Programming through Python

823G5 / 990G5

Errors and Testing

Dr Benjamin Evans

b.d.evans@sussex.ac.uk

Announcements

- Mid-Module Survey
- Live coding lecture and catchup practical

Mid-Module Survey

You said…	We will provide…
“More live coding examples”	A live coding lecture!
“More structure in the labs for beginners”	Introductions and tips to begin the practicals
“More of a sense of community / engagement”	Pair programming exercises

Live coding lecture and catchup practical

- Next week I will use the lecture to write solution(s) to Week 6 (Maze solving) lab exercises. This will:
  - Give live worked examples (as requested!)
  - Draw various concepts and data structures together
  - Explain the thought process behind the code to help develop the “programmer’s mindset”
  - Demonstrate the debugger
- The practical will be for catching up on previous weeks, reworking previous solutions with OOP or working on your assessments

Recap

- More Data Structures that you can use to refine your program design and coding.
  - Lists
  - Sets
  - Dictionaries

  } *Mutable*

  - Tuples — *Immutable*
- These structures are *objects*, so have their own *methods*.
- More on comprehensions.
- Nesting data structures.

The `list`

- The **`list`** is one of Python's *sequence* types (`range` is another) and so is "**iterable**".
- Lists are **mutable** sequences (we can change their values as we like "in place").
- Lists store **ordered** collections (sequences) of items e.g.,
  - exam grades throughout a module
  - animals entering an arc!
- The type does not need to be *homogenous* (the same across a list), but would an *inhomogeneous* collection really be meaningful?

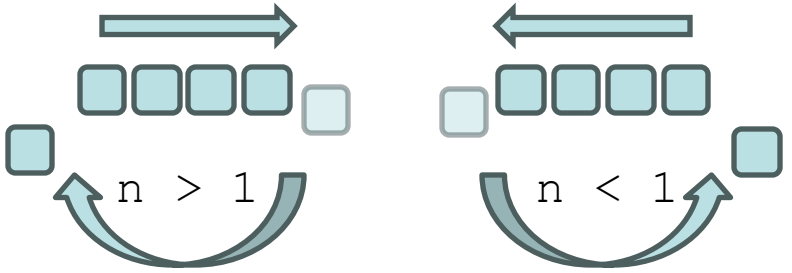
List as a Queue?

- Lists work but are not efficient for this purpose.
  - While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow
  - This is because all the other elements must be shifted by one
- To implement a queue more efficiently, use collections.deque (**d**ouble-**e**nded **que**ue)
  - designed to have fast `append`s and `pop`s from both ends…
  - i.e., additionally, it has `popleft()` (equivalent to `pop(0)`), `appendleft()`, and `extendleft()`

Efficient Queues

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

- `deque` can act as a **circular buffer** with **`rotate`**`(n=1)`, shifting items in a loop
- `n > 0` → rotate `n` steps **right**.
  - i.e., for `n=1`: `d.appendleft(d.pop())`
- `n < 0` → rotate `n` steps **left**.
  - i.e., for `n=-1`: `d.append(d.popleft())`



The `set`

- A `set` is an **unordered** collection that contains **no duplicate elements** e.g., like a hand of playing cards.
- Converting a list (which can contain duplicates) to a set is a useful way of getting its unique items: `set(my_list)`.
- Standard maths set operations available.
- Key notation for containers:
  - `list`s: initialise: `[ ]` square backets (brackets); index: `[ ]`
  - `tuple`s: `( )` round brackets (parentheses); index: `[ ]`
  - `set`s: `{ }` curly brackets (braces); index: N/A!

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Difference

Union

Intersection

Symmetric difference (Xor)

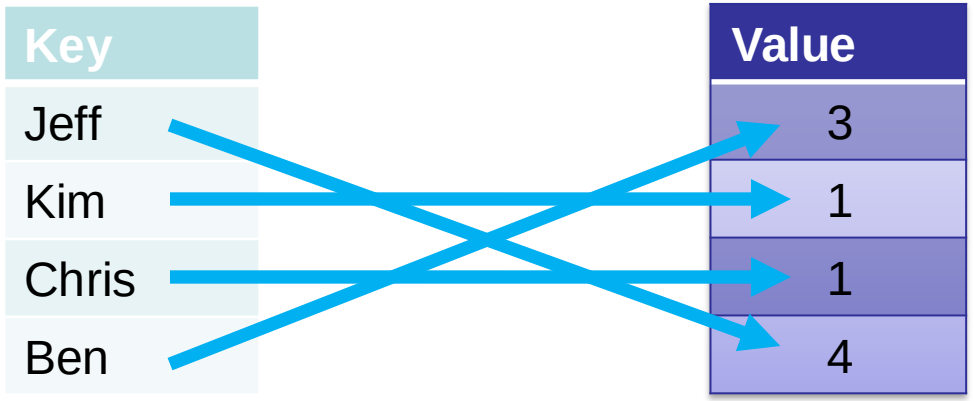
`a <= b`
`a.issubset(b)`

`a >= b`
`a.issuperset(b)`

Dictionary

- A Dictionary (`dict`) is an example of a mapping type.
- Consists of a set of (key, value) pairs (effectively 2-tuples).
- Typically, the key value is used to recover some value.
  - E.g., A language dictionary where you can use a word (the key) and look up its meaning (the value)

- **Keys are unique** but values can be repeated (many-to-one mapping is possible).




```
my_dict = {"Jeff": 4, "Kim": 1, "Chris": 1, "Ben": 3}
```

e.g., coffee consumption for each lecturer…

Basic Dictionary Operations

- Definition

```
my_dict = {
    "Jeff": 4,
    "Kim": 1,
    "Chris": 1,
    "Ben": 3}
```

- Look-up

```
my_dict["Chris"]  # 1
```

- Check for an entry

```
"Ben" in my_dict  # True
```

- Adding a new entry

```
my_dict["Ian"] = 5
```

- Updating an entry

```
my_dict["Jeff"] = 2
```

- Deleting an entry

```
del my_dict["Kim"]
```

- Get a list of keys

```
keys = list(my_dict.keys())
keys = list(my_dict)
```

- Get a list of values

```
vals = list(my_dict.values())
```

- Get a list of (key, value) tuples

```
entries = list(my_dict.items())
```

The `tuple`

- `tuple`s are **immutable** and so useful for protecting data

```
coordinates = (-4.5, 7, 2.5)
```

- Can be accessed by:
  - "unpacking" e.g.: `x, y, z = coordinates`
  - "indexing" e.g.: `z = coordinates[2]`
- When unpacking, you must have the same number of variables on the LHS as there are elements in the tuple.
- Used to represent entities that have a set of data values that mean something collectively.

Copying

- Assignment statements do not copy objects – they create bindings (references) between the target and object

```
orig_list = [1, 2, 3]
new_list = orig_list
new_list[1] = 7
print(orig_list)
# [1, 7, 3]
```

- If we want independent copies of the container (and contents) we need to use the **`copy`** module function: **`deepcopy`**

```
from copy import deepcopy
new_list = deepcopy(orig_list)
new_list[1] = 7
print(orig_list)
# [1, 2, 3]
```

Other `collections`

- As we have seen, `deque` is included in the collections module: `from collections import deque`.
- Other, more advanced / efficient data structures can also be found there which may be occasionally useful, including:
  - **`Counter`**: a `dict` subclass for counting hashable objects
  - **`defaultdict`**: a `dict` subclass which supplies missing values
  - **`namedtuple`**: a `tuple` with named fields

Review

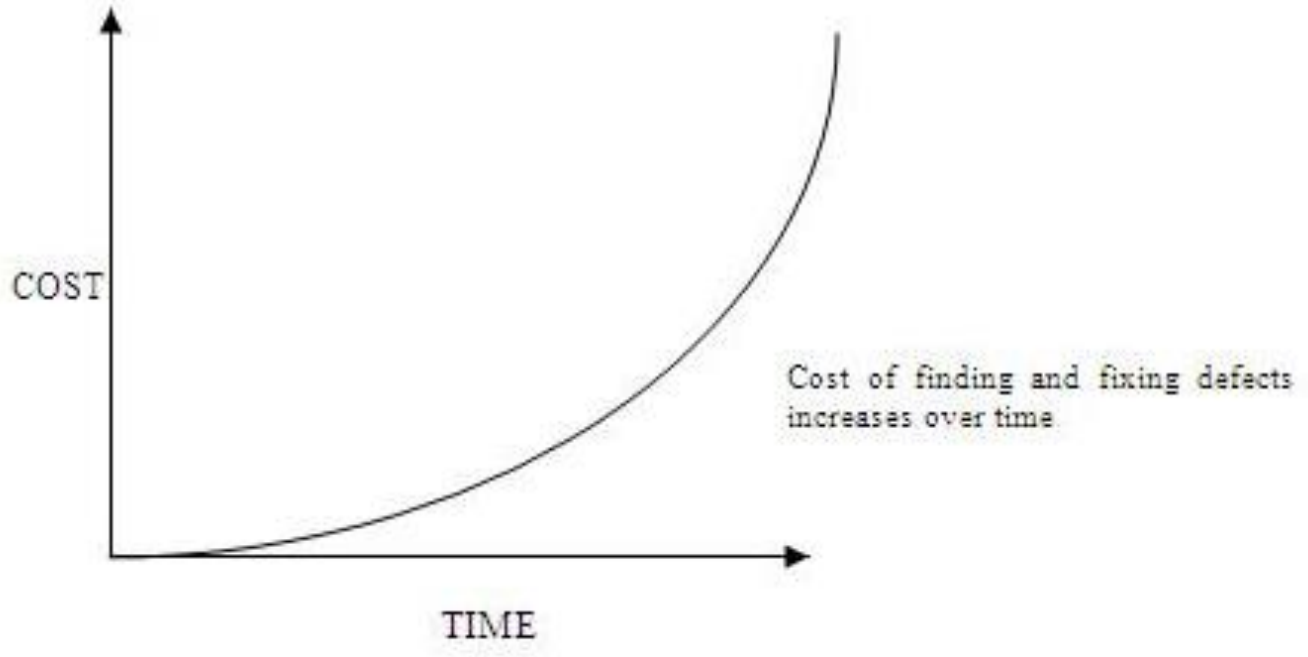
- Python support a wide range of data structures including:
  - **`list`**s: initialise: `[]`; index: `[]`
  - **`set`**s: initialise: `{}`; index: **N/A!** (but you can loop over them)
  - **`dict`**ionaries: initialise: `{}`; index: `[]`
  - **`tuple`**s: initialise: `()`; index: `[]`
- Each have methods suited to different applications.
- Choose a data structure that reflects the real-world problem that you want to model.
- Engage with the exercises and examples on Canvas.

This session

- Dealing with errors
- Using the debugger
- Exceptions
- Unit testing
- System testing

Errors

- For any significant software development, errors are a reality.
- It is neither economic nor practical to build a piece of software that is 100% perfect.



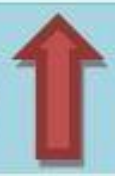
Types of error

- Building the wrong product: failure to understand the user requirements correctly.
- Building the *right product* in the *wrong way*.
- Shift in user requirements.
- Coding / syntax errors.
- Logic errors.
- Changes in hardware and software.

Build the right product in the right way

- A broader issue that we can address in this module
  - Down to good Software Engineering
  - Identifying the user requirements
  - Predicting how these requirements might evolve over time
  - Choosing the right tool sets
- Creating flexible and modular designs
  - OO design principles can help greatly here

Terminology

Error / Mistake	Defect / Bug/ Fault	Failure
Found by 	Found by 	Found by 
Developer	Tester	Customer

Coding / Syntax Errors

- The easiest category of error to deal with.
- For compiled languages like Java and C++, we can’t even run the code until it is at least free of syntax errors.
- For interpreted languages like Python and JavaScript, we only discover many such errors when we try to execute the code.
- So errors can go unnoticed for a while until we try to execute all parts of a codebase.

Coding / Syntax Errors

- But ultimately, as long as we follow the syntax rules of the language, we should be able to solve this category of error.
- But that does not guarantee that our program will do what we want…

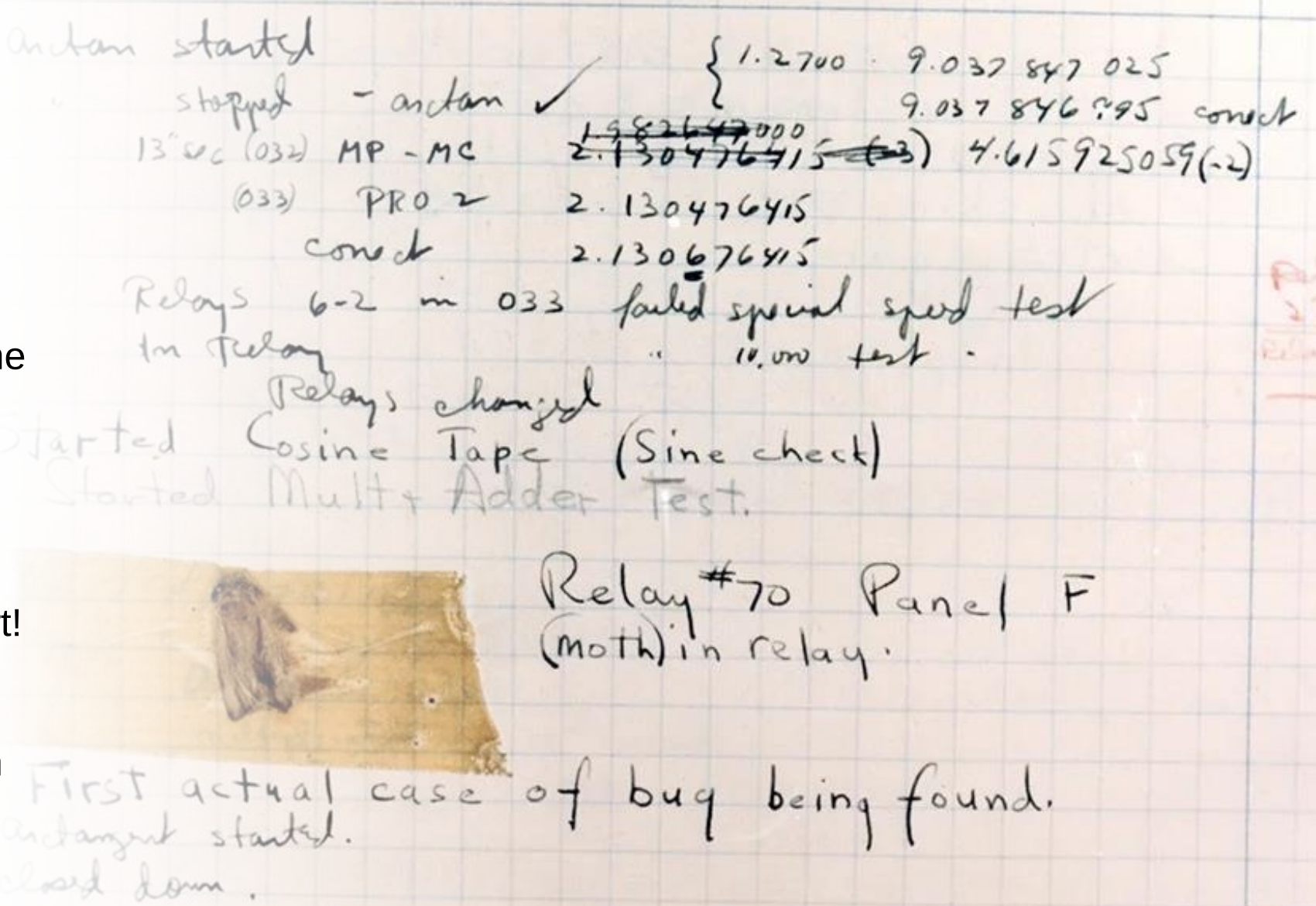
```
def loop1(x):
    x = 0
    while x < 10:
        print(x)
```

Finding bugs

- Once we have all the syntax errors sorted, we then proceed to establish that the program does what we want.
- The search for logic errors is:
  - Partly down to developers
  - Partly down to testers
- Developers typically use debuggers to figure out why code is not working the way they intended.

Bugs!

- In 1947 the team working on the Harvard Mark II computer, including pioneer Grace Hopper, found a moth in the machine causing it to malfunction
- The moth was taped to the logbook – a very literal bug report!
- Note however, contrary to popular belief, this is not the first use of the term to mean a fault in an engineering context – Edison used it in the 19th Century



Debugger

- A tool within an Integrated Development Environment (IDE) that allows you to peek inside a piece of code to see what it is doing:
  - Establish breakpoints (pause points)
  - Examine values of variables
  - Check the call stack
  - Execute code one line at a time
- Let’s see a demo of this in action…

Debugger demo

Finding bugs

- Using a debugger helps the developer to identify and fix many logic problems and get the codebase to a state where it’s worth testing.
- But the codebase will likely evolve over time, so how do we ensure that the code still works properly as we work further on it?
- How do we ensure that our code still works correctly when other developers make changes elsewhere?
- Assertions and Automated unit testing…

Assertions

- We can `assert` boolean statements:

```
assert 0 <= x < max_value
assert not isinstance(item, Statue)
assert type(a) == int, "a must be an int"
```

- If the statement is `True`, execution silently continues
- If `False`, execution is stopped and an error is raised
- Use them liberally to make guarantees about your code and catch errors early

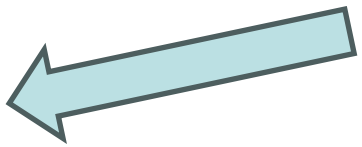
Unit testing

- A unit can be:
  - A single class (in an OO design)
  - A method or function
  - A group of functions that work together
- To provide a clear single element of functionality.
- Can group units together to develop component level testing e.g. a data I/O module or utilities module.

Unit testing

- Typically automated (Continuous Integration).
- Tests are built throughout the coding process.
- Can be run on demand to check that the code is working as expected.
- Defined by:
  - Actions
  - Inputs
  - Expected outputs
- Let’s start with a basic example…

```python
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # Check that s.split() fails with a TypeError
        # when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```



This simple block provide a function to run the tests

Key points

- Import the **`unittest`** module.
  - Test class subclasses `unittest.`**`TestCase`**
  - All test methods start with the name `test`
  - **`assertTrue`**`()` checks whether a function returns `True`
  - **`assertFalse`**`()` checks whether a function returns `False`
  - **`assertEqual`**`()` checks whether two things have the same value
  - **`assertRaises`**`()` checks whether Python raised as exception error (more on exceptions later)
- Right-click code and opt to run unit tests.

Unit testing a class

- General pattern:
  - Write a class
  - Write a unit test class to go with it
  - Unit test class instantiates class and uses test cases to check the object and its methods all work as expected
- Let’s look at an example…

A class to be tested

```
class Backpack:
    """A class to store and retrieve named items with a set capacity."""
    def __init__(self, capacity):
        self.contents = []
        self.capacity = capacity

    def add_item(self, to_add):
        """Adds an item to our backpack."""
        if len(self.contents) < self.capacity:
            self.contents.append(to_add)
            return True
        else:
            return False

    def remove_item(self, to_remove):
        """Removes an item from our backpack.
           Throws a ValueError exception if the item is not in the list.
        """
        self.contents.remove(to_remove)

    def check_item(self, to_check):
        """Returns True if item is in backpack, False otherwise"""
        return to_check in self.contents
```

Unit test class

```
import unittest
from backpack import Backpack

class TestBackPack(unittest.TestCase):

    def setUp(self):
        """Runs prior to each unit test method (optional)"""
        self.b1 = Backpack(3)

    def tearDown(self):
        """Runs after each unit test method (optional)"""
        self.b1.contents.clear()

    def test_1(self):
        """For ease here we define one unit test with lots of tests within it."""
        self.b1.add_item('book')
        self.b1.add_item('pen')
        self.assertTrue(self.b1.check_item('book'))
        self.assertTrue(self.b1.check_item('pen'))
        self.assertFalse(self.b1.check_item('balloon'))
        self.assertEqual(len(self.b1.contents), 2)
        self.assertTrue(self.b1.add_item('candle'))
        self.assertFalse(self.b1.add_item('map'))  # Capacity is 3
```

Key points

- You can have as many test cases as you like (a method starting `test` … is a test case).
- Each test case can have many actual tests.
- Together they form a test suite.
- Use of **`setUp`**`()` and **`tearDown`**`()` are optional
  - They are called after each complete test case (method)
  - *Not* after every assert
- Test runner can be facilitated by PyCharm or at the command line, e.g.: `python -m unittest module`

So how much testing?

- Good range of test cases.
- Consider edge and corner cases, as well as typical cases.
- Good coverage of all class or other code feature.
- Looking for “fit for purpose”.

Exceptions

- But sometimes we just come across a run time problem.
- Like a function that expected a numeric value but received a string.
- Exceptions provide a way of dealing with this in a controlled manner.
- Very useful for handling problems that arise due to things beyond your control e.g. bad user input, flaky internet connection, unreliable hard disk, etc.

Exceptions

- You have likely seen them in action already:

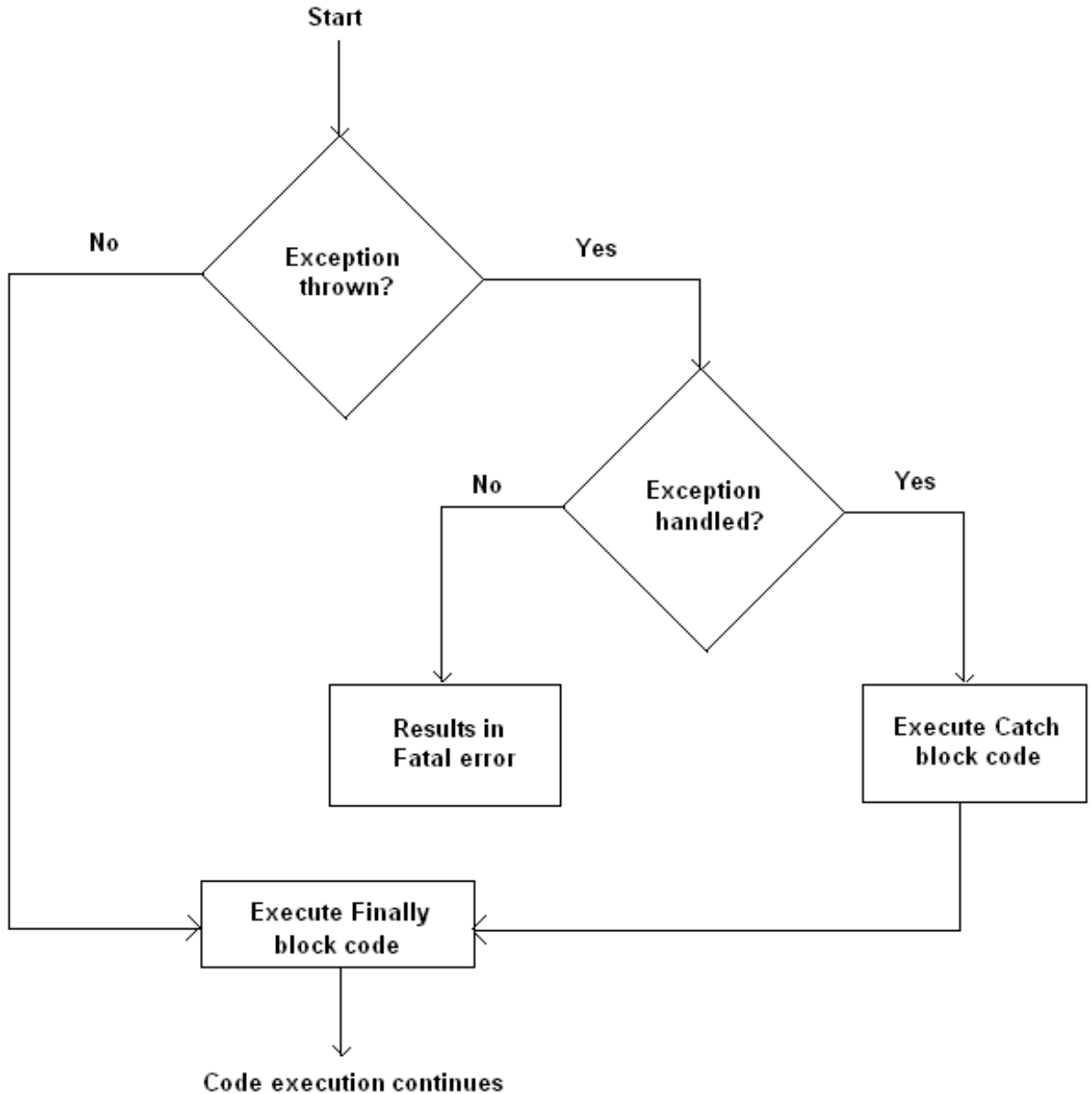
```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Exceptions

- An exception is like raising your hand and saying, “I have a problem”.
- If someone can help, they will “catch” (except) your exception.
- Allows graceful handling of errors e.g. file not found.
- Otherwise, if no-one can catch your exception then the program must terminate (with an uncaught error).
- Exceptions can be raised by the run time environment, or by `raise` in your own code.

```
# Example of catching a built in ValueError.
# ValueError occurs when an incompatible type is provided as
# as an argument for a function/method.


def fetch_value():
    global x  # Makes x global
    while True:
        try:
            x = int(input("Please enter a number: "))
            break
        except ValueError:
            print("Oops! That was no valid number. Try again...")

def main():
    fetch_value()
    print(f'Value of x: {x}')

if __name__ == '__main__':
    main()
```

```
# Example of catching a built in ValueError.
# ValueError occurs when an incompatible type is provided as
# as an argument for a function/method.


def fetch_value():
    global x  # Makes x global
    while True:
        try:
            x = int(input("Please enter a number: "))
            if x < 0:
                raise ValueError()
            break
        except ValueError:
            print("Oops! That was no valid number. Try again...")

def main():
    fetch_value()
    print(f'Value of x: {x}')

if __name__ == '__main__':
    main()
```


Start
Exception thrown?
No
Yes
Exception handled?
No
Yes
Results in Fatal error
Execute Catch block code
Execute Finally block code
Code execution continues

Exception strategy

- Add a `try` clause for code that may fail.
- If an exception is thrown, look for an `except` clause.
  - Exceptions can be the built in, or user defined (by subclassing the `Exception` class to give a more informative error).
  - Only the first exception caught is processed.
- An `else` block executes if there is no exception.
- Lastly, regardless of whether there was an exception or not, the `finally` clause is executed.

Anatomy of exception handling

```
try:
    # Some fragile code to run which may occasionally fail
except SomeError:
    # Optional block: Handle a specific exception here
except SomeOtherError:
    # Optional block: Handle another specific exception here
except:
    # Optional block: Handle any other exception(s) here (if required)
    # (It is better to except multiple specific exceptions)
else:
    # Execute this block if there are no exceptions
finally:
    # This is always executed whether there is an exception or not
```

```
def do_calc():
    global x, y, z  # Makes x, y and z global variables
    while True:
        try:
            x = int(input("Please enter first number: "))
            y = int(input("Please enter second number: "))
            if x < 0 or y < 0:
                raise ValueError()
            z = x / y
        except ValueError:
            print("Oops! Invalid inputs. Try again...")
        except ZeroDivisionError:
            print("Cannot divide by zero. Try again...")
        except:
            print("Something went wrong! :( Try again...")
        else:
            print("Well done - no exceptions raised!")
            break
        finally:
            print("Thank you for using this function today!")


do_calc()
print(f'{x} divided by {y} is {z}')
```

User defined exception

- You can define your own exception.
- Typically, by sub-classing the `Exception` class.
- By convention, defined with `Error` at the end of the name.
- Can have optional parameters, often used to explain the source of the exception.
- Or can attempt to fix the problem (if that's possible).
- https://docs.python.org/3/tutorial/errors.html

```
class Backpack2:
    def __init__(self, capacity):
        self.contents = []
        self.capacity = capacity

    def add_item(self, to_add):
        """Adds an item to our backpack."""
        if len(self.contents) < self.capacity:
            self.contents.append(to_add)
            return True
        else:
            return False

    def remove_item(self, to_remove):
        try:
            if to_remove not in self.contents:
                raise NotInBackpackError(to_remove, \
                                         'not in the backpack')
            else:
                self.contents.remove(to_remove)
        except NotInBackpackError:
            print('Handled here...')
        finally:
            print('Carrying on...')
```

```
class NotInBackpackError(Exception):
    def __init__(self, item, message):
        print(f'{item} {message}')


def main():
    b1 = Backpack2(3)
    b1.add_item('pen')
    b1.add_item('medkit')
    b1.remove_item('pen')
    b1.remove_item('sword')
```

Looking for exceptions

- When an exception is raised, we look for an exception handler...
  - In the current method
  - In the method that called that method
  - And the method that called that method…
- Until either one is found and deals with the issue.
- Or if none is found, then the program terminates :'((unchecked exception).

```
class Backpack2:
    def __init__(self, capacity):
        self.contents = []
        self.capacity = capacity

    def add_item(self, to_add):
        """Adds an item to our backpack."""
        if len(self.contents) < self.capacity:
            self.contents.append(to_add)
            return True
        else:
            return False

    def remove_item(self, to_remove):
        """”Removes an item from our backpack."""
        if to_remove not in self.contents:
            raise NotInBackpackError(to_remove, \
                                     'not in the backpack')
        else:
            self.contents.remove(to_remove)

    def check_item(self, to_check):
        """Returns True if item is in backpack, False otherwise."""
        return to_check in self.contents
```

```
class NotInBackpackError(Exception):
    def __init__(self, item, message):
        print(f'{item} {message}')


def main():
    b1 = Backpack2(3)
    b1.add_item('pen')

    try:
        b1.remove_item('sword')
    except NotInBackpackError:
        print('Exception handled here...')
```

System level testing

- Once we think all the different units and component parts of our software are working, it’s time to try the program as a whole.
- That’s system level testing.
- Harder (though not impossible) to automate.
- Trickier where Graphic User Interfaces are integrated.

System level testing

- Often conducted using a documentation-based approach.
- Working from a list of user or system requirements.
- Checking that the software does what was intended on an end-to-end basis…

System level testing

Test Ref	Description	Inputs	Expected Outputs	Actual Outputs	Pass/Fail
1	User enters a name and password into login form	Username (String) and password (String) where both are valid	Progress through login screen	Progressed through login screen	PASS
2	User enters a name and password into login form	Username only and password is blank	Error message and return to login screen	Progressed through login screen	FAIL
…	….	…	…	…	…

Review

- Understand the impact of errors at different stages of the Software Development Lifecycle.
- Make sure you know how to use a **debugger** to help you fix problems.
- Create **unit tests** for key software components.
- Consider **system testing** strategy.
- Engage with the exercises on Canvas.