Lecture 05:

Functions

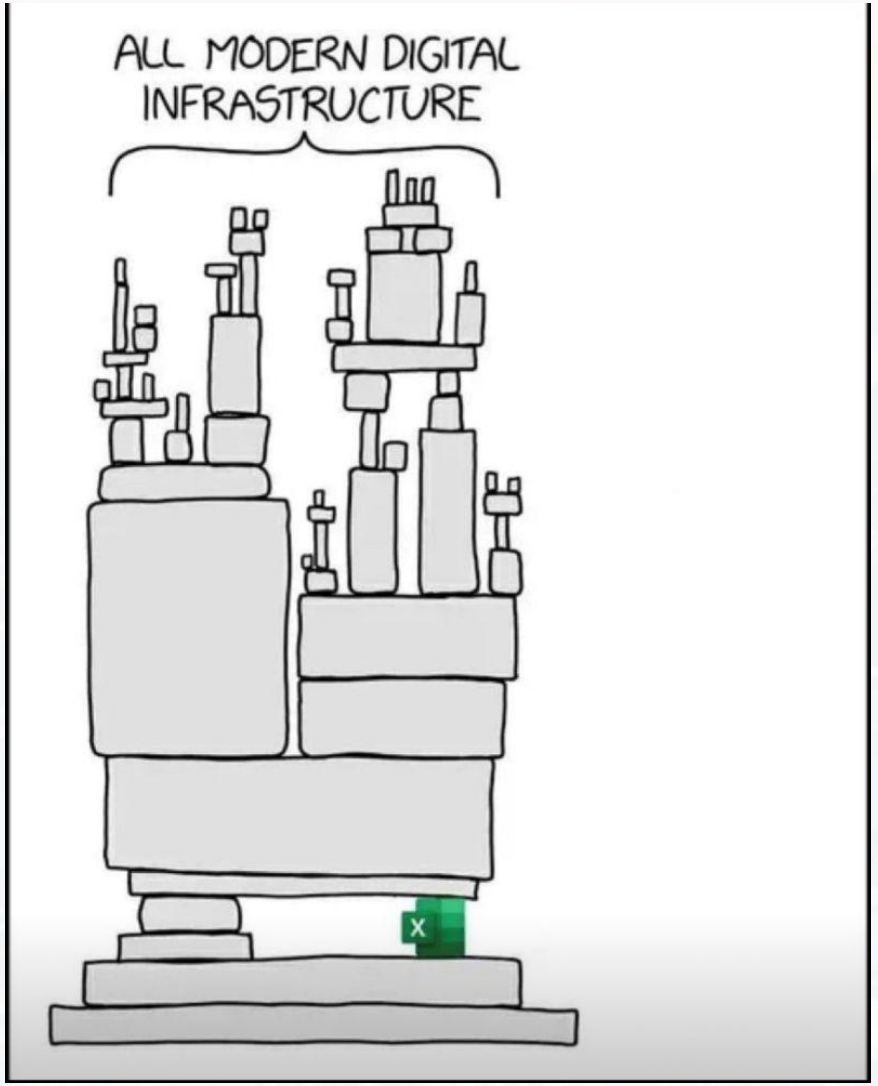
Markus Hohle

University California, Berkeley

Python for Molecular Sciences

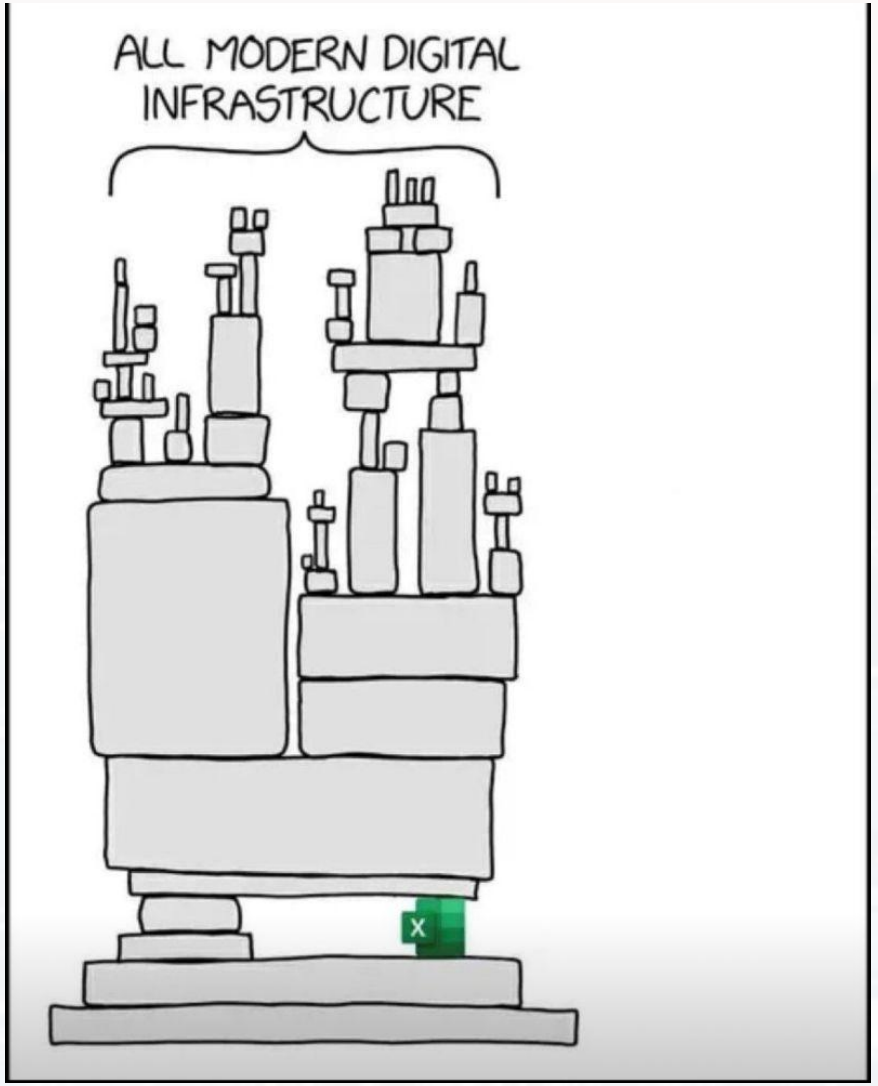
MSSE 272, 3 Units

Berkeley MSSE
MOLECULAR SCIENCE & SOFTWARE ENGINEERING



Outline

- `lambda`
- `map`
- `def`
- `*args and **kwargs`



Outline

- `lambda`
- `map`
- `def`
- `*args and **kwargs`

Functions

recall loops (**comprehension**)

```
from math import *

Factorial = [factorial(n) for n in range(10)]
```

Index	Type	Size	
0	int	1	1
1	int	1	1
2	int	1	2
3	int	1	6
4	int	1	24
5	int	1	120
6	int	1	720
7	int	1	5040
8	int	1	40320
9	int	1	362880

```
NT    = 'ACGT'
Code = [[1,0,0,0],
        [0,1,0,0],
        [0,0,1,0],
        [0,0,0,1]]

Dict = {nt: code for code, nt in zip(Code, NT)}

Dict['A']
```

```
In [2]: Dict['A']
Out[2]: [1, 0, 0, 0]
```

What if we want to encode an entire sequence, say *'ACGGTCCGACCT'*?

```
NT    = 'ACGT'
Code = [[1,0,0,0],
        [0,1,0,0],
        [0,0,1,0],
        [0,0,0,1]]

Dict = {nt: code for code, nt in zip(Code, NT)}
```

one possibility: loop

```
S = 'ACGGTCCGACCT'
L = []

for s in S:
    L += [Dict[s]]
```

```
[[1, 0, 0, 0],
 [0, 1, 0, 0],
 [0, 0, 1, 0],
 [0, 0, 1, 0],
 [0, 0, 0, 1],
 [0, 1, 0, 0],
 [0, 1, 0, 0],
 [0, 0, 1, 0],
 [1, 0, 0, 0],
 [0, 1, 0, 0],
 [0, 1, 0, 0],
 [0, 0, 0, 1]]
```

What if we want to encode an entire sequence, say *'ACGGTCCGACCT'*?

However, we wanted to avoid loops as much as possible

benchmarking the loop:

However, we wanted to avoid loops as much as possible

```
import time


t1 = time.monotonic()

for i in range(100000):
    L = []
    for s in S:
        L += [Dict[s]]

t2 = time.monotonic()
dt = t2 - t1


print("Total runtime: " + str(dt) + ' seconds')
```

total runtime: 0.14s

Functions


```
S = 'ACGGTCCGACCT'
L = []

for s in S:
    L += [Dict[s]]
```

alternative: `lambda`

```
Encode = lambda Seq: [Dict[s] for s in Seq]
```

defines the sequence of commands the function has to execute

the input argument (name is arbitrary)

defining a function, we name Encode using the keyword `lambda`

Functions


```
S = 'ACGGTCCGACCT'
L = []

for s in S:
    L += [Dict[s]]
```

alternative: lambda

```
Encode = lambda Seq: [Dict[s] for s in Seq]
```

```
Encode(S) [[1, 0, 0, 0],
 [0, 1, 0, 0],
 [0, 0, 1, 0],
 [0, 0, 1, 0],
 [0, 0, 0, 1],
 [0, 1, 0, 0],
 [0, 1, 0, 0],
 [0, 0, 1, 0],
 [1, 0, 0, 0],
 [0, 1, 0, 0],
 [0, 1, 0, 0],
 [0, 0, 0, 1]]
```

Once, we have defined Encode, we **don't need to run the loop** for every new sequence S1, S2, … we just call the function

```
Encode(S1)
Encode(S2)
Encode(S3)
```

benchmarking `lambda`:

```
t1 = time.monotonic()

for i in range(100000):
    E = Encode(S)

t2 = time.monotonic()

dt = t2 - t1
print("Total runtime: " + str(dt) + ' seconds')
```

total runtime: **0.06s**

vs loop total runtime: **0.14s**

Functions

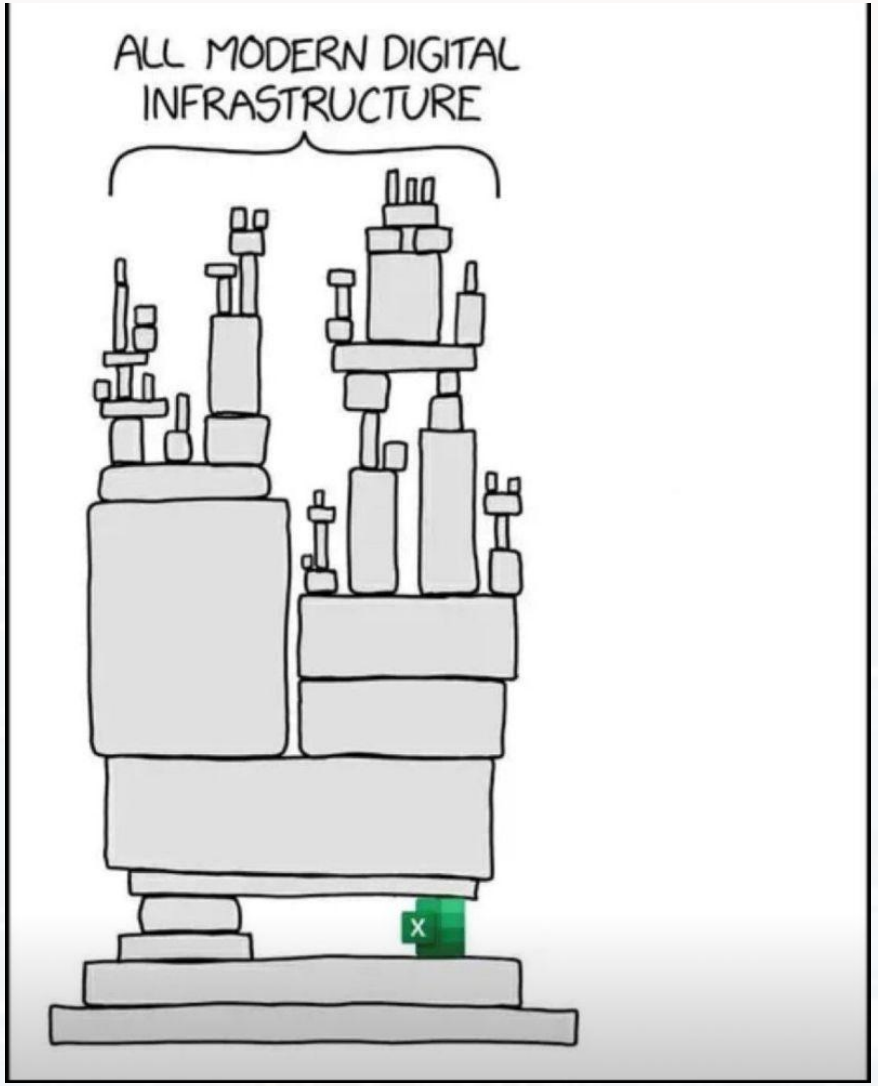

```
NT    = 'ACGT'
Code = [[1,0,0,0],
        [0,1,0,0],
        [0,0,1,0],
        [0,0,0,1]]

S = 'ACGGTCCGACCT'

Dict = {nt: code for code, nt in zip(Code, NT)}
```

```
L = []

for s in S:
    L += [Dict[s]]
```

versus

```
Encode = lambda Seq: [Dict[s] for s in Seq]

Encode(S)
```

- **faster** (factor of 2)
- has to be defined **only once**
- named **“anonymous function”** (not stored in an extra file)



Outline

- lambda
- **map**
- def
- *args and **kwargs

Functions

Now we want to run **multiple sequences** most efficiently

```
S1 = 'ACGGTCCGACCT'
S2 = 'TTCAGGT'
S3 = 'ATCGGCAATCTGCTTCA'
S4 = 'CGTCCGTA'

Encode = lambda Seq: [Dict[s] for s in Seq]
```

as before, we could run a loop

```
S = [S1, S2, S3, S4]

L = []

for s in S:
    L += [Encode(s)]
```

Index	Type	Size	Value
0	list	12	[[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 1, 0], [0,…
1	list	7	[[0, 0, 0, 1], [0, 0, 0, 1], [0, 1, 0, 0], [1, 0, 0, 0], [0,…
2	list	17	[[1, 0, 0, 0], [0, 0, 0, 1], [0, 1, 0, 0], [0, 0, 1, 0], [0,…
3	list	8	[[0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1], [0, 1, 0, 0], [0,…

Functions

Now we want to run **multiple sequences** most efficiently

```
S1 = 'ACGGTCCGACCT'
S2 = 'TTCAGGT'
S3 = 'ATCGGCAATCTGCTTCA'
S4 = 'CGTCCGTA'

Encode = lambda Seq: [Dict[s] for s in Seq]
```

a more efficient way is using map

```
S = [S1, S2, S3, S4]

L = list(map(Encode, S))
```

`map` needs a **function** (here Encode) as first input argument and an **iteratable** - over which the function runs - as a second argument

finally, turning the map object into a list

Functions

Now we want to run **multiple sequences** most efficiently

```
S1 = 'ACGGTCCGACCT'
S2 = 'TTCAGGT'
S3 = 'ATCGGCAATCTGCTTCA'
S4 = 'CGTCCGTA'

Encode = lambda Seq: [Dict[s] for s in Seq]
```

a more efficient was is using map

```
S = [S1, S2, S3, S4]

L = list(map(Encode, S))
```

Index	Type	Size	Value
0	list	12	[[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 1, 0], [0,…
1	list	7	[[0, 0, 0, 1], [0, 0, 0, 1], [0, 1, 0, 0], [1, 0, 0, 0], [0,…
2	list	17	[[1, 0, 0, 0], [0, 0, 0, 1], [0, 1, 0, 0], [0, 0, 1, 0], [0,…
3	list	8	[[0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1], [0, 1, 0, 0], [0,…

finally, we benchmark the **complete loop vs `lambda`/`map`**:

```
S1 = 'ACGGTCCGACCT'
S2 = 'TTCAGGT'
S3 = 'ATCGGCAATCTGCTTCA'
S4 = 'CGTCCGTA'

S = [S1, S2, S3, S4]
```

```
t1 = time.monotonic()

for i in range(100000):
    for s in S:
        L = []
        for nt in s:
            L += [Dict[nt]]

t2 = time.monotonic()
dt = t2 - t1

print("Total runtime: " + str(dt) + ' seconds')
```

total runtime: 0.58s

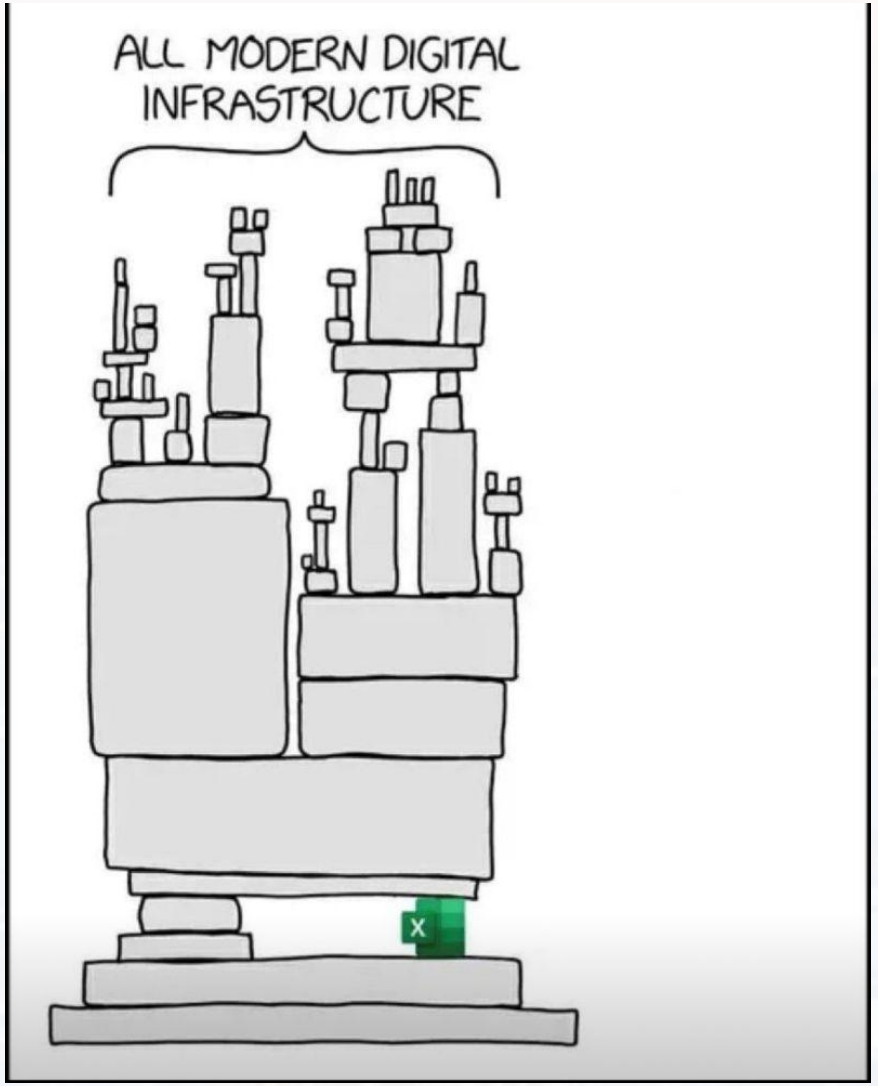
finally, we benchmark the **complete loop vs `lambda`/`map`**:

```
S1 = 'ACGGTCCGACCT'
S2 = 'TTCAGGT'
S3 = 'ATCGGCAATCTGCTTCA'
S4 = 'CGTCCGTA'

S = [S1, S2, S3, S4]
```

```
t1 = time.monotonic()

for i in range(100000):

    L = list(map(Encode, S))


t2 = time.monotonic()
dt = t2 - t1

print("Total runtime: " + str(dt) + ' seconds')
```

total runtime: 0.22s

We saved two nested loops and it is almost 3x faster!

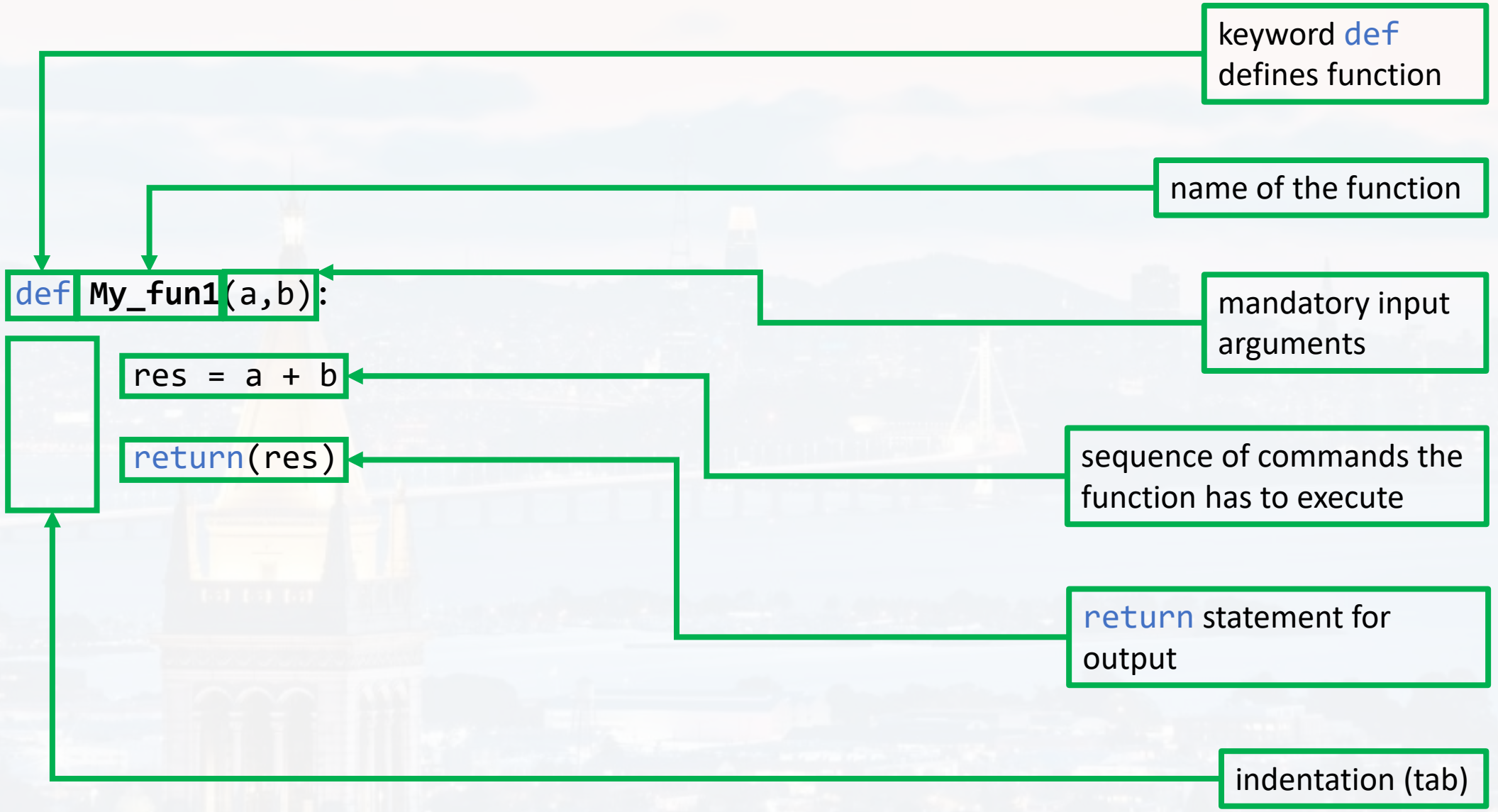


Outline

- lambda
- map
- **def**
  - *args and **kwargs

Functions

actual functions/methods in Python:



Functions

actual functions/methods in Python:

```
def My_fun1(a,b):
    res = a + b
    return(res)
```

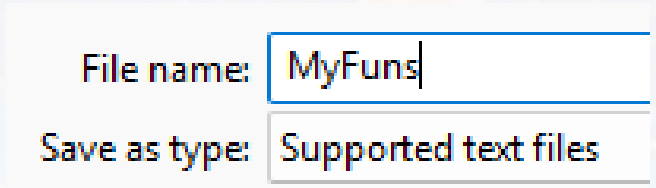
also called **body** of the function

Functions

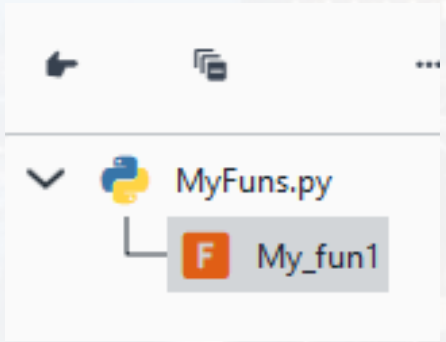
actual functions/methods in Python:

```
def My_fun1(a,b):

        res = a + b

        return(res)
```

```
 8  def My_fun1(a,b):
 9      
10      res = a + b
11
12      return(res)
```



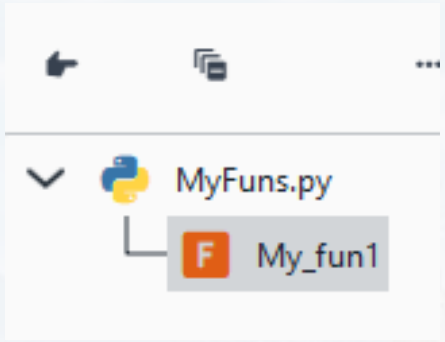
saving the `.py` script that contains `My_fun1` as `MyFuns.py`



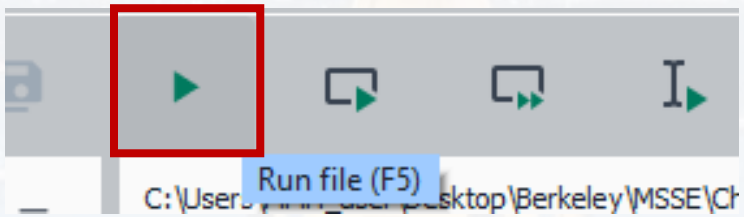
the `.py` script `MyFuns.py` contains `My_fun1`

Functions

actual functions/methods in Python:



the `.py` script `MyFuns.py` contains `My_fun1`



compiling the function (green arrow)

```
In [1]: runfile('C:/[redacted]/Berkeley/MSSE/Chem 272/05
Functions/MyFuns.py', wdir='C:/[redacted]/Berkeley/MSSE/Chem
272/05 Functions')
```

Functions


```
In [3]: My
        My_fun1
        MyFuns.py
```

check via autocomplete if function is recognized

```
In [3]: My_fun1(
        My_fun1(a, b)

        No documentation available
```

autocomplete tells us that we need two mandatory input arguments: a and b

```
R = My_fun1(12, 10)
```

running the function and saving the output as variable R

```
In [4]: print(R)
22
```

Functions

what can go wrong:

```
def My_fun1(a,b):
    res = a + b
    return(res)
```

too **many** input arguments

```
In [6]: R = My_fun1(12,10,4)
Traceback (most recent call last):

  Cell In[6], line 1
    R = My_fun1(12,10,4)

TypeError: My_fun1() takes 2 positional arguments but 3 were given
```

too **few** input arguments

```
In [8]: R = My_fun1(12)
Traceback (most recent call last):

  Cell In[8], line 1
    R = My_fun1(12)

TypeError: My_fun1() missing 1 required positional argument: 'b'
```

Functions

def

what can go wrong:

confusing **calling** the function vs **saving the function as a new variable**

```
R = My_fun1
```

```
def My_fun1(a,b):
    res = a + b
    return(res)
```

```
In [11]: print(R)
<function My_fun1 at 0x000001598 37C3100>
```

function My_fun1 is **copied** to R

→ will not prompt an error message

```
In [13]: type(R)
Out[13]: function
```

```
In [12]: R(2,3)
Out[12]: 5
```

Functions

what can go wrong:

confusing **calling** the function vs **saving the function as a new variable**

```
R = My_fun1()
```

function `My_fun1` needs no input argument

→ returns output to R

```
def My_fun1(a,b):
    res = a + b
    return(res)
```

Functions

what can go wrong:

```
def My_fun1(a,b):
    res = a + b
    return(res)
```

when output is not stored in a new variable → generates output in console

```
In [15]: My_fun1(12,10)
Out[15]: 22
```

calling **more output** variables then provided by function

```
[R1, R2] = My_fun1(12, 10)
```

```
In [16]: [R1, R2] = My_fun1(12,10)
Traceback (most recent call last):

  Cell In[16], line 1
    [R1, R2] = My_fun1(12,10)

TypeError: cannot unpack non-iterable int object
```

Functions

generating multiple outputs

```
def My_fun1(a,b):

        res1 = a + b
        res2 = a * b
        res3 = a**b

        return res1, res2, res3
```

save and compile…

```
[R1, R2, R3] = My_fun1(5,6)
```

Nam	Type	Size	
R1	int	1	11
R2	int	1	30
R3	int	1	15625

Functions

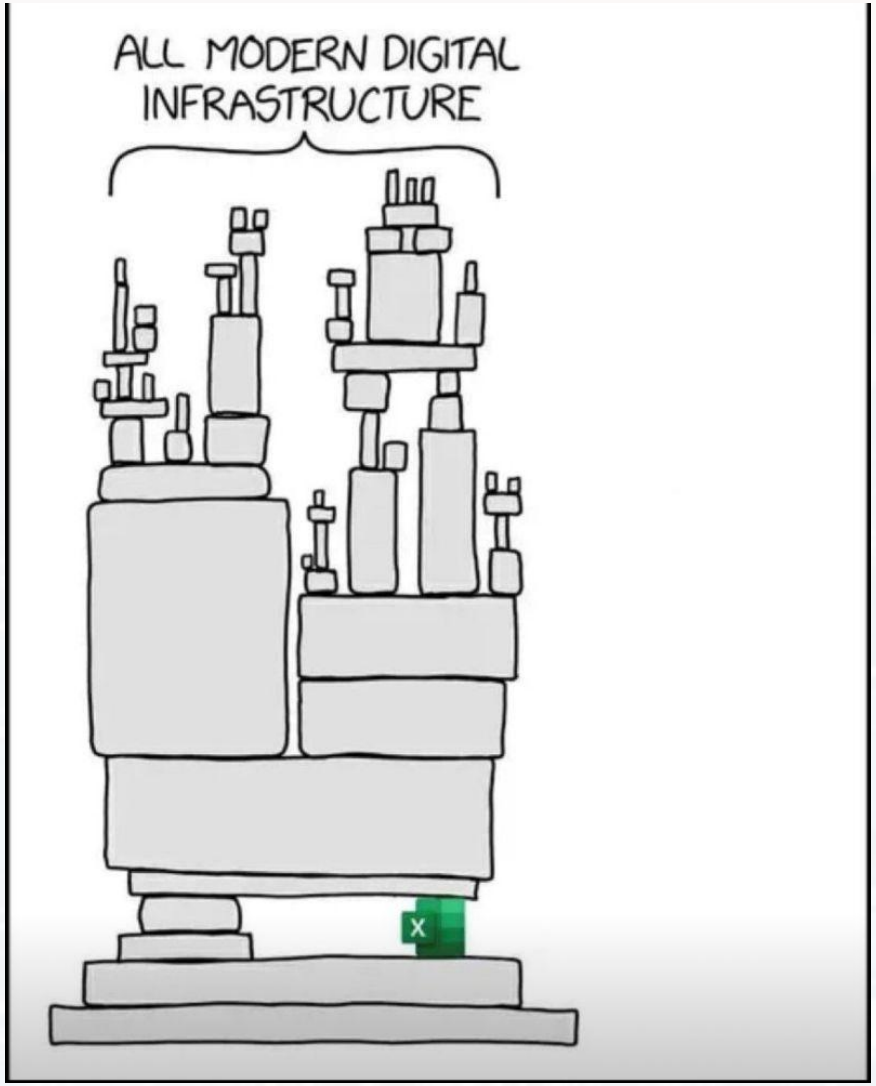
generating multiple outputs

```
def My_fun1(a,b):

        res1 = a + b
        res2 = a * b
        res3 = a**b

        return res1, res2, res3
```

sometimes a function generates more output than we need
→ using _ for suppressing specific output

```
[_, R2, _] = My_fun1(5,6)
```

Nam	Type	Size	
R2	int	1	30

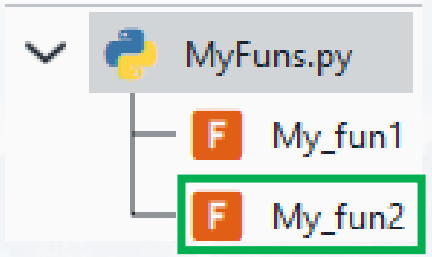


Outline

- lambda
- map
- def
- ***args and **kwargs**

default arguments → specific value unless stated otherwise

```
def My_fun2(a, b = 2):

        res1 = a + b
        res2 = a * b
        res3 = a**b

        return res1, res2, res3
```



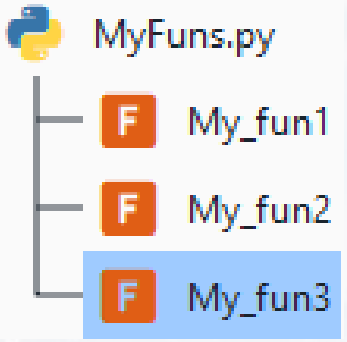

```
In [26]: My_fun2(5)
Out[26]: (7, 10, 25)
```

```
In [27]: My_fun2(5,3)
Out[27]: (8, 15, 125)
```

optional ***args** arguments → when needed for specific settings

```
def My_fun3(a, b = 2, *power):

        if power:
                print(type(power))

        res1 = a + b
        res2 = a * b
        res3 = a**b

        return res1, res2, res3

[R1, R2, R3] = My_fun3(1, 2, 3)
```

```
<class 'tuple'>
```



Why is it a tuple?

optional *args arguments → when needed for specific settings

Why is it a tuple?

We switch to a better example:

```
def BuildSentences(*words):

        Sentence = ''

        for w in words:
                Sentence += ' ' + w

        print(Sentence)
```

iterating over the tuple words

adding as many words as given as input

```
In [41]: BuildSentences('This', 'is', 'a', 'sentence', '!')
 This is a sentence !
```

*args → as many input arguments as we want!

better solution here:

```
def My_fun3(a, b = 2, power = 1):

        res1 = a**power + b**power
        res2 = a * b
        res3 = a**b

        return res1, res2, res3
```

optional keyword ****kwargs** arguments

Let us first check the type:

```
def KWargExample(**test):

        print(type(test))


KWargExample(test = 'this is a test')
```

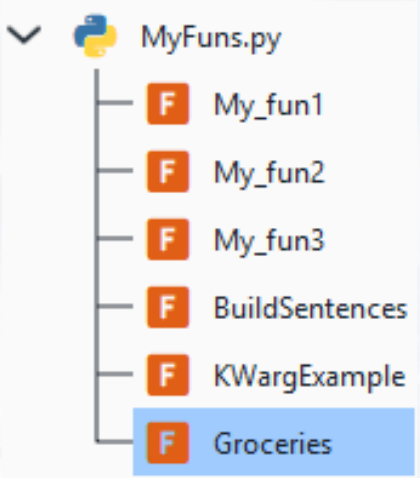
```
In [46]: KWargExample(test = 'this is a test')
<class 'dict'>
```

Why is it a dictionary?

optional keyword ****kwargs** arguments

Why is it a dictionary?

```
def Groceries(**items):

        print(items)
```




```
Groceries(butter = '250g', beer = 20,
          sausage = 'salami', wine = 'red')
```

**kwargs → as many input keyword arguments as we want!

```
{'butter': '250g', 'beer': 20, 'sausage': 'salami', 'wine': 'red'}
```

optional keyword **kwargs arguments

Why is it a dictionary?

```
def Groceries(**items):

        print(items)



Groceries(butter = '250g', beer = dict(IPA = 20, WheatBeer = 5),
          sausage = 'salami', wine = 'red')
```

```
{'butter': '250g', 'beer': {'IPA': 20, 'WheatBeer': 5}, 'sausage': 'salami',
'wine': 'red'}
```

examples in python:

```
plt.plot(
```

```
plot(*args: 'float | ArrayLike | str', scalex: 'bool' =
     True, scaley: 'bool' = True, data=None, **kwargs,)

Plot y versus x as lines and/or markers.

Call signatures::

plot([x], y, [fmt], *, data=None, **kwargs)
plot([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
...
```

```
plt.plot(x_plot, y_plot, 'k-', linewidth = 3, alpha = 0.5)
```

plotting x vs y: mandatory argument

optional: line color ('k' = black) and line style ('-' = solid line)

key word arguments

Thank you very much for your attention