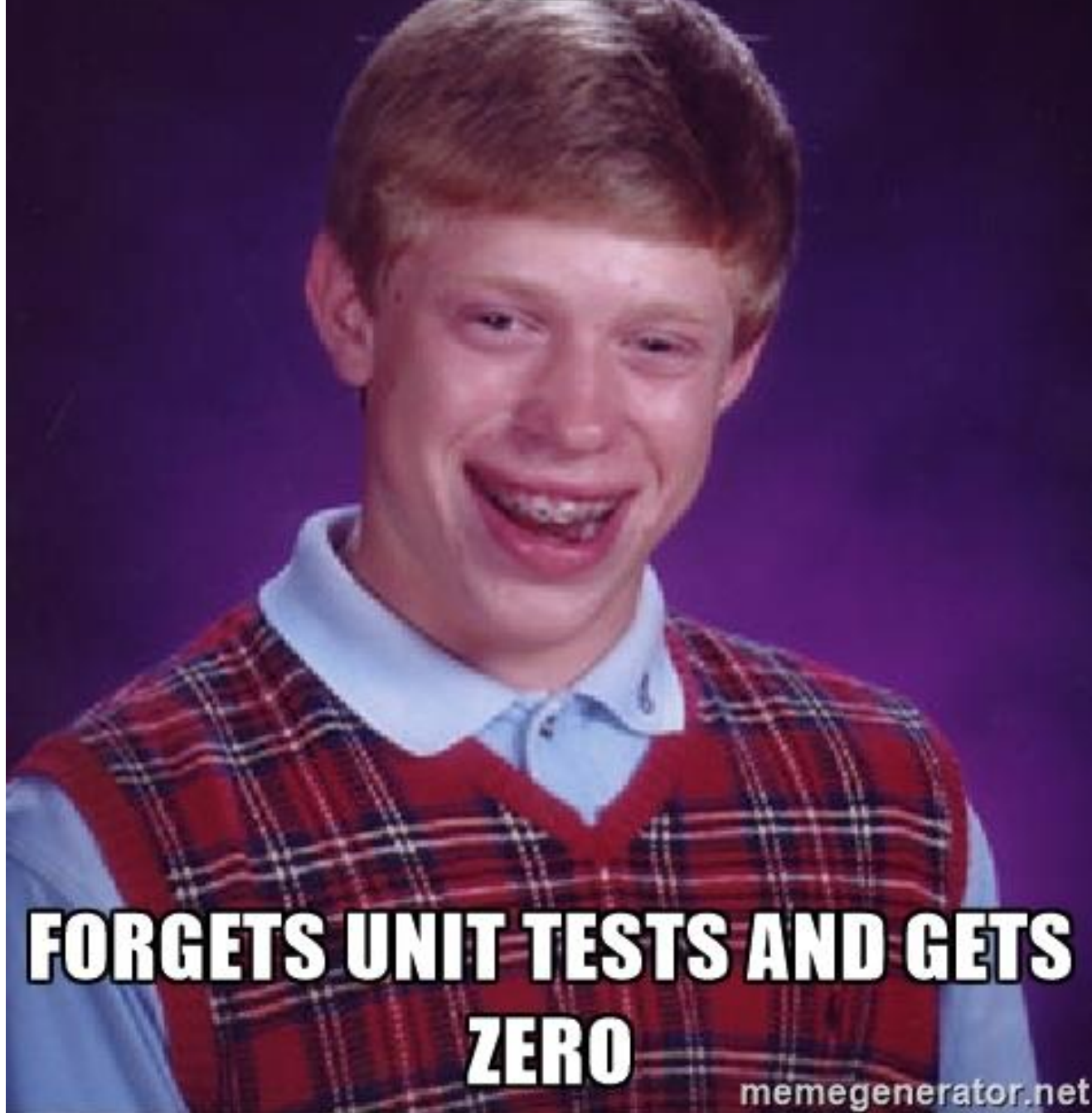
power.py

```python
def test_power():
    assert power(1, 0) == 1**0, "Test failed: 1^0 != 1^0"
    assert power(2, 2) == 4, "Test failed: 2^2 != 4"
    assert power(2.5, 4) == 2.5**4, "Test failed: 2.5^4 != 4"


if __name__ == '__main__':
    test_power()
    print("All tests passing...")
```

test_power.py

From now on… we will require all functionality to be tested.

What if you don't?

SPENDS THREE HOURS WRITING UP A PRAC

FORGETS UNIT TESTS AND GETS ZERO

memegenerator.net

Disclaimer

- This is a simple way to do testing…

- This is a large topic in itself… you will learn more about it in other units

- Python has a unit testing framework…
  you are not required to use it, but you are welcome to if you want to…

Summary

- Exceptions

- Assertions - Preconditions

- Tests