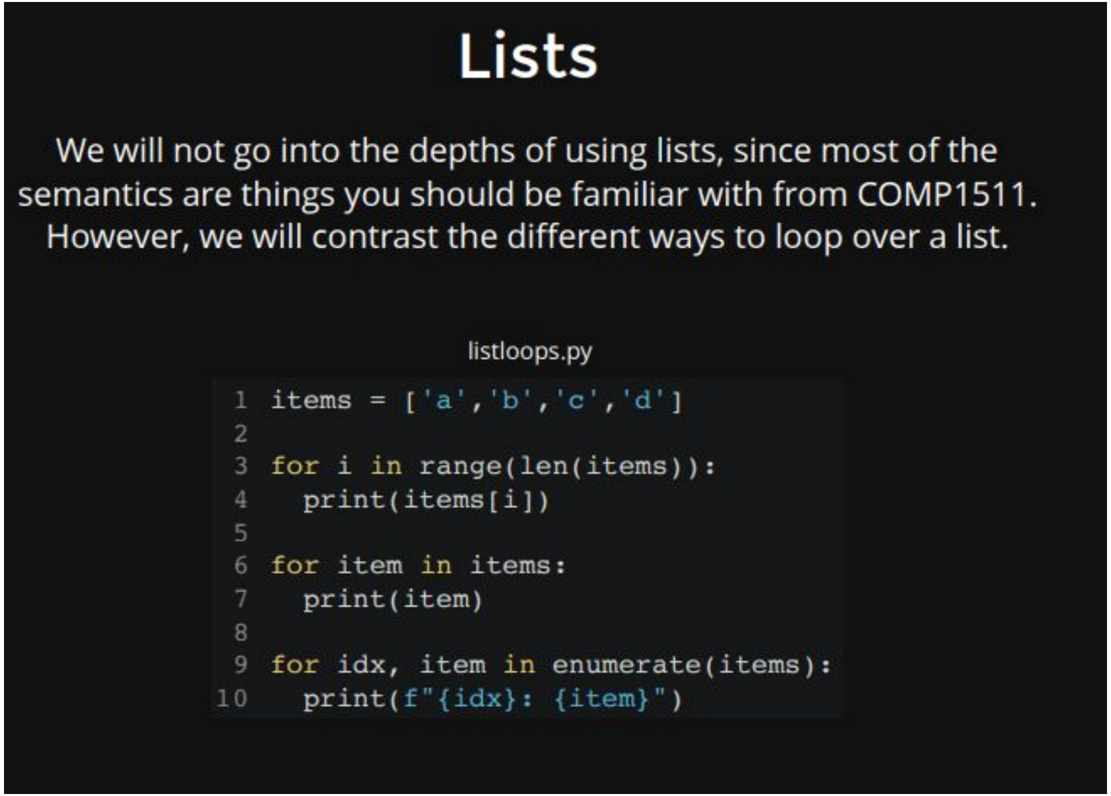
Week 03

Lecture Summaries



Lists

We will not go into the depths of using lists, since most of the semantics are things you should be familiar with from COMP1511. However, we will contrast the different ways to loop over a list.

listloops.py

```python
items = ['a','b','c','d']

for i in range(len(items)):
  print(items[i])

for item in items:
  print(item)

for idx, item in enumerate(items):
  print(f"{idx}: {item}")
```

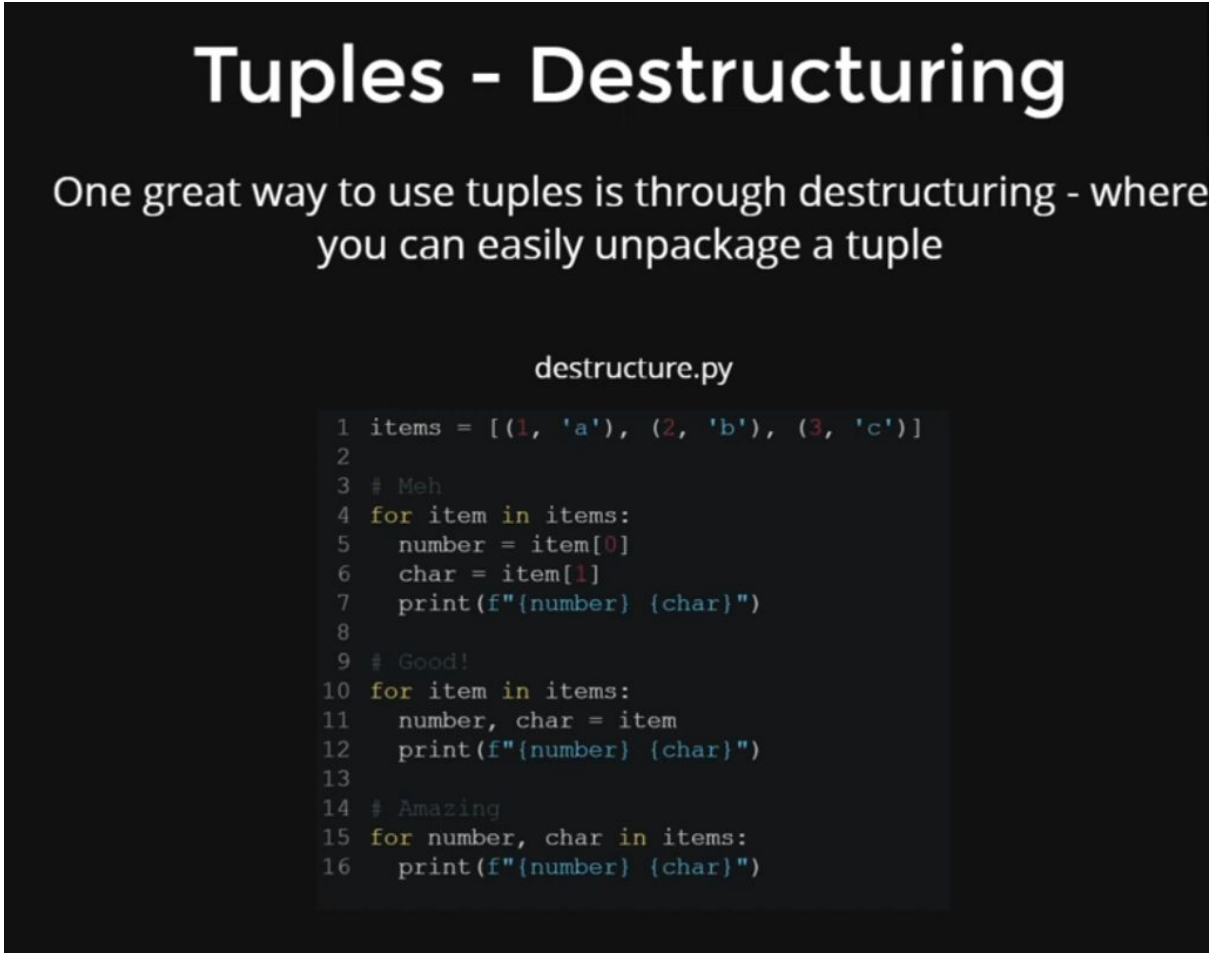
Tuples

Syntactically, tuples are similar to lists except:

1. They are read-only once created
2. You create them with () instead of []

In terms of usage, tuples tend to be used more often where:

1. You have a collection of different types
2. The collection is of fixed size

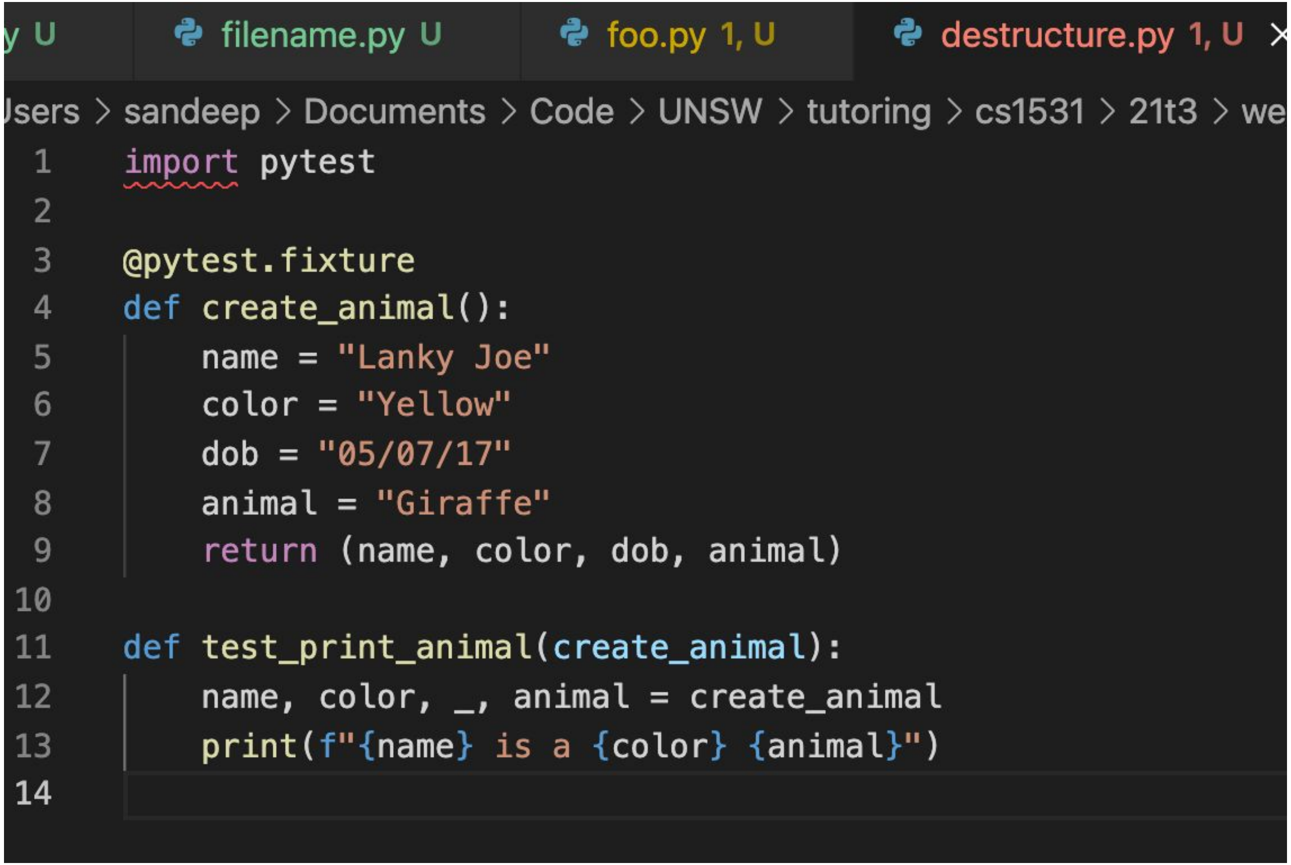


Tuples - Destructuring

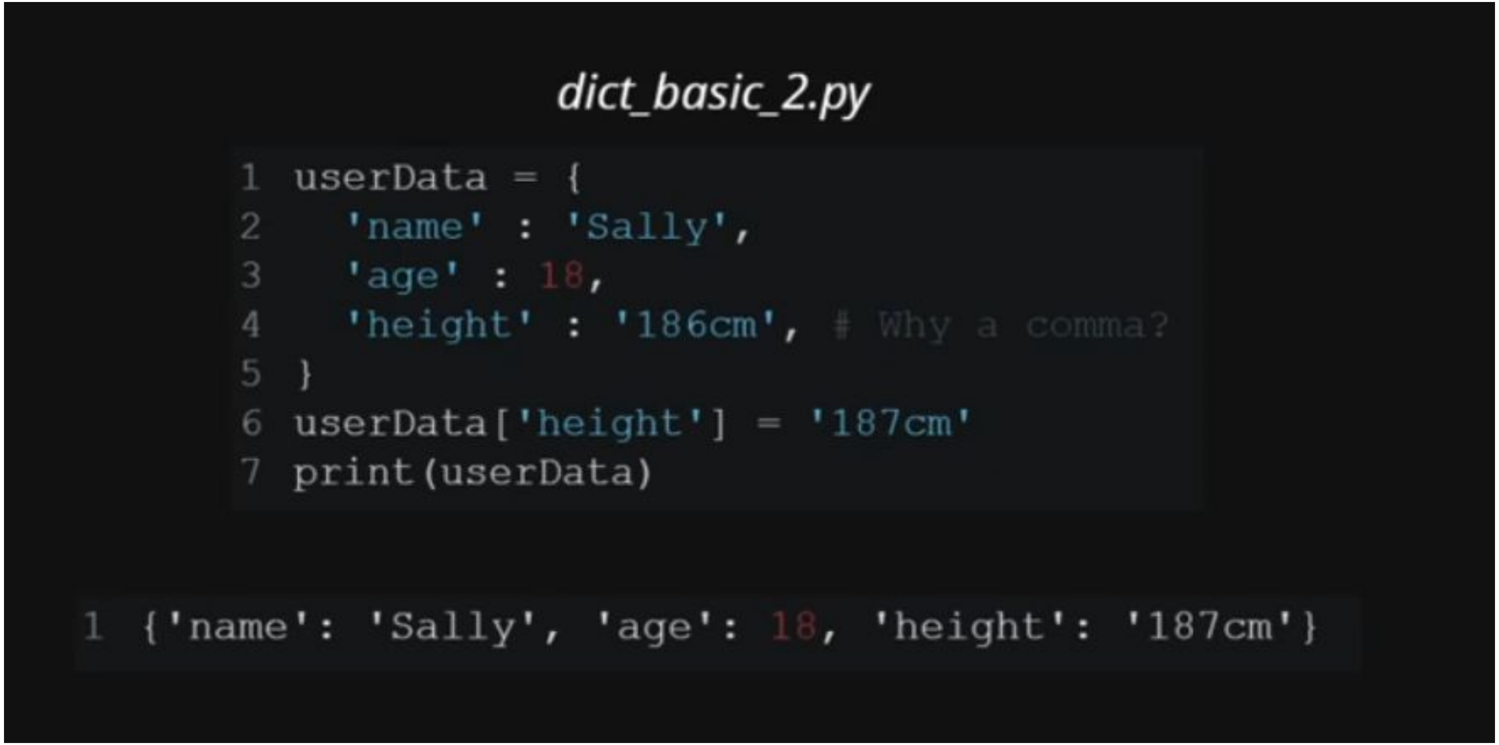
One great way to use tuples is through destructuring - where you can easily unpackage a tuple

destructure.py

```python
items = [(1, 'a'), (2, 'b'), (3, 'c')]

# Meh
for item in items:
  number = item[0]
  char = item[1]
  print(f"{number} {char}")

# Good!
for item in items:
  number, char = item
  print(f"{number} {char}")

# Amazing
for number, char in items:
  print(f"{number} {char}")
```


y U
filename.py U
foo.py 1, U
destructure.py 1, U
Jsers > sandeep > Documents > Code > UNSW > tutoring > cs1531 > 21t3 > we

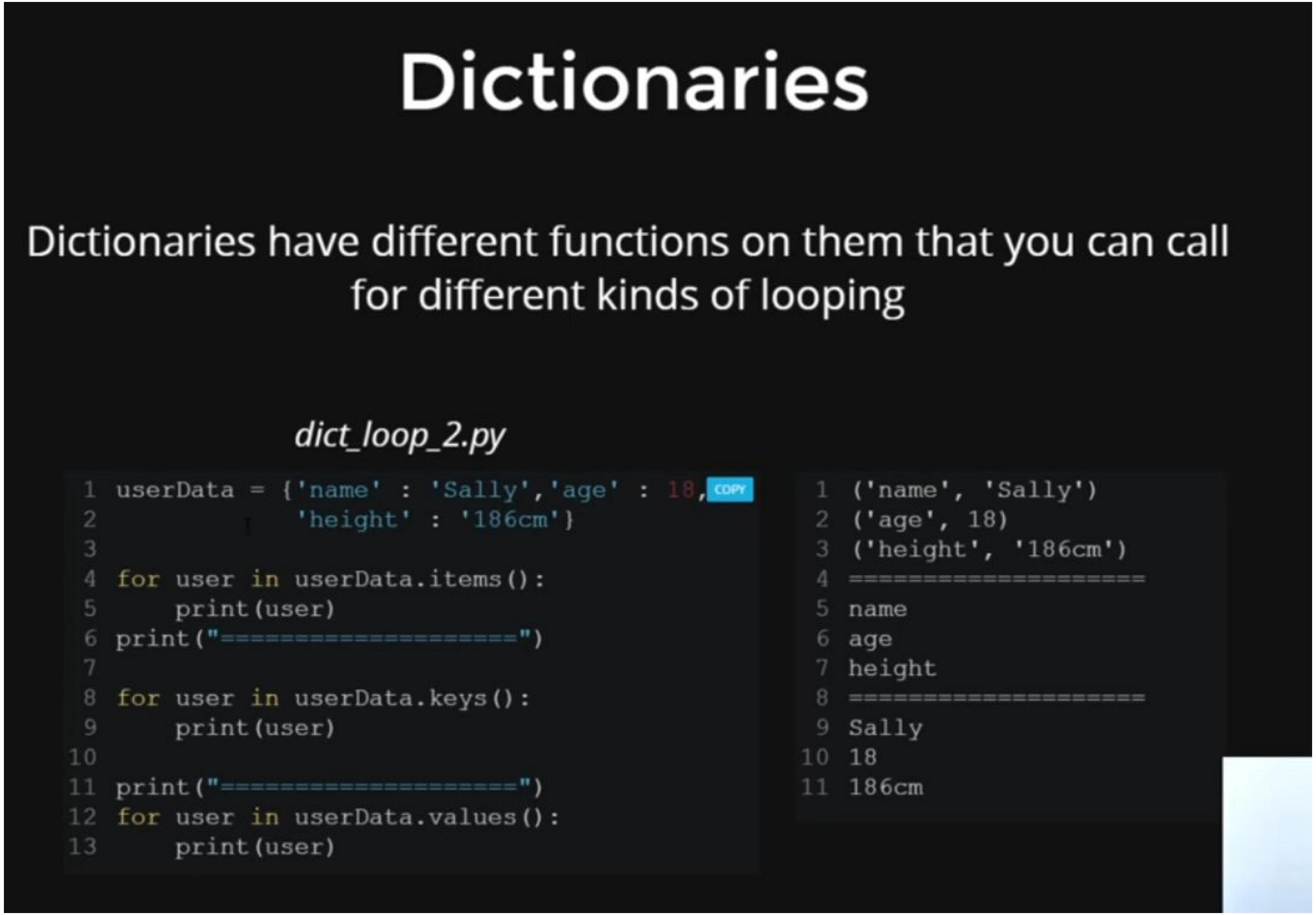
```python
import pytest

@pytest.fixture
def create_animal():
    name = "Lanky Joe"
    color = "Yellow"
    dob = "05/07/17"
    animal = "Giraffe"
    return (name, color, dob, animal)

def test_print_animal(create_animal):
    name, color, _, animal = create_animal
    print(f"{name} is a {color} {animal}")

```



dict_basic_2.py

```python
userData = {
  'name' : 'Sally',
  'age' : 18,
  'height' : '186cm', # Why a comma?
}
userData['height'] = '187cm'
print(userData)
```

```
{'name': 'Sally', 'age': 18, 'height': '187cm'}
```



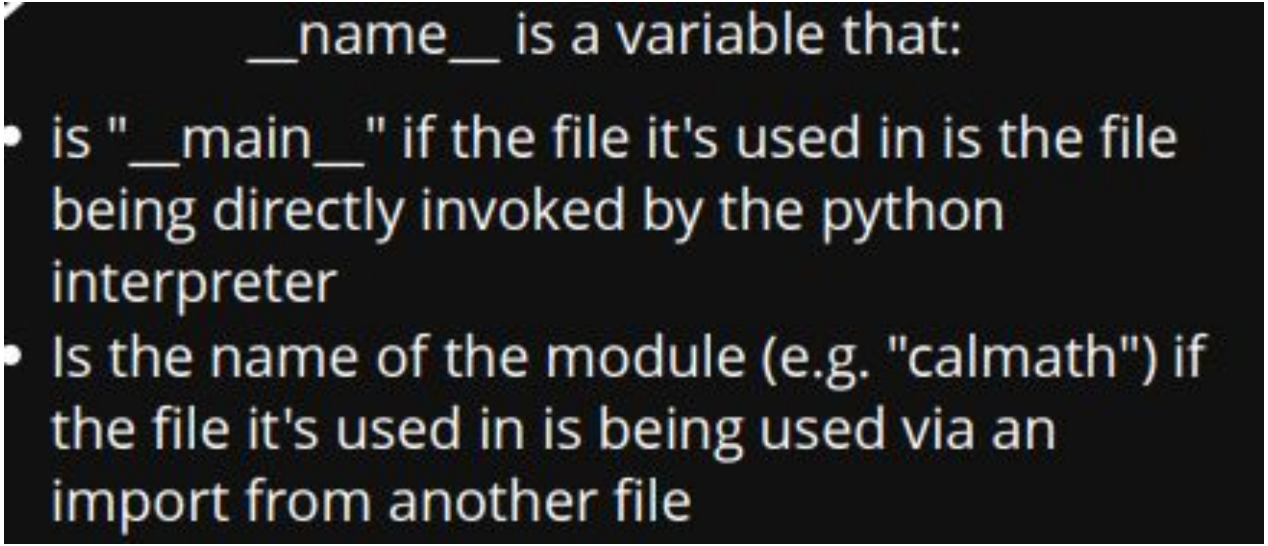
Dictionaries

Dictionaries have different functions on them that you can call for different kinds of looping

dict_loop_2.py

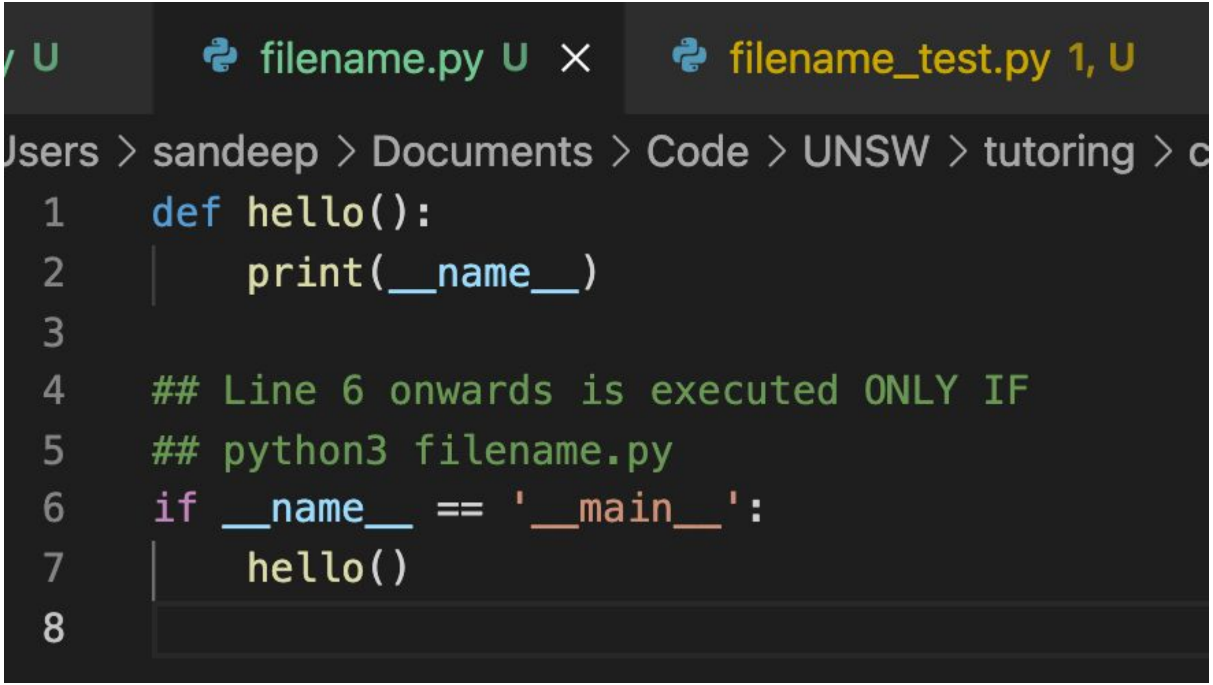
```python
userData = {'name' : 'Sally','age' : 18,
            'height' : '186cm'}

for user in userData.items():
    print(user)
print("====================")

for user in userData.keys():
    print(user)

print("====================")
for user in userData.values():
    print(user)
```

```
('name', 'Sally')
('age', 18)
('height', '186cm')
====================
name
age
height
====================
Sally
18
186cm
```

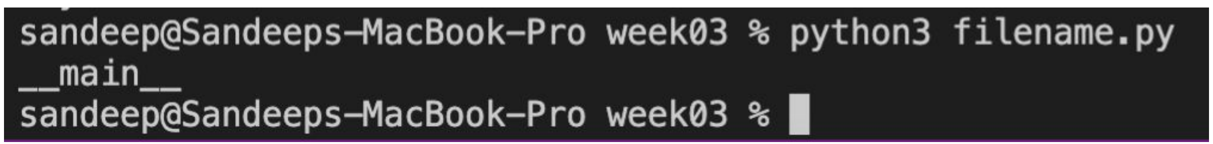


__name__ is a variable that:

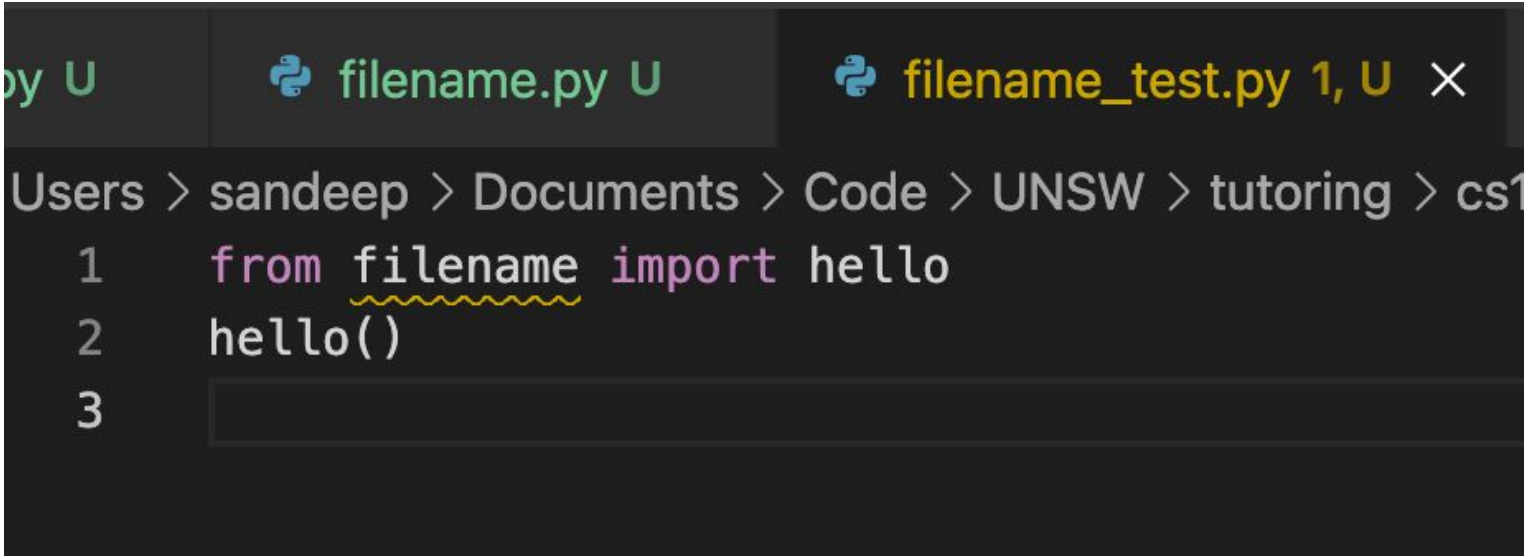
- is "__main__" if the file it's used in is the file being directly invoked by the python interpreter
- Is the name of the module (e.g. "calmath") if the file it's used in is being used via an import from another file



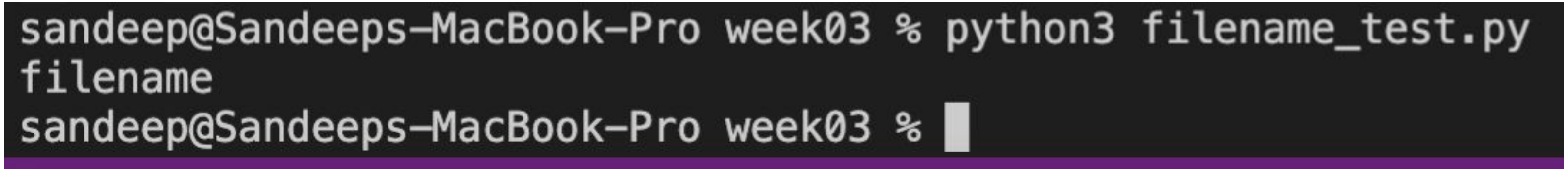
```
1   def hello():
2       print(__name__)
3
4   ## Line 6 onwards is executed ONLY IF
5   ## python3 filename.py
6   if __name__ == '__main__':
7       hello()
8
```



```
sandeep@Sandeeps-MacBook-Pro week03 % python3 filename.py
__main__
sandeep@Sandeeps-MacBook-Pro week03 % 
```



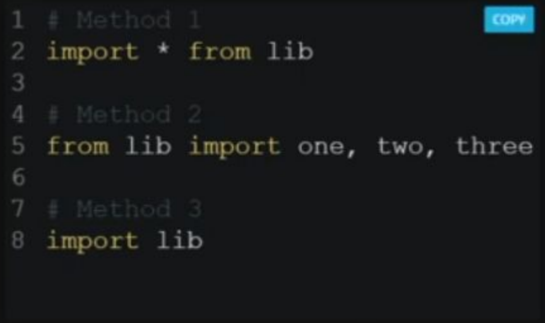
```python
from filename import hello
hello()

```



```
sandeep@Sandeeps-MacBook-Pro week03 % python3 filename_test.py
filename
sandeep@Sandeeps-MacBook-Pro week03 % 
```

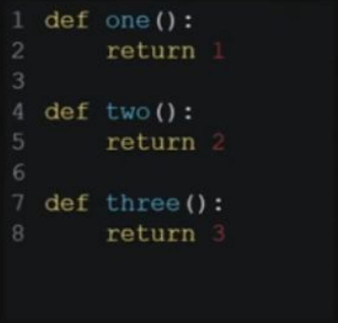
Ways to import

use.py



```python
# Method 1
import * from lib

# Method 2
from lib import one, two, three

# Method 3
import lib
```

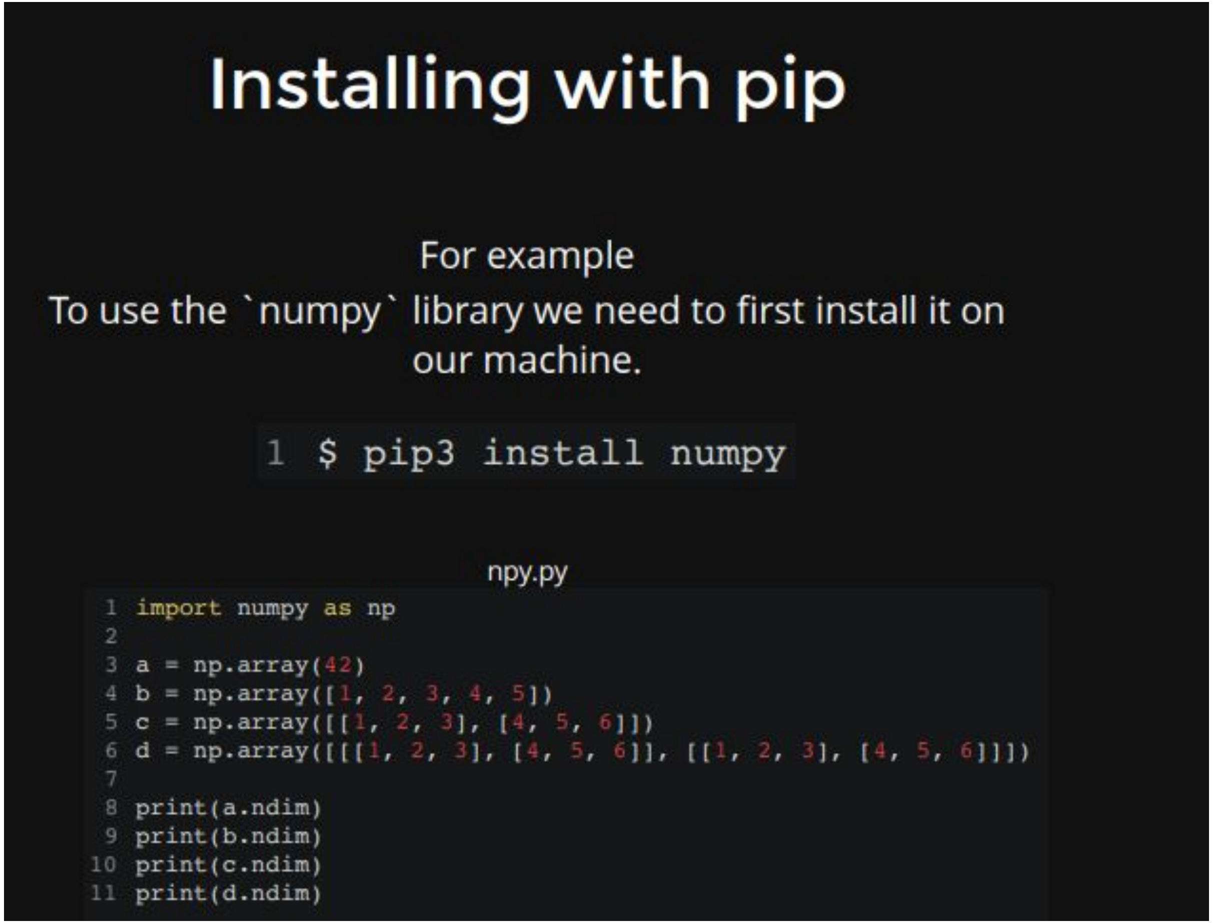
lib.py



```python
def one():
    return 1

def two():
    return 2

def three():
    return 3
```

Which ways do we prefer and why?

- Method 1 pollutes the namespace
- Method 2 generally clearest
- Method 3 useful if imported items need context



Installing with pip

For example
To use the `numpy` library we need to first install it on our machine.

```
$ pip3 install numpy
```

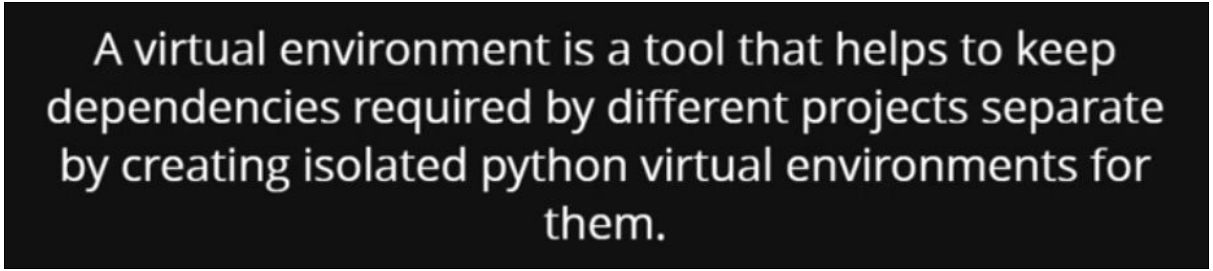
npy.py

```python
import numpy as np

a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

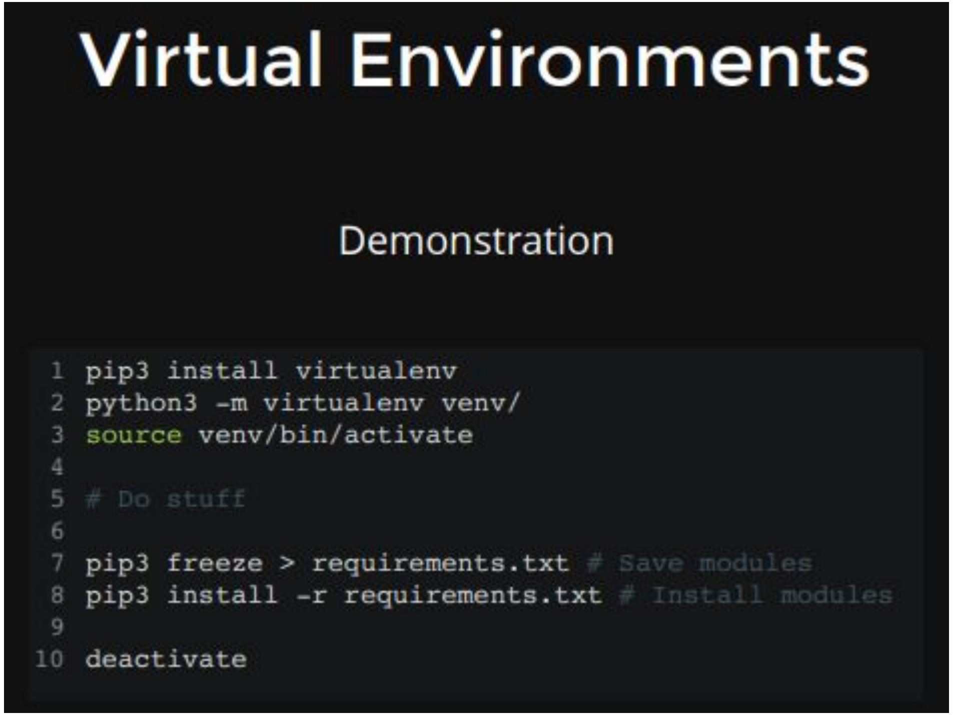
Potential issues with installing

Even though we know how to install modules, we now run into a problem:

- How do I easily share the modules that I've installed with my team members?
- How do I ensure my project doesn't end up accidentally using installed modules from other projects, and vice versa?



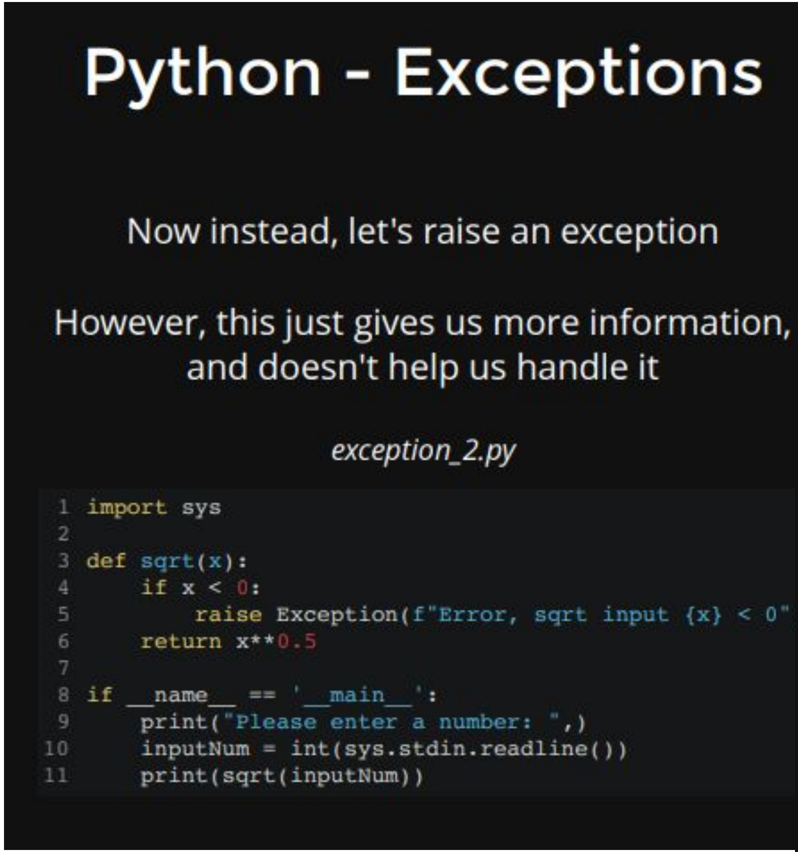
A virtual environment is a tool that helps to keep dependencies required by different projects separate by creating isolated python virtual environments for them.



Virtual Environments

Demonstration

```
pip3 install virtualenv
python3 -m virtualenv venv/
source venv/bin/activate

# Do stuff

pip3 freeze > requirements.txt # Save modules
pip3 install -r requirements.txt # Install modules

deactivate
```



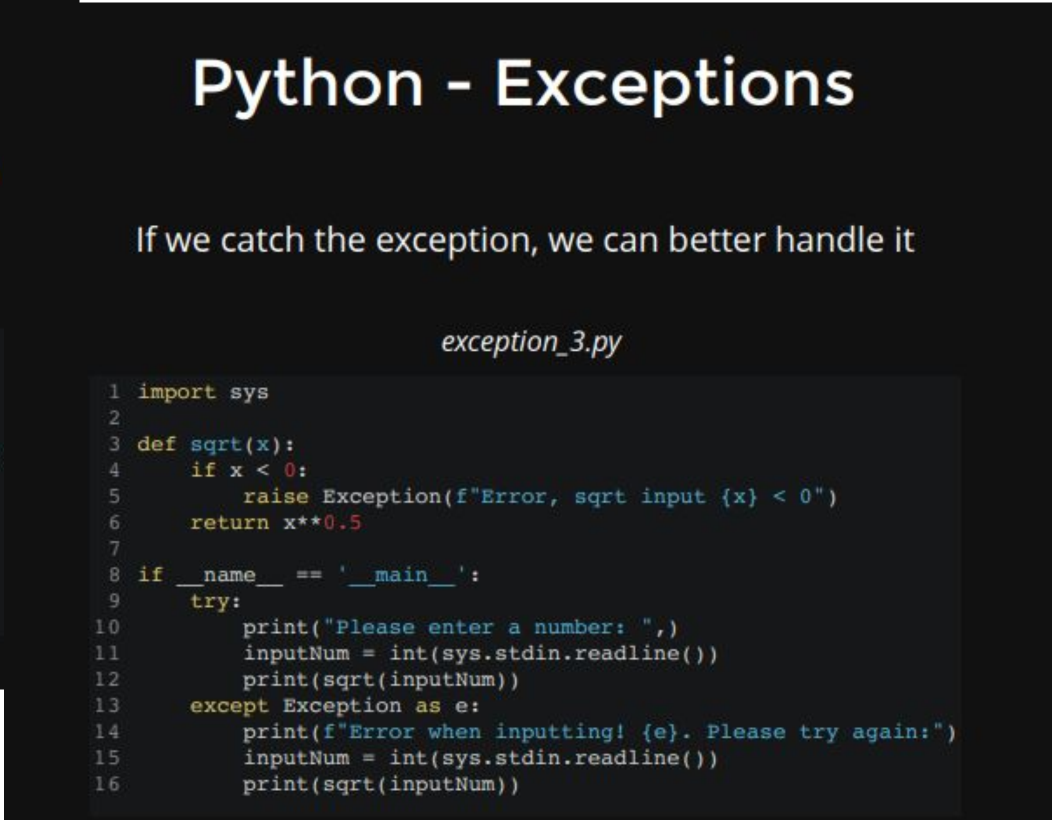
Python - Exceptions

Now instead, let's raise an exception

However, this just gives us more information, and doesn't help us handle it

exception_2.py

```python
import sys

def sqrt(x):
    if x < 0:
        raise Exception(f"Error, sqrt input {x} < 0"
    return x**0.5

if __name__ == '__main__':
    print("Please enter a number: ",)
    inputNum = int(sys.stdin.readline())
    print(sqrt(inputNum))
```

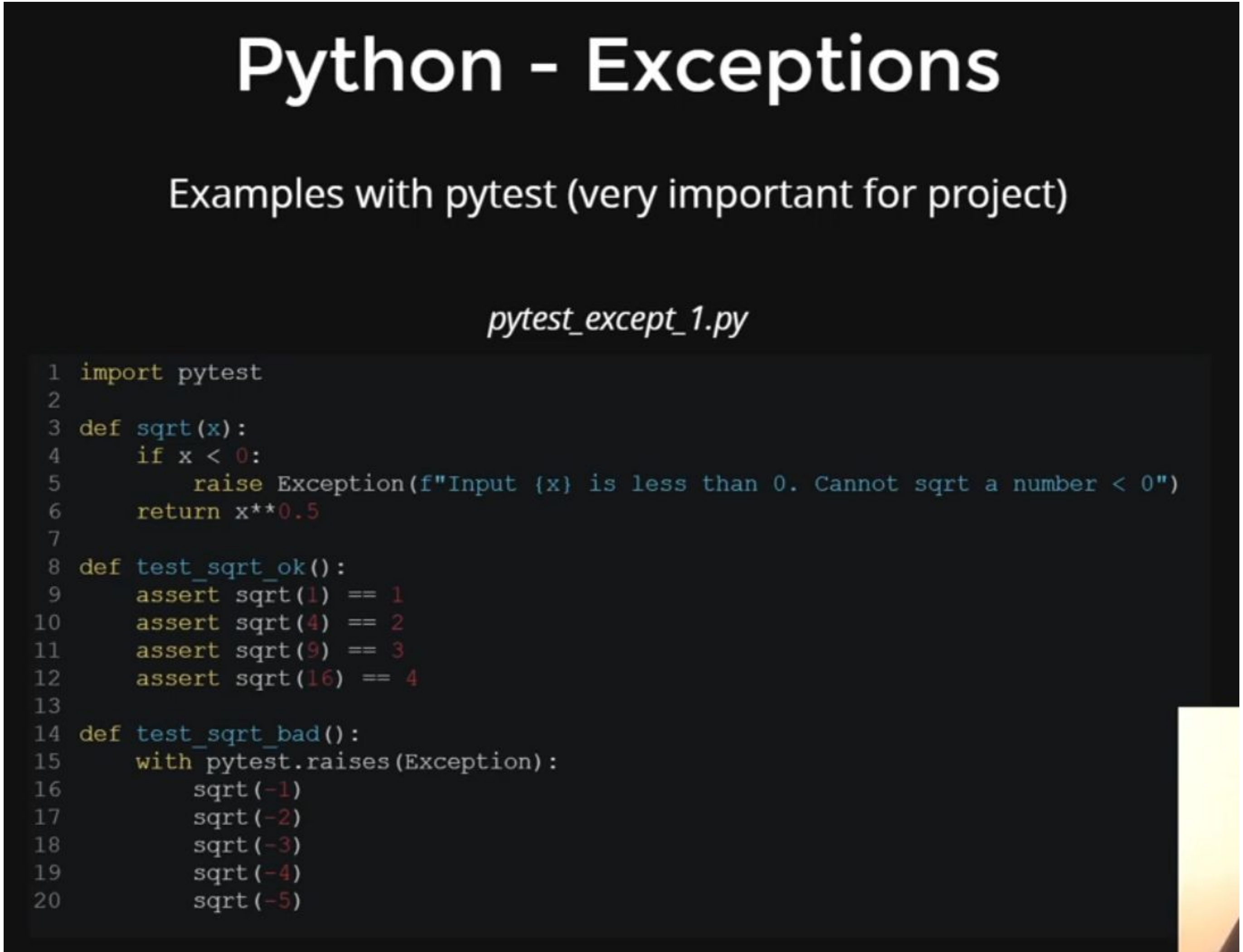


Python - Exceptions

If we catch the exception, we can better handle it

exception_3.py

```python
import sys

def sqrt(x):
    if x < 0:
        raise Exception(f"Error, sqrt input {x} < 0")
    return x**0.5

if __name__ == '__main__':
    try:
        print("Please enter a number: ",)
        inputNum = int(sys.stdin.readline())
        print(sqrt(inputNum))
    except Exception as e:
        print(f"Error when inputting! {e}. Please try again:")
        inputNum = int(sys.stdin.readline())
        print(sqrt(inputNum))
```

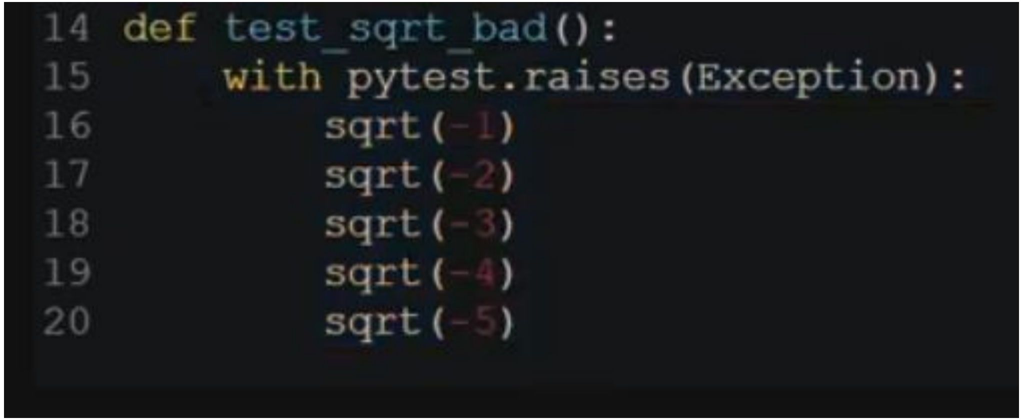


Python - Exceptions

Examples with pytest (very important for project)

pytest_except_1.py

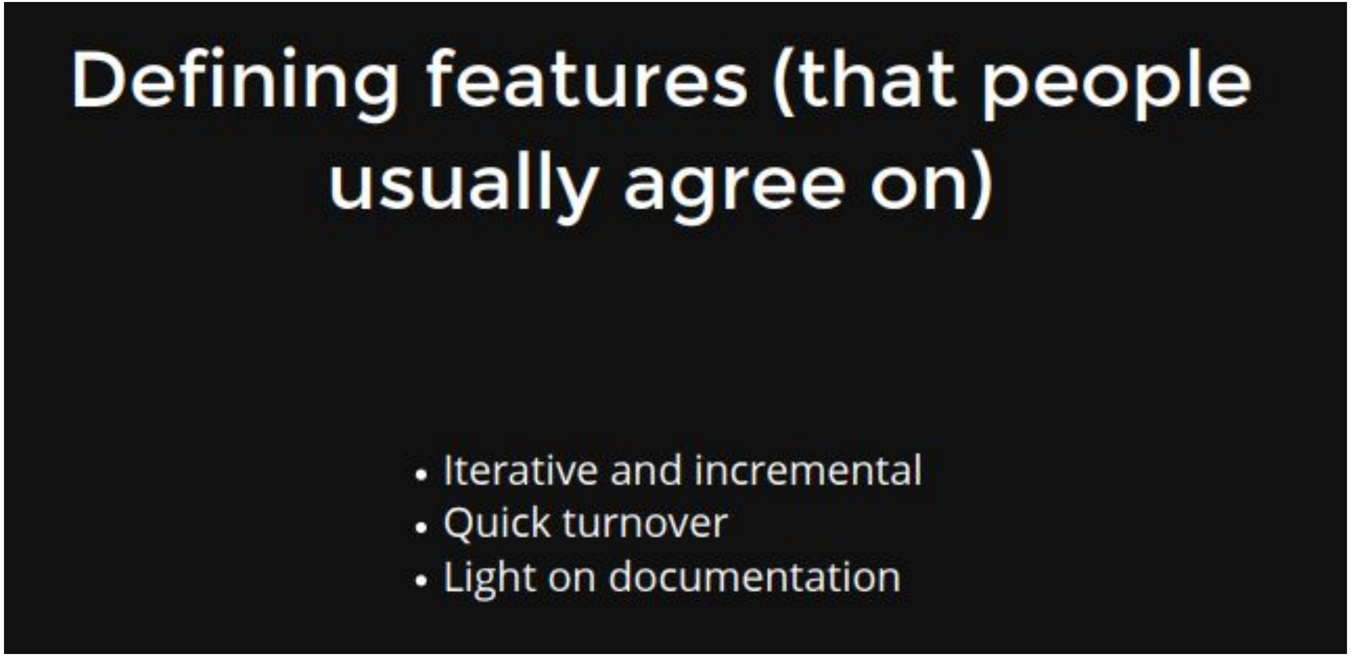
```python
import pytest

def sqrt(x):
    if x < 0:
        raise Exception(f"Input {x} is less than 0. Cannot sqrt a number < 0")
    return x**0.5

def test_sqrt_ok():
    assert sqrt(1) == 1
    assert sqrt(4) == 2
    assert sqrt(9) == 3
    assert sqrt(16) == 4

def test_sqrt_bad():
    with pytest.raises(Exception):
        sqrt(-1)
        sqrt(-2)
        sqrt(-3)
        sqrt(-4)
        sqrt(-5)
```



```
def test_sqrt_bad():
    with pytest.raises(Exception):
        sqrt(-1)
        sqrt(-2)
        sqrt(-3)
        sqrt(-4)
        sqrt(-5)
```

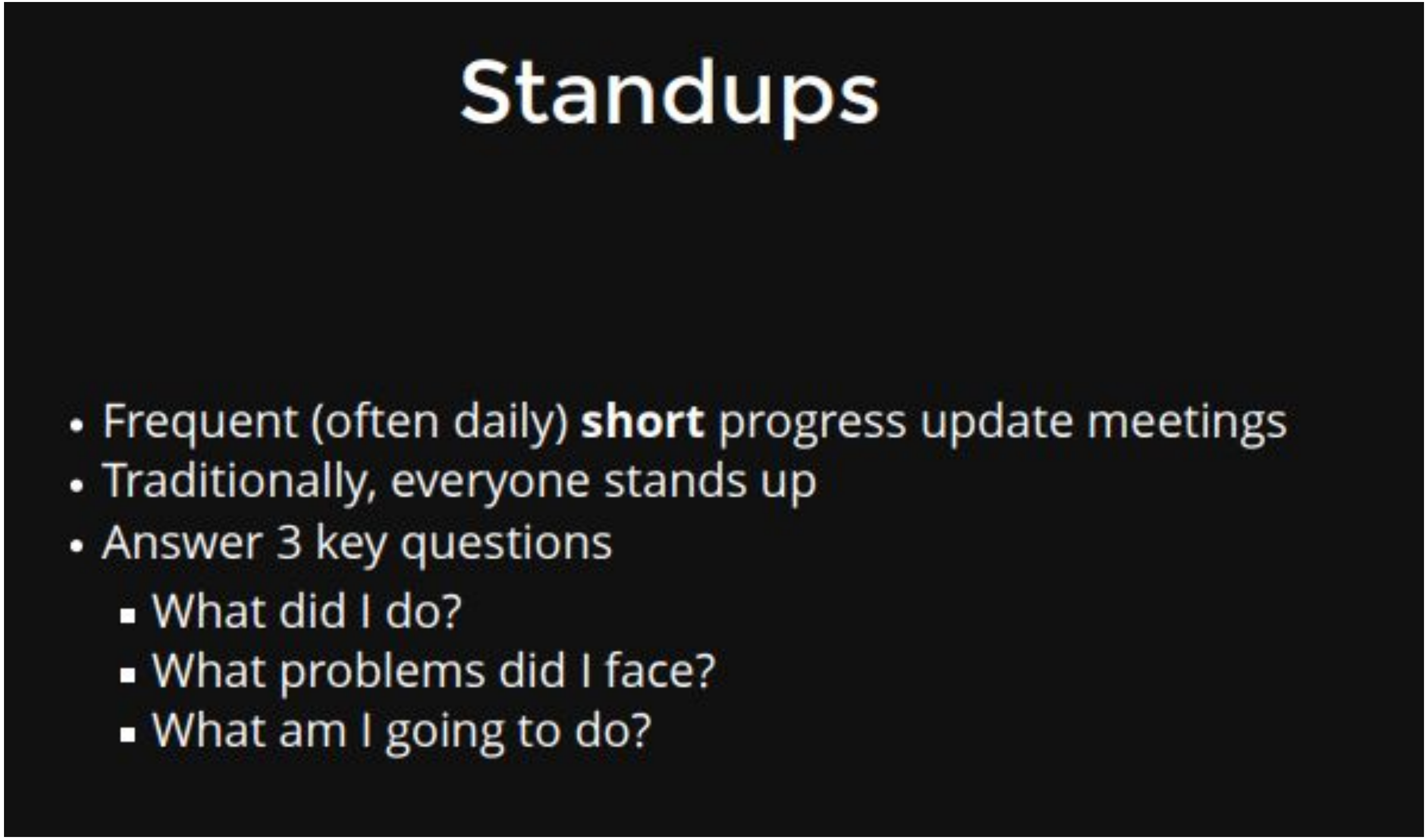
Warning: as long as one of these throw exception, then it passes the test

Agile



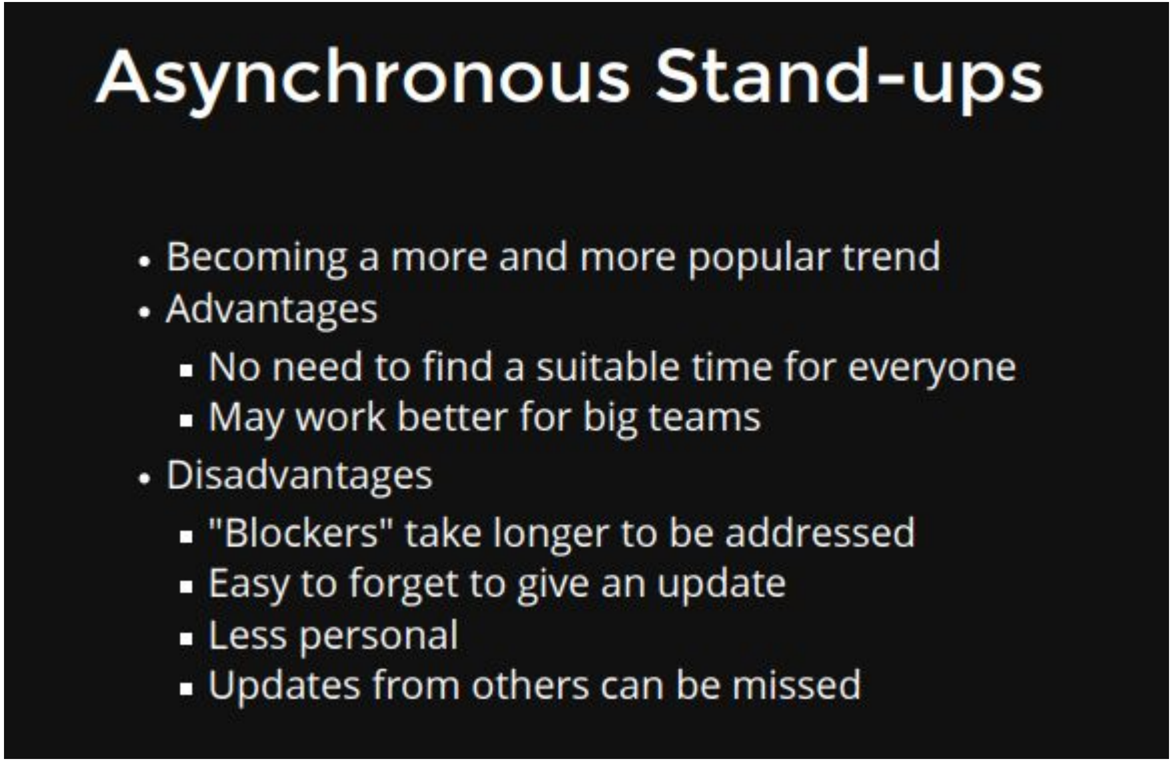
Defining features (that people usually agree on)

- Iterative and incremental
- Quick turnover
- Light on documentation



Standups

- Frequent (often daily) **short** progress update meetings
- Traditionally, everyone stands up
- Answer 3 key questions
  - What did I do?
  - What problems did I face?
  - What am I going to do?

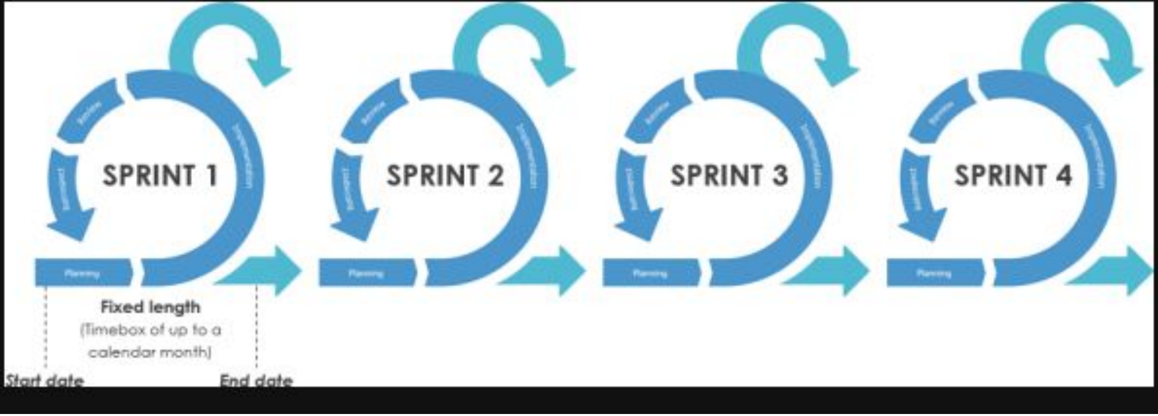


Asynchronous Stand-ups

- Becoming a more and more popular trend
- Advantages
  - No need to find a suitable time for everyone
  - May work better for big teams
- Disadvantages
  - "Blockers" take longer to be addressed
  - Easy to forget to give an update
  - Less personal
  - Updates from others can be missed

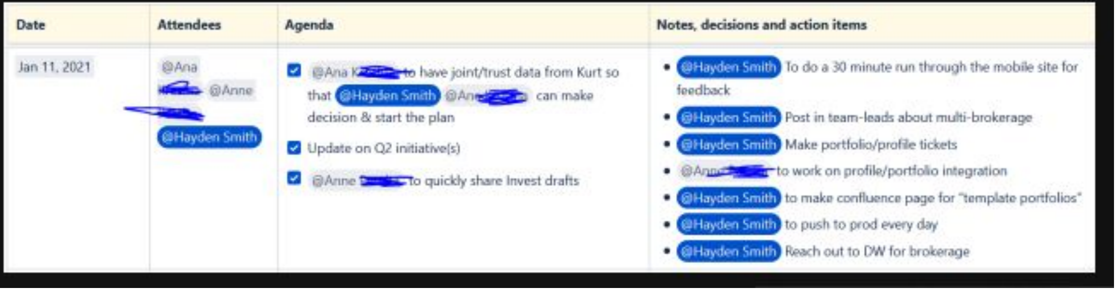
Sprints

- A fixed of time (e.g. week, fortnight) where you set a number of tasks to be completed in the team.
- After that period is up, you review progress, and set tasks for the next sprint.
- Time is fixed, scope is flexible
- Plan only for the next sprint
- Typically have a release at the end of each sprint

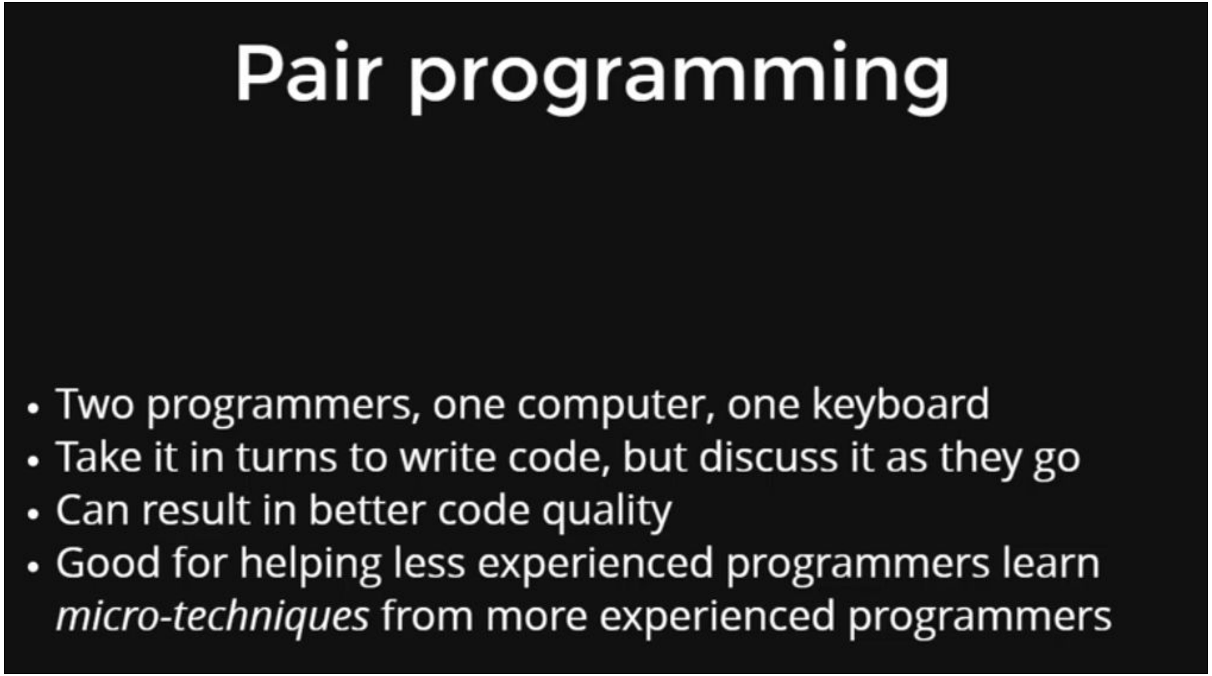


Meeting Minutes

- It's a good idea for teams to have more formal meetings at least once a fortnight.
- Typically this kind of meeting would fall into a start-of-sprint style meeting, though in the absence of sprints just a "weekly meeting" is adequate.
- During meeting it's usually a good idea to have someone take meeting minutes (i.e. "notes")
- Meeting minutes will typically consist of documenting:
  - Attendees
  - (Optional) Agenda
  - Discussion Points
  - Actions



Date	Attendees	Agenda	Notes, decisions and action items
Jan 11, 2021	@Ana [redacted] @Anne [redacted] @Hayden Smith	☑ @Ana K[redacted] have joint/trust data from Kurt so that @Hayden Smith @An[redacted] can make decision & start the plan ☑ Update on Q2 initiative(s) ☑ @Anne [redacted] to quickly share Invest drafts	• @Hayden Smith To do a 30 minute run through the mobile site for feedback • @Hayden Smith Post in team-leads about multi-brokerage • @Hayden Smith Make portfolio/profile tickets • @Ann[redacted] to work on profile/portfolio integration • @Hayden Smith to make confluence page for "template portfolios" • @Hayden Smith to push to prod every day • @Hayden Smith Reach out to DW for brokerage



Pair programming

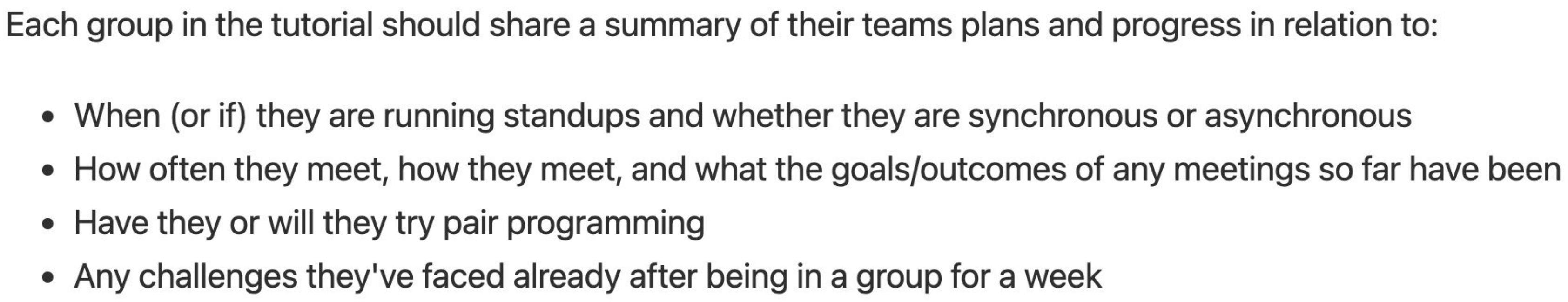
- Two programmers, one computer, one keyboard
- Take it in turns to write code, but discuss it as they go
- Can result in better code quality
- Good for helping less experienced programmers learn *micro-techniques* from more experienced programmers

Note

Use your name as it appears on roll, for zoom (participation marks)

- Couldn’t find one person from last lesson

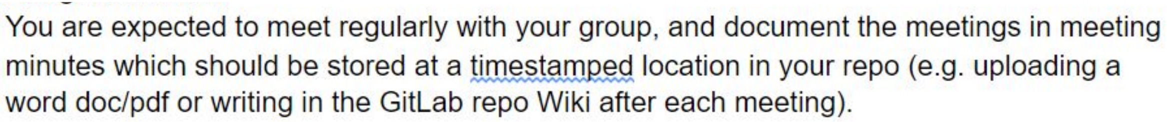
Part B (if time)



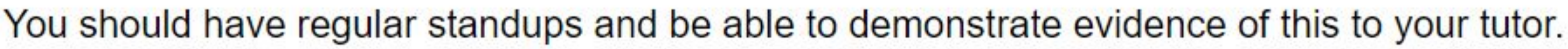
Each group in the tutorial should share a summary of their teams plans and progress in relation to:

- When (or if) they are running standups and whether they are synchronous or asynchronous
- How often they meet, how they meet, and what the goals/outcomes of any meetings so far have been
- Have they or will they try pair programming
- Any challenges they've faced already after being in a group for a week

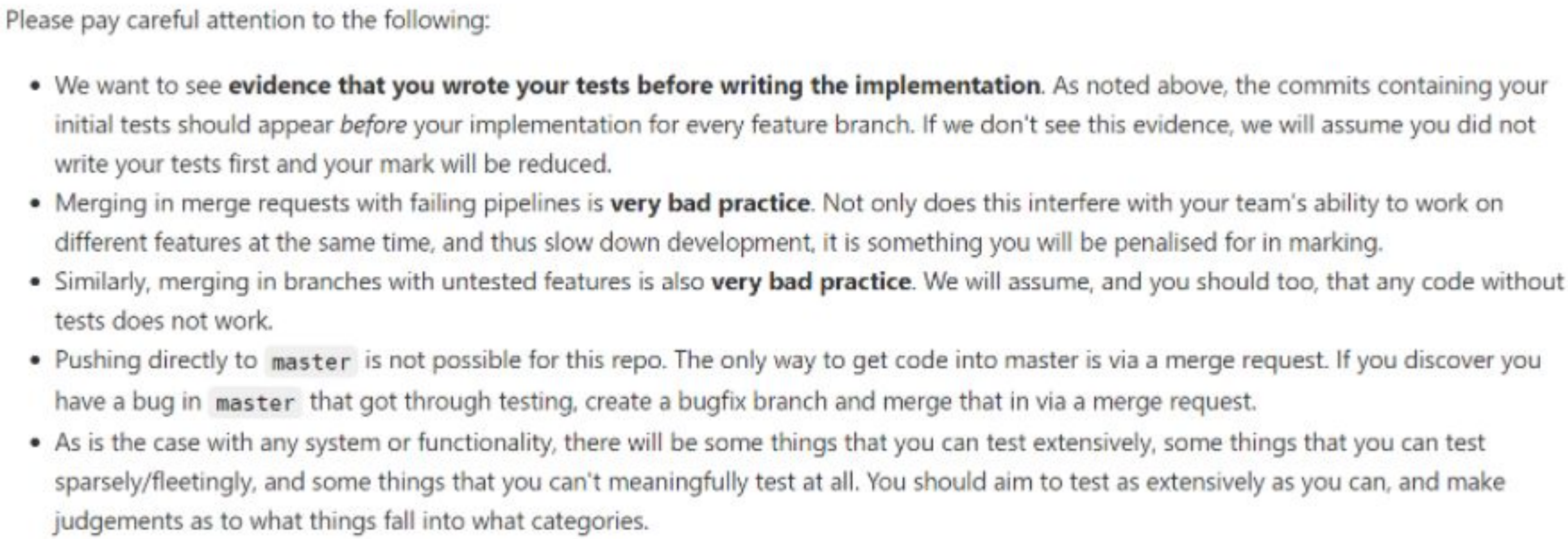
Assignment stuff



You are expected to meet regularly with your group, and document the meetings in meeting minutes which should be stored at a timestamped location in your repo (e.g. uploading a word doc/pdf or writing in the GitLab repo Wiki after each meeting).



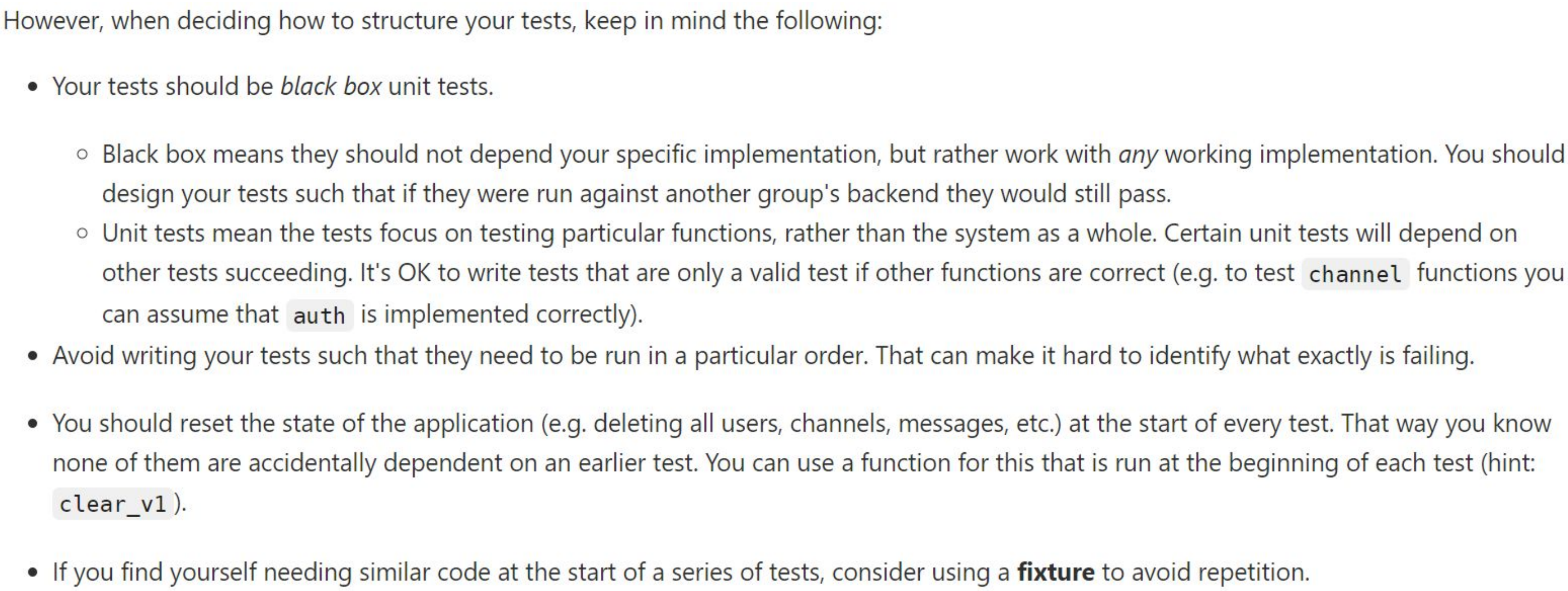
You should have regular standups and be able to demonstrate evidence of this to your tutor.



Please pay careful attention to the following:

- We want to see **evidence that you wrote your tests before writing the implementation**. As noted above, the commits containing your initial tests should appear *before* your implementation for every feature branch. If we don't see this evidence, we will assume you did not write your tests first and your mark will be reduced.
- Merging in merge requests with failing pipelines is **very bad practice**. Not only does this interfere with your team's ability to work on different features at the same time, and thus slow down development, it is something you will be penalised for in marking.
- Similarly, merging in branches with untested features is also **very bad practice**. We will assume, and you should too, that any code without tests does not work.
- Pushing directly to `master` is not possible for this repo. The only way to get code into master is via a merge request. If you discover you have a bug in `master` that got through testing, create a bugfix branch and merge that in via a merge request.
- As is the case with any system or functionality, there will be some things that you can test extensively, some things that you can test sparsely/fleetingly, and some things that you can't meaningfully test at all. You should aim to test as extensively as you can, and make judgements as to what things fall into what categories.

Fixtures

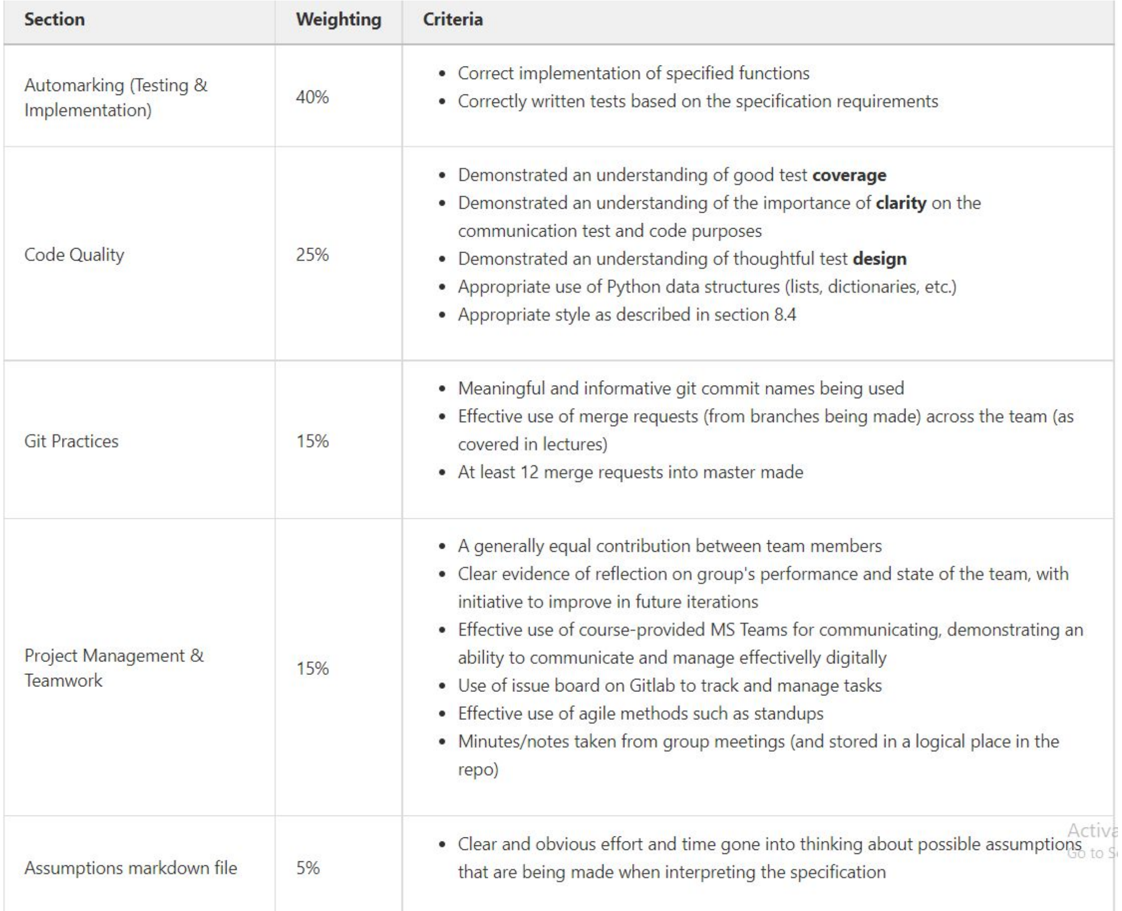


However, when deciding how to structure your tests, keep in mind the following:

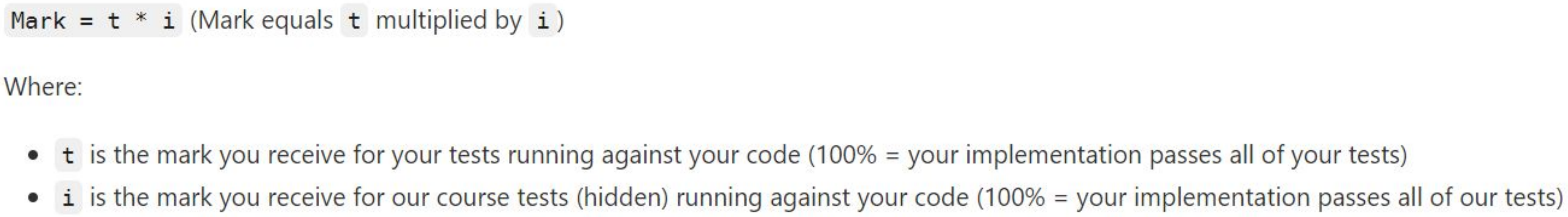
- Your tests should be *black box* unit tests.
  - Black box means they should not depend your specific implementation, but rather work with *any* working implementation. You should design your tests such that if they were run against another group's backend they would still pass.
  - Unit tests mean the tests focus on testing particular functions, rather than the system as a whole. Certain unit tests will depend on other tests succeeding. It's OK to write tests that are only a valid test if other functions are correct (e.g. to test `channel` functions you can assume that `auth` is implemented correctly).
- Avoid writing your tests such that they need to be run in a particular order. That can make it hard to identify what exactly is failing.
- You should reset the state of the application (e.g. deleting all users, channels, messages, etc.) at the start of every test. That way you know none of them are accidentally dependent on an earlier test. You can use a function for this that is run at the beginning of each test (hint: `clear_v1`).
- If you find yourself needing similar code at the start of a series of tests, consider using a **fixture** to avoid repetition.

Help session:

- Some groups tested too much, you should assume your tests will be run against Hayden’s implementations



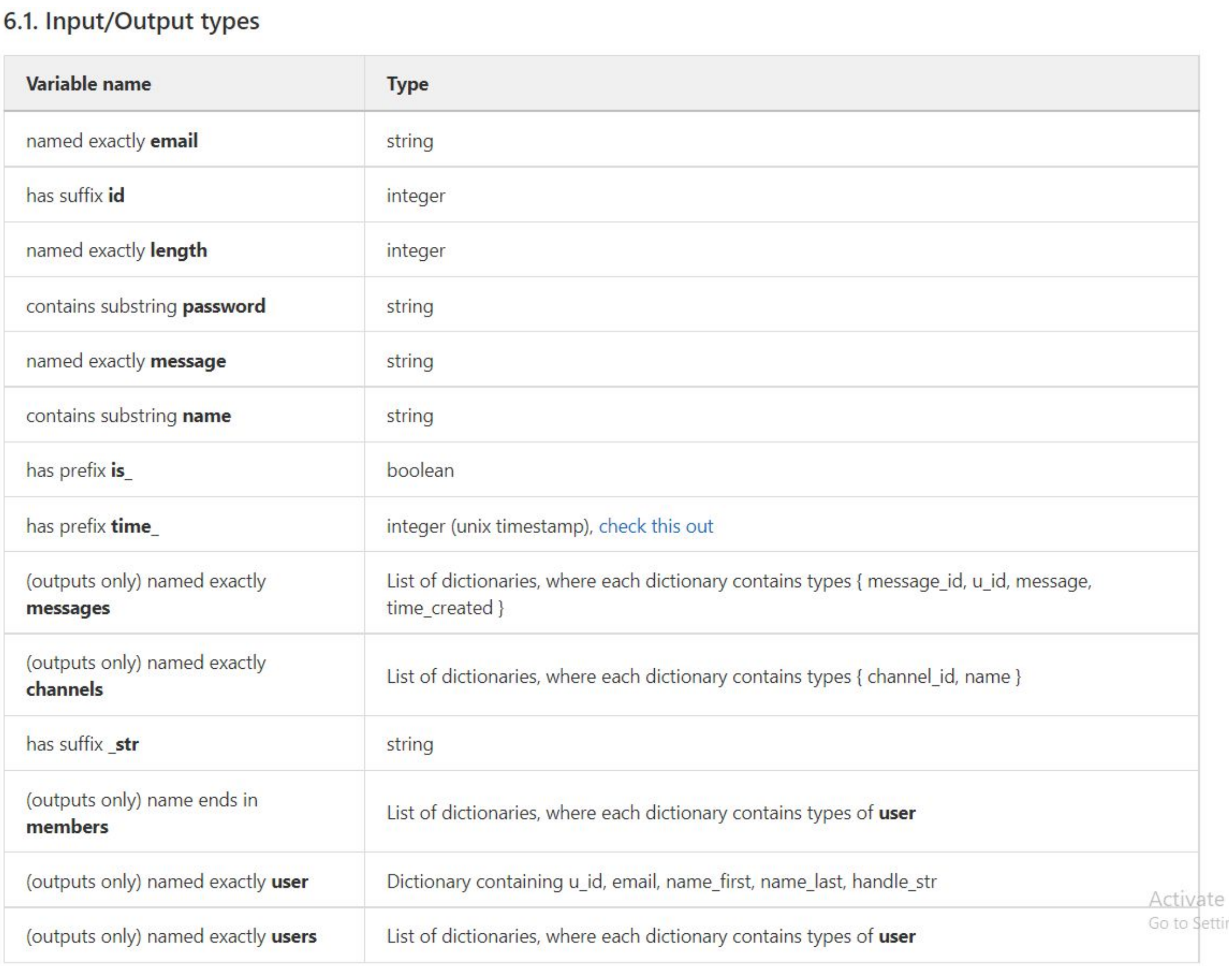
Section	Weighting	Criteria
Automarking (Testing & Implementation)	40%	• Correct implementation of specified functions • Correctly written tests based on the specification requirements
Code Quality	25%	• Demonstrated an understanding of good test **coverage** • Demonstrated an understanding of the importance of **clarity** on the communication test and code purposes • Demonstrated an understanding of thoughtful test **design** • Appropriate use of Python data structures (lists, dictionaries, etc.) • Appropriate style as described in section 8.4
Git Practices	15%	• Meaningful and informative git commit names being used • Effective use of merge requests (from branches being made) across the team (as covered in lectures) • At least 12 merge requests into master made
Project Management & Teamwork	15%	• A generally equal contribution between team members • Clear evidence of reflection on group's performance and state of the team, with initiative to improve in future iterations • Effective use of course-provided MS Teams for communicating, demonstrating an ability to communicate and manage effectivelly digitally • Use of issue board on Gitlab to track and manage tasks • Effective use of agile methods such as standups • Minutes/notes taken from group meetings (and stored in a logical place in the repo)
Assumptions markdown file	5%	• Clear and obvious effort and time gone into thinking about possible assumptions that are being made when interpreting the specification



`Mark = t * i` (Mark equals `t` multiplied by `i`)

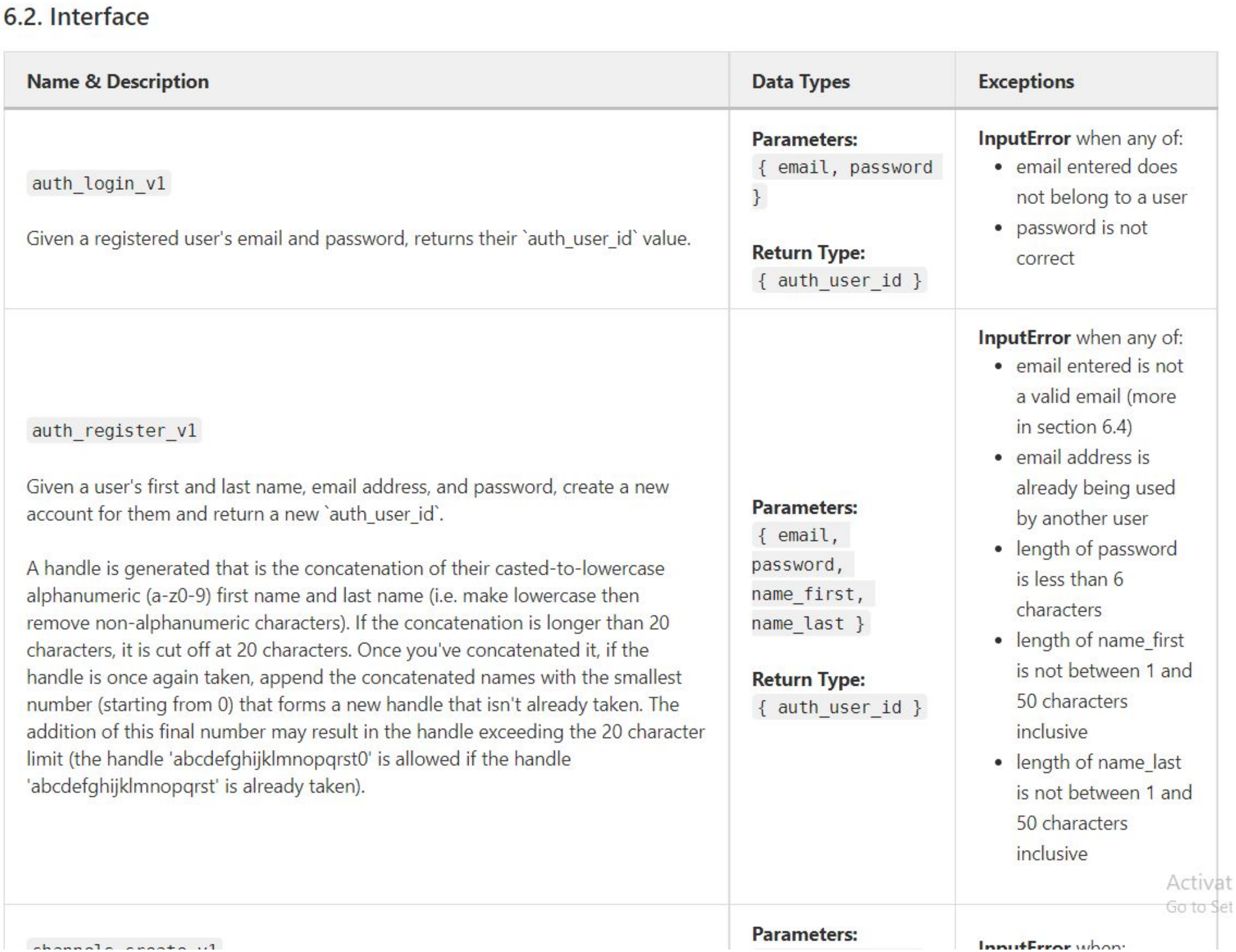
Where:

- `t` is the mark you receive for your tests running against your code (100% = your implementation passes all of your tests)
- `i` is the mark you receive for our course tests (hidden) running against your code (100% = your implementation passes all of our tests)



6.1. Input/Output types

Variable name	Type
named exactly **email**	string
has suffix **id**	integer
named exactly **length**	integer
contains substring **password**	string
named exactly **message**	string
contains substring **name**	string
has prefix **is_**	boolean
has prefix **time_**	integer (unix timestamp), check this out
(outputs only) named exactly **messages**	List of dictionaries, where each dictionary contains types { message_id, u_id, message, time_created }
(outputs only) named exactly **channels**	List of dictionaries, where each dictionary contains types { channel_id, name }
has suffix **_str**	string
(outputs only) name ends in **members**	List of dictionaries, where each dictionary contains types of **user**
(outputs only) named exactly **user**	Dictionary containing u_id, email, name_first, name_last, handle_str
(outputs only) named exactly **users**	List of dictionaries, where each dictionary contains types of **user**



6.2. Interface

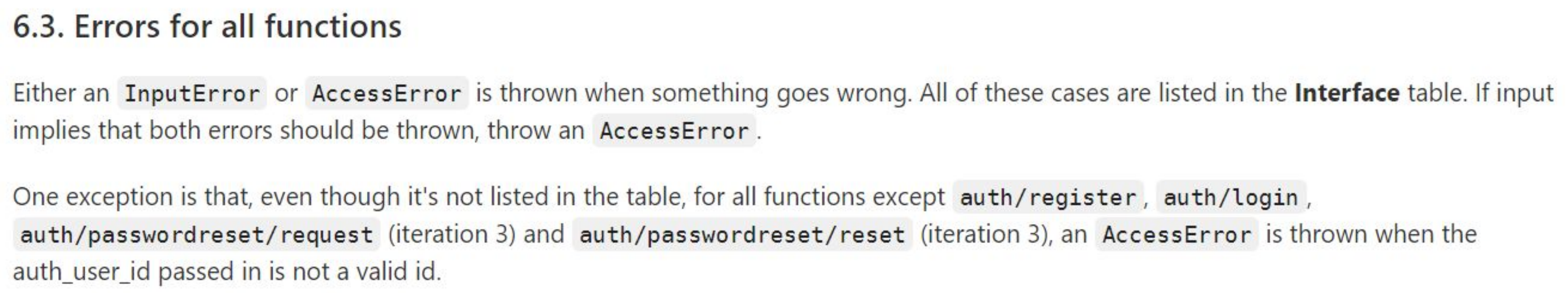
Name & Description	Data Types	Exceptions

`auth_login_v1`
 	
Given a registered user's email and password, returns their `auth_user_id` value.
	Parameters:
	`{ email, password }`
	
	Return Type:
	`{ auth_user_id }`
		InputError when any of:
		• email entered does not belong to a user
		• password is not correct

`auth_register_v1`
 	
Given a user's first and last name, email address, and password, create a new account for them and return a new `auth_user_id`.
 	
A handle is generated that is the concatenation of their casted-to-lowercase alphanumeric (a-z0-9) first name and last name (i.e. make lowercase then remove non-alphanumeric characters). If the concatenation is longer than 20 characters, it is cut off at 20 characters. Once you've concatenated it, if the handle is once again taken, append the concatenated names with the smallest number (starting from 0) that forms a new handle that isn't already taken. The addition of this final number may result in the handle exceeding the 20 character limit (the handle 'abcdefghijklmnopqrst0' is allowed if the handle 'abcdefghijklmnopqrst' is already taken).
	Parameters:
	`{ email, password, name_first, name_last }`
	
	Return Type:
	`{ auth_user_id }`
		InputError when any of:
		• email entered is not a valid email (more in section 6.4)
		• email address is already being used by another user
		• length of password is less than 6 characters
		• length of name_first is not between 1 and 50 characters inclusive
		• length of name_last is not between 1 and 50 characters inclusive

`channels_create_v1`	**Parameters:**	**InputError** when:

AccessError is thrown when the auth_user_id passed in is not a valid id.



6.3. Errors for all functions

Either an `InputError` or `AccessError` is thrown when something goes wrong. All of these cases are listed in the **Interface** table. If input implies that both errors should be thrown, throw an `AccessError`.

One exception is that, even though it's not listed in the table, for all functions except `auth/register`, `auth/login`, `auth/passwordreset/request` (iteration 3) and `auth/passwordreset/reset` (iteration 3), an `AccessError` is thrown when the auth_user_id passed in is not a valid id.