Lecture 3:
While, Dictionaries, Modules, Reproducibility

TAs: Ehsan Baratnezhad (ethan.b@dal.ca); Precious Osadebamwen (precious.osadebamwen@dal.ca)

Overview

- List functions (range, zip, lazy evaluation)
- While loops
- Dictionaries
- Modules, Packages, and Import
- Namespaces
- Reproducibility/Clean Notebook

Handy loop/list related functions

Generating a list of numbers with “range”

```
>>> list(range(4))
[0, 1, 2, 3]
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
>>> list(range(2, 9, 2))
[2, 4, 6, 8]
```

- Same option start, stop, increment as list indexing.
- range(4) same as range(0, 4, 1)

Many list related functions do “lazy evaluation”

```
>>> x = enumerate([‘a’, ‘b’, ‘c’])

>>> x

<enumerate at 0x738dcf5156c0>



>>> list(x)

[(0, 'a'), (1, 'b'), (2, 'c')]
```

- Enumerate gives use the the index - value pairs
- Imagine x is very very big
- What if we only needed to enumerate the first couple of items in the list?
- Lazy evaluation means only doing calculations when (and therefore **IF**) they are actually needed

Zip efficiently combines lists

```
>>> x = [10, 50, 100]
>>> y = [‘a’, ‘b’, ‘c’]
>>> zipped = zip(x, y)
>>> zipped
<zip at 0x738dce007ec0>
>>> list(zipped)
[(10, 'a'), (50, 'b'), (100, 'c')]
```

Range is often used to generate indices for strings

```
for x in range(5, 15, 3):
    print(x)

5
8
11
14
```

```
a = ‘abc’

b = ‘123’

for i in range(len(a)):

    print(a[i] + b[i])

a1

b2

c3
```

We can also iterate over more than 1 list with zip

```
list1 = [‘a’, ‘b’, ‘c’]
list2 = [‘1’, ‘2’, ‘3’]
for a,b in zip(list1, list2):
    print(a + b)


a1
b2
c3
```

Doing a loop until something is True

Don’t always know how many times we need to loop

```
curr_temp, room_temp, minutes = 50, 25, 0

for i in range(1000000000):

    temp_diff = curr_temp - room_temp

    if abs(temp_diff) <= 0.5:

        break

    curr_temp = curr_temp - (0.1 * temp_diff)

    minutes += 1
```

```
curr_temp, room_temp, minutes = 50, 25, 0

while abs(curr_temp - room_temp) > 0.5:
    temp_diff = curr_temp - room_temp
    curr_temp = curr_temp - (0.1 * temp_diff)
    minutes += 1
```

while condition:
    keep running body

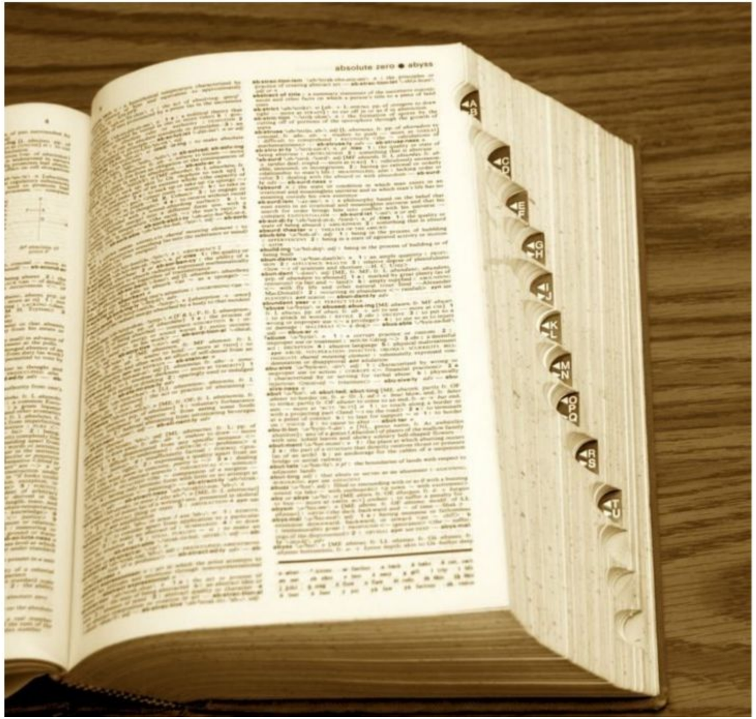
Beware - Infinite Loops

```
while False:
    print("Never execute")

while True:
    print("Will never end")

x = 50
while x < 100:
    x - 5
```

- If condition is **False** at beginning loop body will not run
- If condition can never be made **False** loop body will repeat until python crashes ("infinite loop")
    - Make sure variables in conditional are actually changed during the loop
    - Be careful with direction of inequalities
    - Be careful using "while True:"

Indexes that aren’t numbers?

Dictionaries (aka hashmaps, maps, hash…)



the *key*

Each word in a dictionary has a definition

the *value*

Words are stored in a way that enables easy look-up.

Dictionaries store link keys to values

```
d = {}  # create an empty dictionary
d[“key”] = “value” # map key -> value
d[“any number/string”] = 10
d[42] = “any variable”
d[“even list”] = [1, 2, 3, (4, 5)]


x = 5
y = {x: x+1, x-5: x/2} # variables
```

```
d
{‘key’: ‘value’,
 ‘any number/string’: 10,
 42: “any variable”,
 “even_list”: [1, 2, 3, (4, 5)]}


y
{5: 6, 0: 2.5}
```

Access contents of a dictionary

```
d = {‘key1’: ‘value1’,
      ‘key2’: [‘value2a’, ‘value2b’]}
d[‘key1’]      # ‘value1’
d[‘key2’][1]   # ‘value2b’
d.get(‘key2’)  # [‘value2a’,‘value2b’]
d.keys() # dict_keys([‘key1’, ‘key2’])
d.values()
# dict_values([‘value1’,
#              [‘value2a’, ‘value2b’])
```

- You can access specific values in dictionary with key in [] or with the .get() method
- d.keys() will provide list of all keys in a dictionary
- d.values() will provide list of all values in a dictionary
- d.items() will provide:
  - zip(d.keys(), d.values())

[(‘key1’, ‘value1’), (‘key2, [‘value2a’, ‘value2b’])]

Testing for keys in dictionary

```
d = {‘foo’: ‘bar’}
‘foo’ in d  # True
‘bar’ in d  # False (only checks keys)


‘key1’ in d # False
D[‘key1’]
KeyError: 'key1’
```

Get can be used to give value if key missing

```
d = dict([[1, ‘a’], [10, ‘c’]])

d.get(50, ‘Nope’)

‘Nope’


d.get(1, ‘Nope’)

‘a’
```

Note: .setdefault lets you do something similar when adding keys

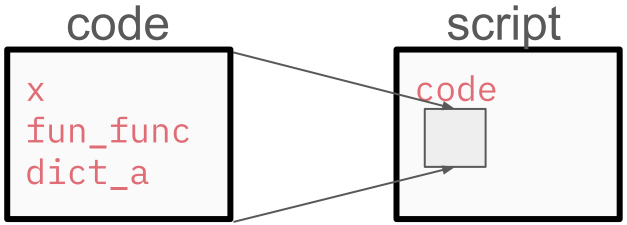
Modules

Folder containing multiple python scripts

```
CSCI_2202/

 -  script.py
 -  code.py
 -  my_module/
      -  stats.py
      -  micro.py
      -  physics/
           -  code.py
           -  answers.py
```

import lets us access functions (and objects) in other files

```python
# code.py

x = 10

def fun_func(y):

        z = y + 10

    return z

dict_a = {x: x + 1}
```



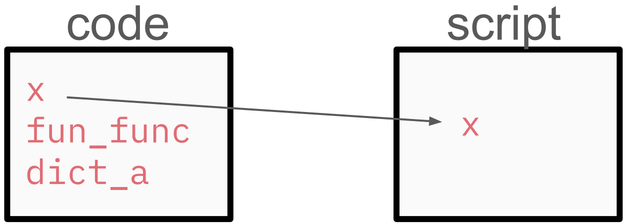

```python
# script.py (or notebook)

import code

print(code.x)

print(code.fun_func(30))

print(code.dict_a[code.x])
```

```
10

40

11
```

Import to .ipynb is easy but importing FROM a .ipynb is more complicated

import lets us access functions (and objects) in other files

```python
# code.py

x = 10

def fun_func(y):

        z = y + 10

    return z

dict_a = {x: x + 1}
```




```python
# script.py (or notebook)

from code import x

print(x)

print(code.x)

print(code.fun_func(10))
```

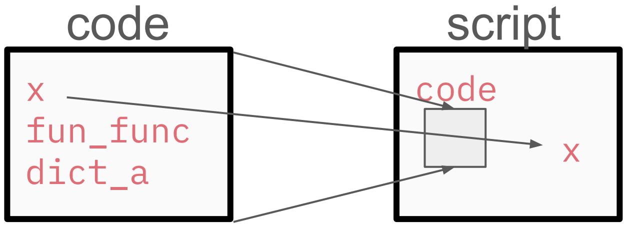
10

Error

Error

import lets us access functions (and objects) in other files

```
# code.py

x = 10

def fun_func(y):

        z = y + 10

    return z

dict_a = {x: x + 1}
```

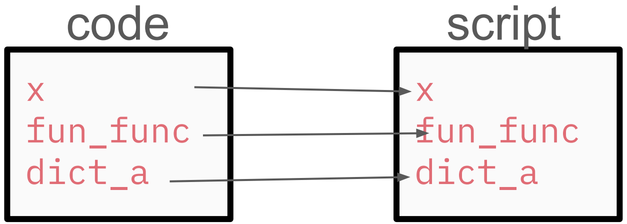



```
# script.py (or notebook)

from code import x

import code

print(x)

print(code.x)

print(code.fun_func(30))

print(code.dict_a[code.x])
```

```
10

10

40

11
```

import lets us access functions (and objects) in other files

```python
# code.py

x = 10

def fun_func(y):

        z = y + 10

    return z

dict_a = {x: x + 1}
```

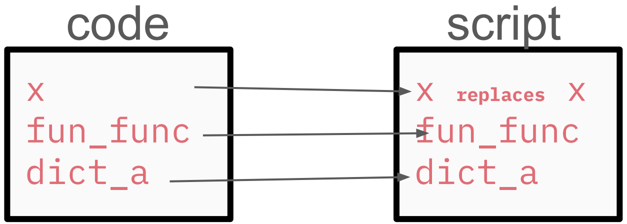



```python
# script.py (or notebook)

from code import *

print(x)

print(fun_func(30))

print(dict_a[x])

print(code.x)
```

```
10

40

11

ERROR
```

import lets us access functions (and objects) in other files

```python
# code.py

x = 10

def fun_func(y):

        z = y + 10

    return z

dict_a = {x: x + 1}
```

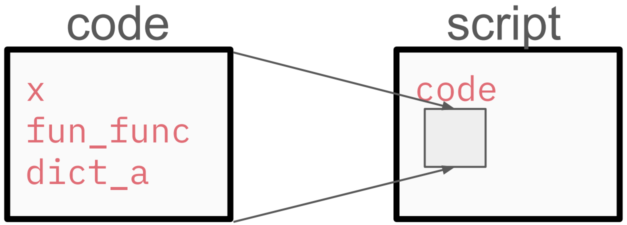



```python
# script.py (or notebook)

x = 50

from code import *

print(x)
```

10

dir function can be used to see what is in a module

```python
# code.py

x = 10

def fun_func(y):
	z = y + 10
    return z

dict_a = {x: x + 1}
```




```python
# script.py (or notebook)

import code

dir(code)

['__builtins__',
'__cached__', '__doc__',
'__file__', '__loader__',
'__name__', '__package__',
'__spec__', 'dict_a',
'fun_func', 'x']
```

PYTHONPATH - where python looks for modules/packages

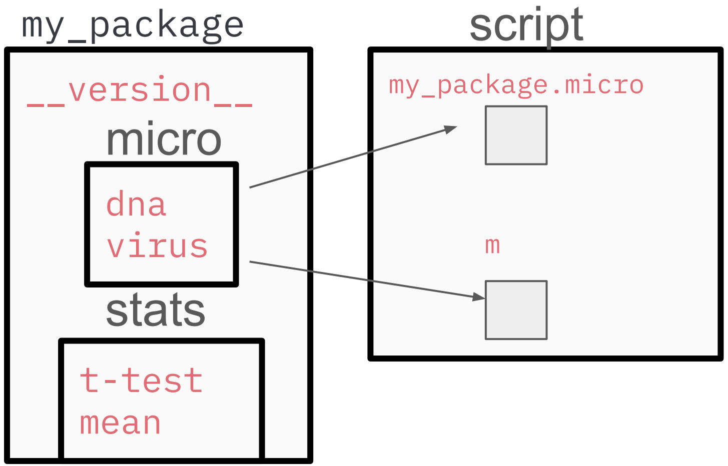
```
import sys

print(sys.path)

[current_folder, python_library,
python_libdynload,
python_sitepackages…]
```

- System variable that controls where python checks for packages
- Order matters - it will check list in order and stop if it finds the import
    - Our `code.py` will be imported instead of “`code`” in the standard library because it comes first
- Anaconda is managing this for you (via `conda`)
- You can edit this in script using `sys.path` but usually a BAD IDEA!

What if you have lots of modules?

Packages are made of modules - dotted import

- script.py
- my_package
    - __init__.py
    - micro.py
    - stats.py




```python
# script.py (or notebook)
import my_package.micro
print(my_package.micro.dna)
'agc'
import my_package.micro as m
print(m.is_virus(m.dna))
True
```

Packages - aliases keep code simpler!

- script.py
- my_package
    - __init__.py
    - micro.py
    - stats.py
    - physics/
        - __init__.py
        - astro.py

```
# script.py (or notebook)

import my_package.physics.astro

print(my_package.physics.astro.cosmo_const)


import my_package.physics.astro as astro

print(astro.cosmo_const)


from my_package.physics.astro import *

print(cosmo_const)
```

How does python keep track of things?
Namespaces

Namespaces map names to objects

```python
# code.py

x = 10

def fun_func(y):

        z = y + 10

    return z

dict_a = {x: x + 1}
```

code

```
x
fun_func
dict_a
```

code_v2

```
x
fun_func
dict_a
```

Namespaces are basically dictionaries

```python
code = {‘fun_func': fun_func,
    'x’: x,
    ‘Dict_a’: dict_a}
```

What is useful is we often have multiple namespaces in our code

```python
code_v2 = {‘fun_func': fun_func,
        'x’: x,
        ‘Dict_a’: dict_a}
```

Namespaces map names to objects

```python
# code.py

x = 10

def fun_func(y):

        z = y + 10

    return z

dict_a = {x: x + 1}
```

code

x
fun_func
dict_a

code_v2

x
fun_func
dict_a

Namespaces are basically dictionaries

```python
code = {'fun_func': fun_func,
        'x': x,
        'Dict_a': dict_a}
```

What is useful is we often have multiple namespaces in our code

```python
code_v2 = {'fun_func': fun_func,
           'x': x,
           'Dict_a': dict_a}
```

Namespaces let us use the same variable name to mean different things.

functions, loops, etc can create new namespaces

```
i = 50

def func(i):

    i = 10

    return i


j = func(i)

print(i, j)


50, 10
```

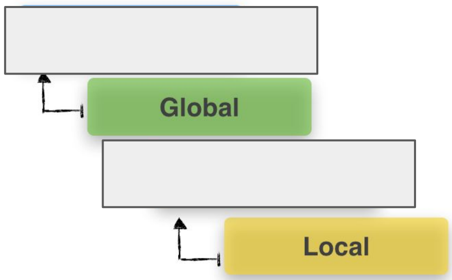
```
top_namespace = {'i': obj1, #50

                 'j': obj2, #10

                'func': func}

func_namespace = {'i': obj4} # 10
```

Namespaces form a hierarchy.

The “Scope” of a bit of code determines which level of this hierarchy it searches for namespace mapping

Scopes: search local namespace first

```python
i = 50
def func(i):
    i = 10
    return i

j = func(i)
print(i, j)

50, 10
```

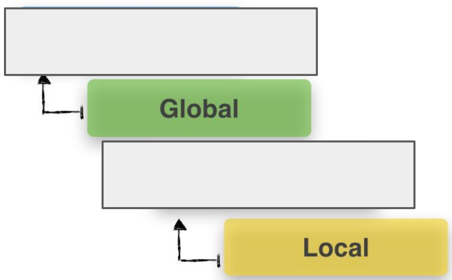



```python
global = {'i': obj1, #50
          'j': obj2, #10
          'func': func}
local = {'i': obj4} # 10
```

Error if you try to change a variable in different namespace

```
x = 1

def func():

    x = x - 5

    print(x, '[ x inside func() ]')


func()
```

```
UnboundLocalError: local variable x
referenced before assignment
```

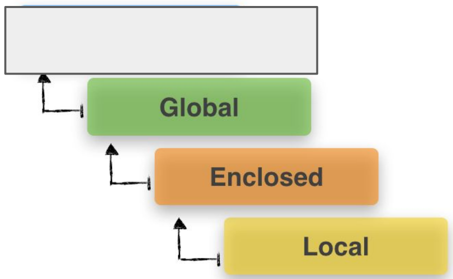


Note: there are ways to force python to do this but it is usually a bad idea

Nesting can add additional enclosed scopes

```
a_var = 'global value'
def outer():
    a_var = 'enclosed value'
    def inner():
        a_var = 'local value'
        print(a_var)
    inner()
outer()
```

```
local value
```




```
global = {‘a_var’: ‘global value’, ‘outer’: outer}
```

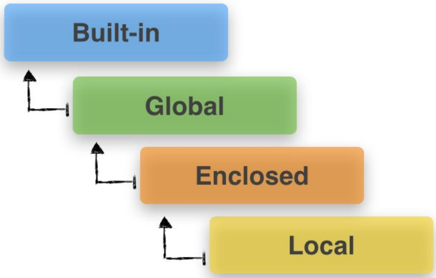
```
enclosed = {‘a_var’: ‘enclosed value’, ‘inner’: inner}
```

```
local = {‘a_var’: ‘local value’}
```

Nesting can add additional enclosed scopes

```
def len(in_var):
    print('my len() function')
    l = 0
    for i in in_var:
        l += 1
    print(l)
def a_func(in_var):
    len_in_var = len(in_var)
    print(len_in_var)
a_func('Hello, World!')
```

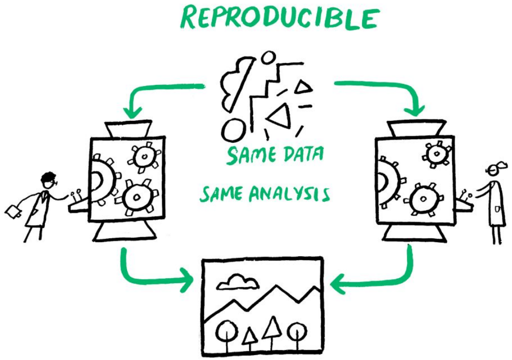
```
my len() function
13
```



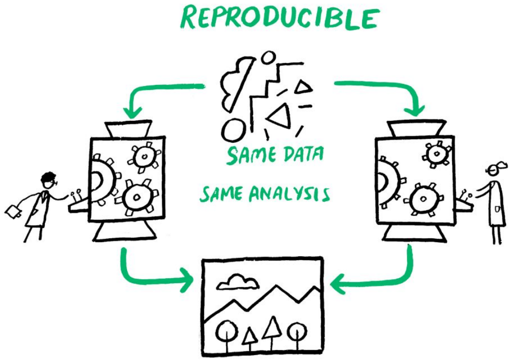

```
Built-in = {'len': len, …}

global = {'len': len, 'a_func': a_func}

local_len = {'l': l, 'i': i, 'in_var':
in_var}

Local_a_func = {'len_in_var': len_in_var,
               'In_var': in_var}
```

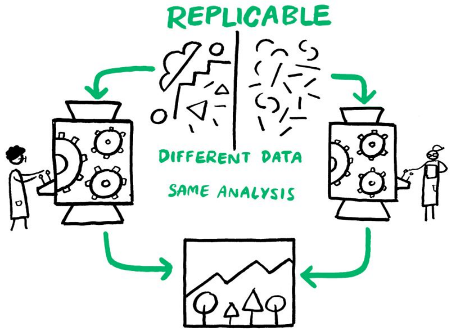
Scientific analyses must reproducible

Reproducibility should be the bare minimum

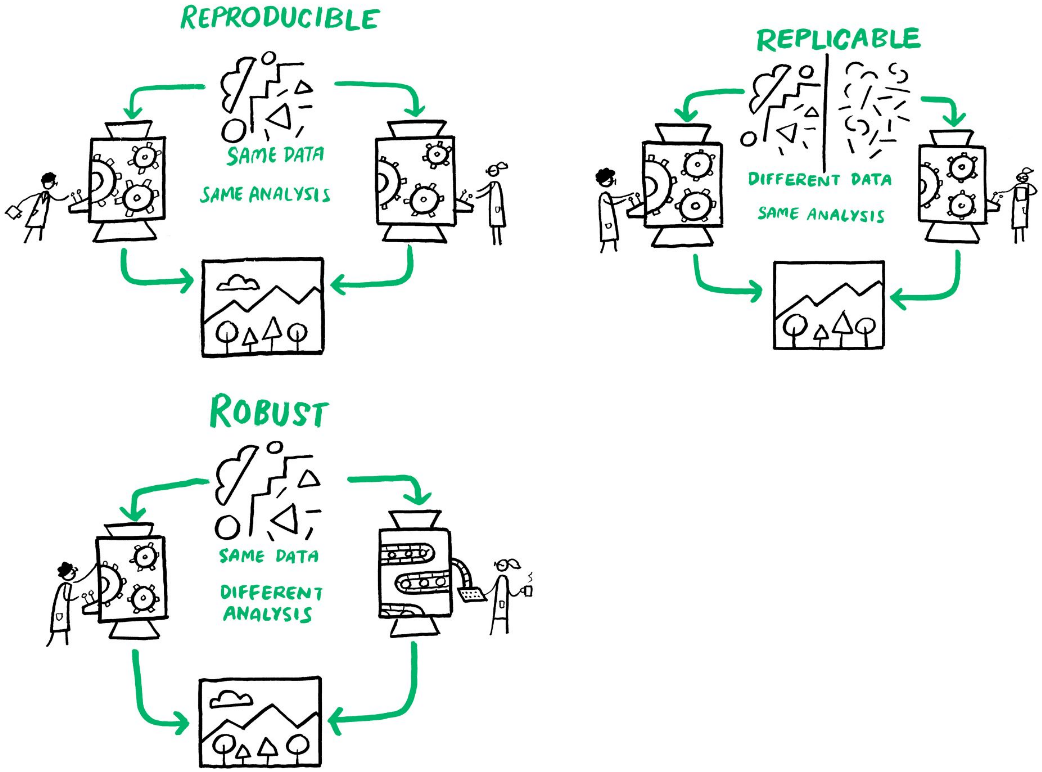


Reproducibility should be the bare minimum



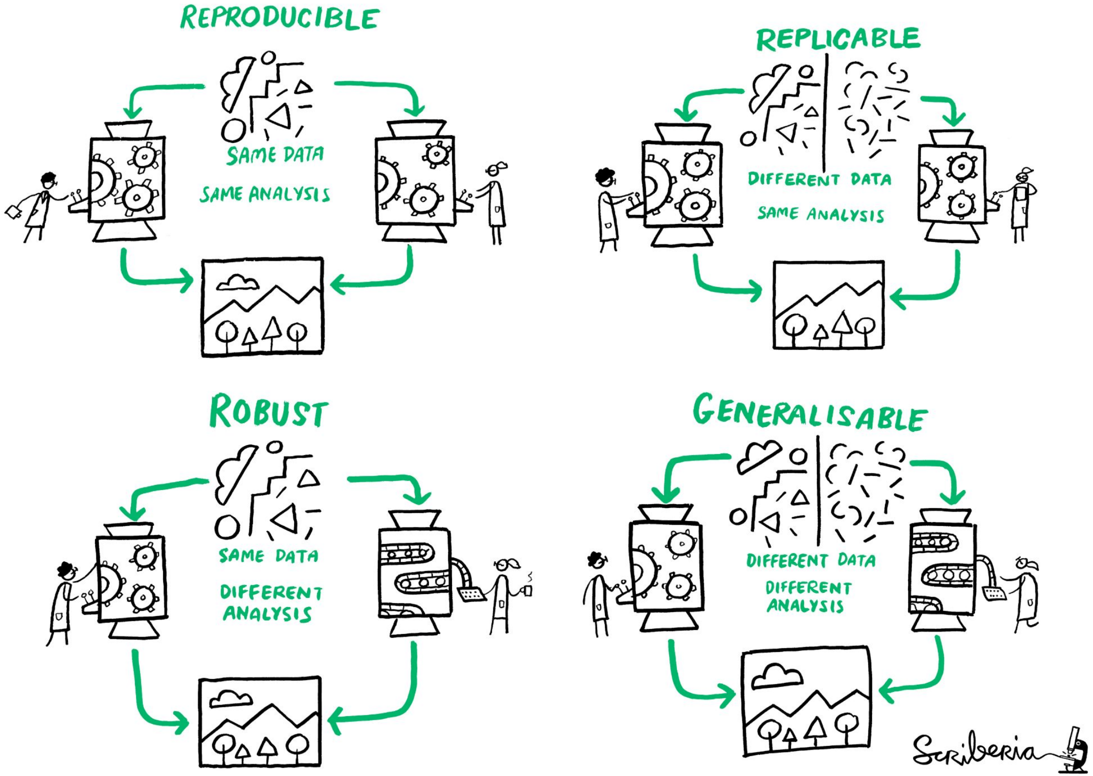


Reproducibility should be the bare minimum



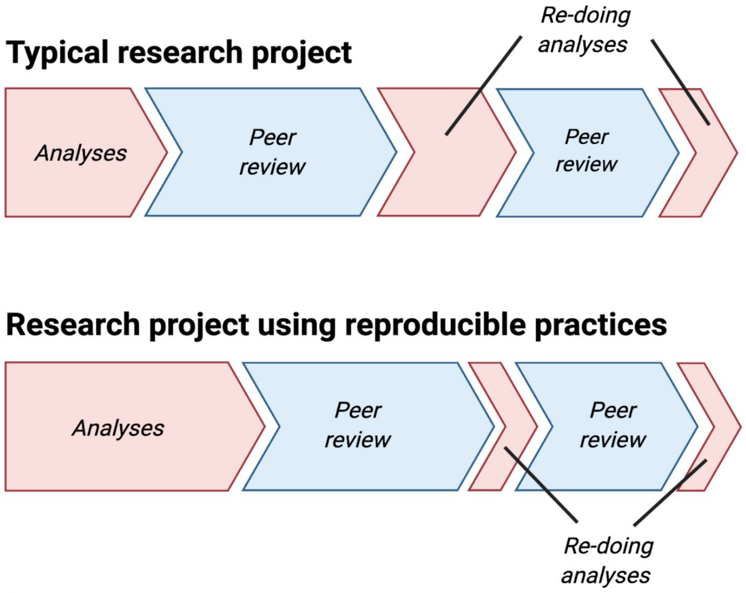
oliviergimenez.github.io/reproducible-science-workshop

Reproducibility should be the bare minimum



oliviergimenez.github.io/reproducible-science-workshop

Makes your own life easier



@dsquintana

oliviergimenez.github.io/reproducible-science-workshop

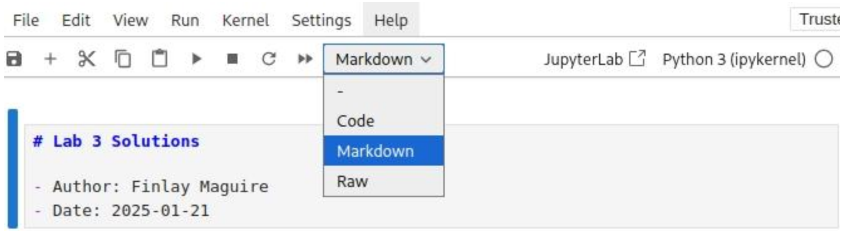
What do we need to do to have reproducible research?

Reproducibility checklist

- Don’t do anything by hand (even “one-off” tasks)
- Script every interaction with data:
  - Data collection
  - Moving data on your computer
  - Formatting datasets
  - Cleaning data
  - Exploratory data analysis
  - Main analyses
  - Report generation
- Minimise interactivity/point and click interactions
- Keep track of the exact version of every library/program you use
- <u>*Version control all data, code, and documentation*</u>
- <u>*Use a random seed*</u>

Notebooks are a key tool for doing this

Jupyter supports markdown OR code cells



Markdown is a quick way to indicate how to format plaintext

Converts to HTML in background

Supported by many developer tools

`# Heading level 1`	`<h1>Heading level 1</h1>`	Heading level 1

Heading	`# H1`
	`## H2`
	`### H3`

Bold	`**bold text**`
Italic	`*italicized text*`
Blockquote	`> blockquote`

Ordered List	`1. First item`
	`2. Second item`
	`3. Third item`

Unordered List	`- First item`
	`- Second item`
	`- Third item`

Code	`` `code` ``
Horizontal Rule	`---`
Link	`[title](https://www.example.com)`
Image	`![alt text](image.jpg)`

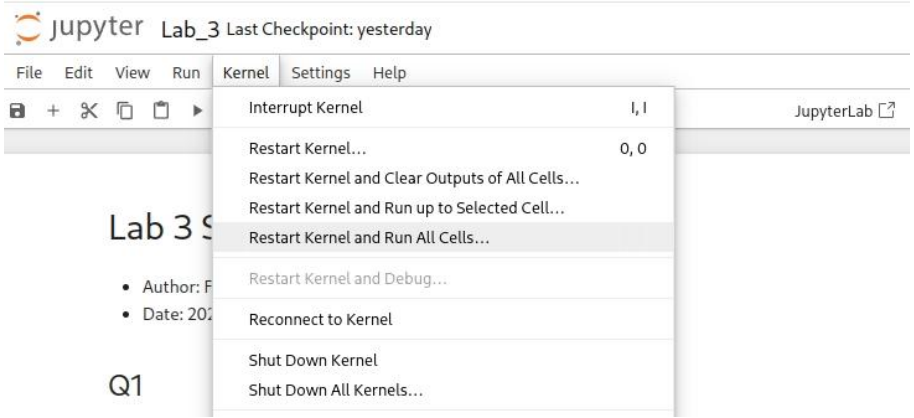
Out of order cell-execution can lead to bugs and errors

```
[3] y = x(5)
```

```
[1] def x(x):
      return x * 2
```

```
[2] y = x(100)
```

```
[4] y
10
```



Always run this at the end of a notebook - the notebook on the left will error now!

Making a good notebook

Structure your Notebook

- Give your notebook a **title** (H1 header) and a meaningful **preamble** to describe its purpose and contents.
- Use **headings and documentation** in Markdown cells to structure your analysis and explain your steps.

Refactor & outsource code into modules

- After you've written plain code in cells to get ahead quickly, acquire the habit of **turning stable code into functions** and **move them to a dedicated module**. This makes your notebook more readable and is incredibly helpful when productionizing your workflow. Following is clearer and easier to test than repeating the same code many times in your notebook.

```
import dataprep

df = dataprep.load_and_preprocess_data(filename)
```

- **Stick to the standards of good coding** — Standardise your formatting, use meaningful variable and function names, comment sensibly, modularize your code and don't be too lazy to refactor.

Overview

- Several handy list functions (range, zip) use lazy evaluation
- While loops enable easy conditional loops
- Dictionaries store key -> value pairs
- Reusing/organising code can be done using modules & packages
- Namespaces are how python keeps track of variables
- Reproducibility makes for better science
- Creating clean, well-documented, notebooks that run the cells in order is useful for this.