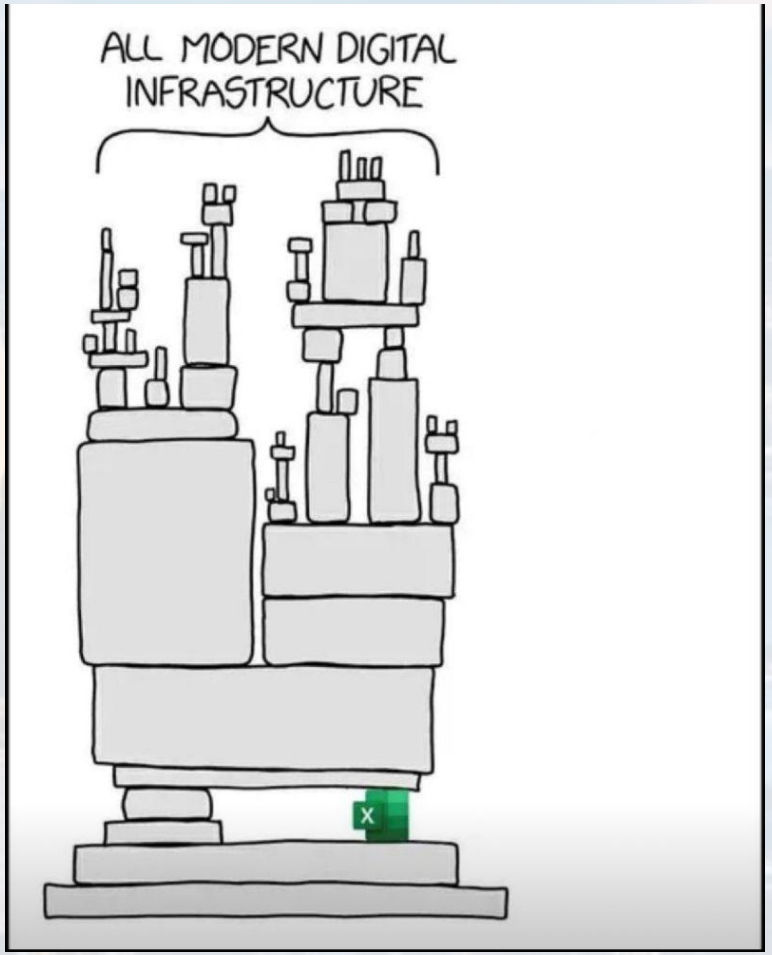
Berkeley
UNIVERSITY OF CALIFORNIA

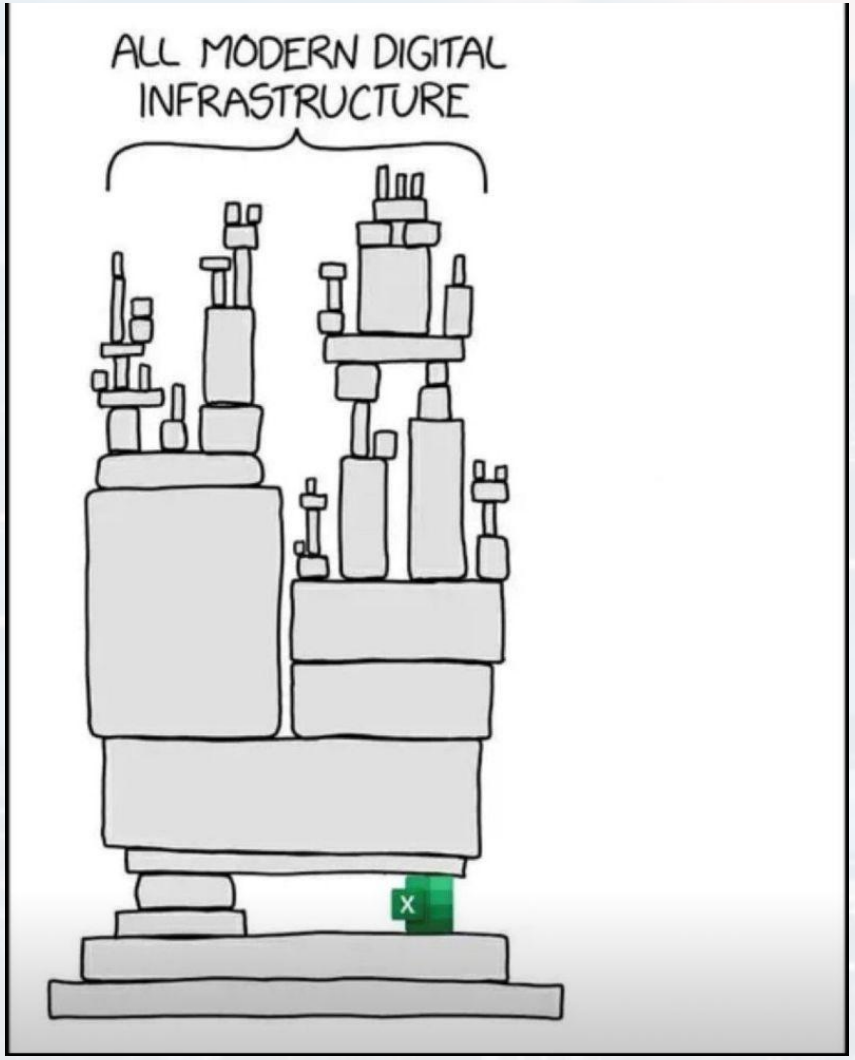
M. Hohle:

Physics 77: Introduction to Computational Techniques in Physics



Introduction to Computational Techniques in Physics: syllabus

Week	Date	Topic
1	June 12th	Programming Environment & UIs for Python, Programming Fundamentals
2	*June 19th*	Basic Types in Python
3	June 26th	Parsing, Data Processing and File I/O, Visualization
4	**July 3rd**	**Functions, Map & Lambda**
5	July 10th	Random Numbers & Probability Distributions, Interpreting Measurements
6	July 17th	Numerical Integration and Differentiation
7	July 24th	Root finding, Interpolation
8	July 31st	Systems of Linear Equations, Ordinary Differential Equations (ODEs)
9	Aug 7th	Stability of ODEs, Examples
10	Aug 14th	Final Project Presentations



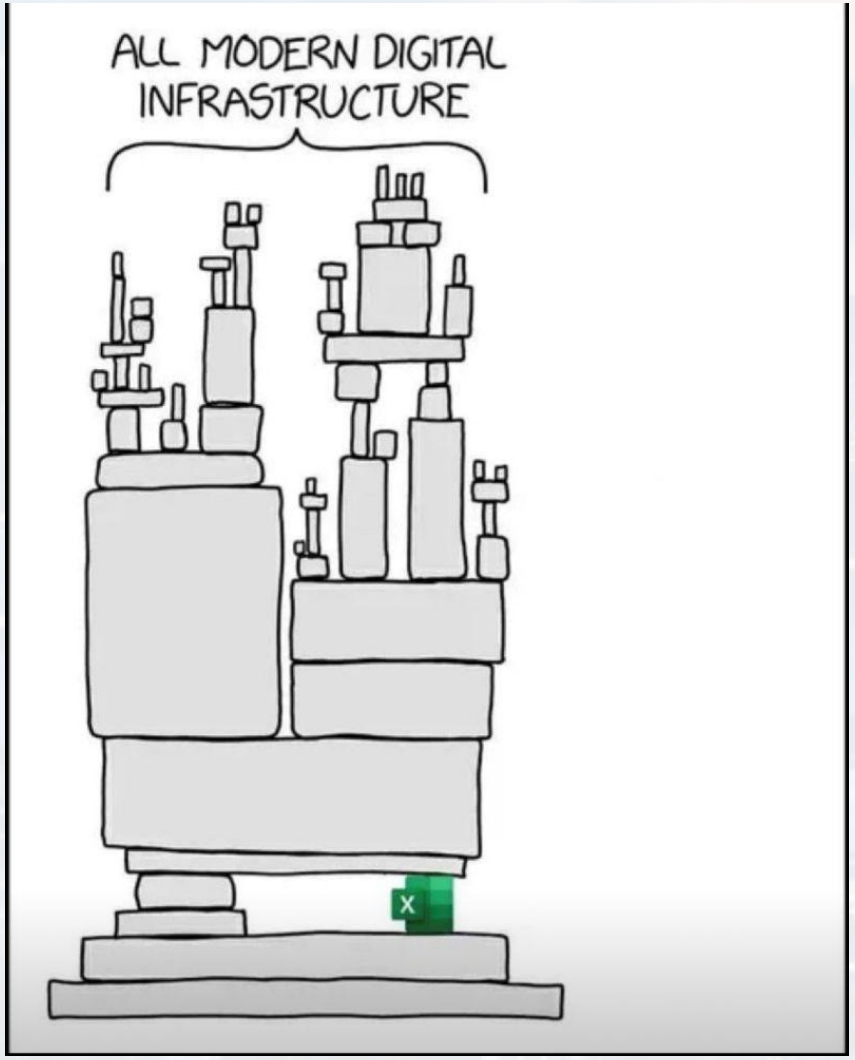
Outline

Control Structures

- General Idea and Structure
- `for` Loops and Comprehension
- `if`, `else` and `elif`
- `while`
- `break`, `continue` and `pass`

Functions

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



Outline

Control Structures

- General Idea and Structure

- `for` Loops and Comprehension
- `if`, `else` and `elif`
- `while`
- `break`, `continue` and `pass`

Functions

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

often (not always): iterating over an object

	numeric: `int, float, complex`	`5, 5.55, (5+5j)`
iteratable	strings: `str`	`'this is a string', "this is a string"`
	sequence: `list, tuple, range`	`my_tuple = (3, 'a', [2,3,4,5])` `range(10)`
mutable		`my_list = [1, 2, 'a']`
	mapping: `dict`	`my_dict = {1: 'a', 2: 'b'}`
	mapping: `set`	`my_set = {1, 2, 'a'}`
	boolean:	`True False`
	none type:	`None`
	callable: functions, methods, classes	`def, class, map, lambda`
	modules:	`from my_module import my_method as my_alias`

often (not always): iterating over an object

iteratable

strings: `str` — `'this is a string', "this is a string"`

sequence: `list, tuple, range`

```
my_tuple = (3, 'a', [2,3,4,5])
range(10)
```

mutable

```
my_list  = [1, 2, 'a']
```

mapping: `dict`

```
my_dict  = {1: 'a', 2: 'b'}
```

mapping: `set`

```
my_set   = {1, 2, 'a'}
```

often (not always): iterating over an object

iteratable

strings: `str`	`'this is a string', "this is a string"`

sequence: `list`, `tuple`, `range`	`my_tuple = (3, 'a', [2,3,4,5])`
	`range(10)`
	`my_list = [1, 2, 'a']`

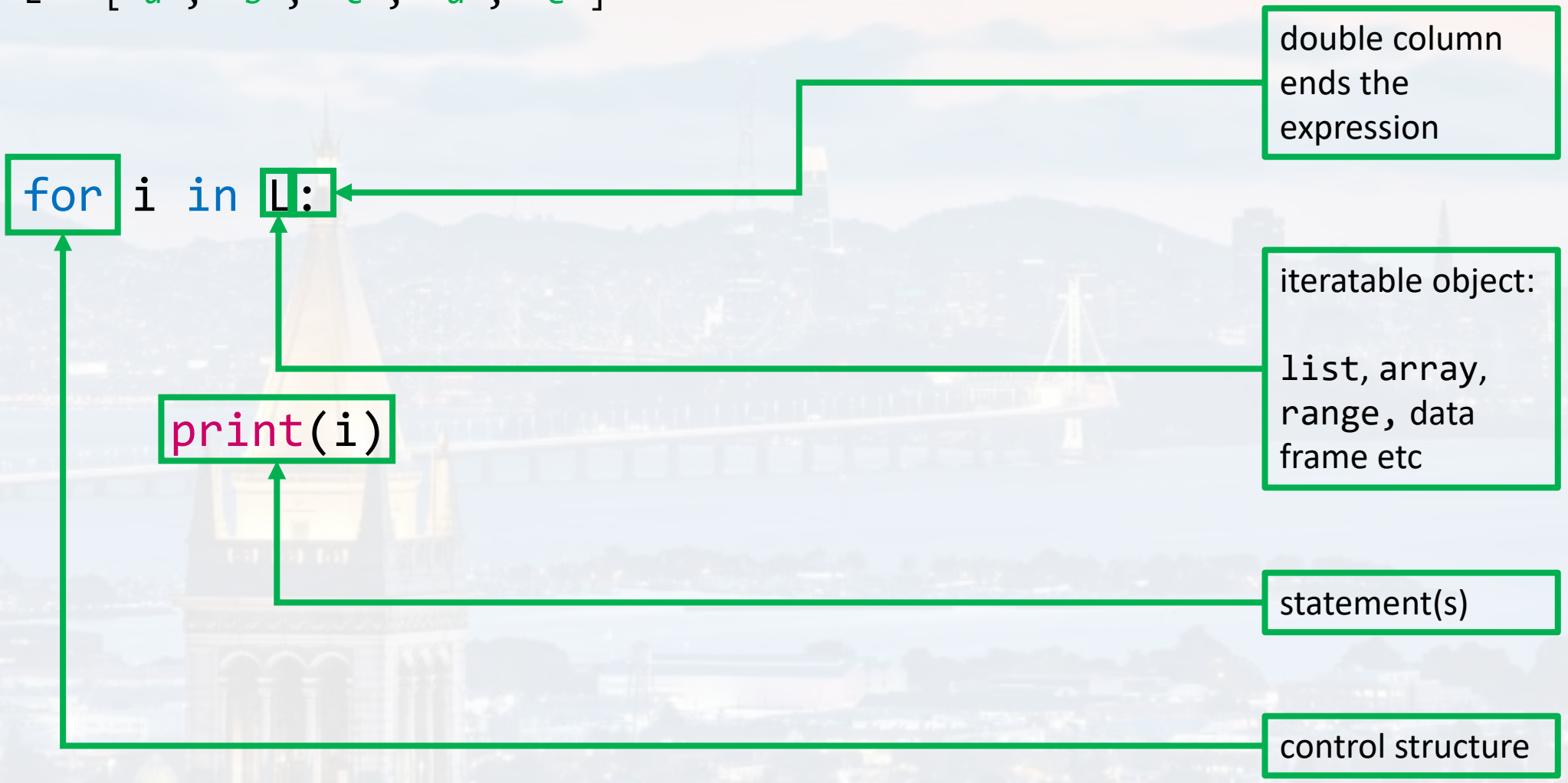
mapping: `dict`	`my_dict = {1: 'a', 2: 'b'}`
mapping: `set`	`my_set = {1, 2, 'a'}`

when to use:
- repetitive operations
- distinguish between different cases/ parts of the DA pipeline

but:
- avoid (especially `for` and `while`) loops as far as possible
- loops are slow → vectorization (see later)

often (not always): iterating over an object

```
L = ['a', 'b', 'c', 'd', 'e']
```



often (not always): iterating over an object

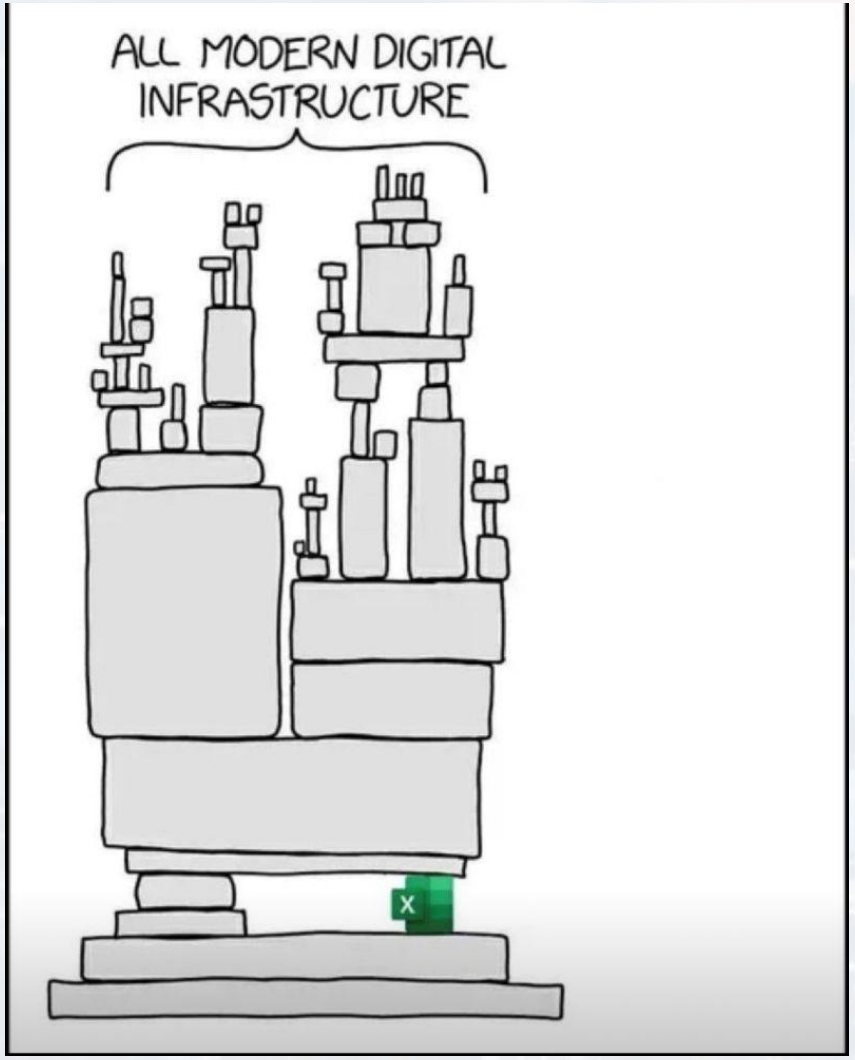
```
L = ['a', 'b', 'c', 'd', 'e']
```

```
for i in L:
    print(i)
```

free index variable

four blank spaces or tab

```
In [2]: L = ['a', 'b', 'c', 'd', 'e']

In [3]: for i in L:
   ...:
   ...: print(i)
   ...:
a
b
c
d
e
```



Outline

Control Structures

- General Idea and Structure
- `for` **Loops and Comprehension**
- `if`, `else` and `elif`
- `while`
- `break`, `continue` and `pass`

Functions

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

Introduction to Computational Techniques in Physics: for Loops and Comprehension

```
L = ['a', 'b', 'c', 'd', 'e']

for i in L:
    print(i)
```

iteratable object:

list, array, range, data frame etc

```
for i in 5:
    print(i)
```

vs

```
for i in range(5):
    print(i)
```

```
Traceback (most recent call last):

  Cell In[1], line 1
    for i in 5:

TypeError: 'int' object is not iterable
```

```
In [3]: for i in range(5):
   ...:
   ...:     print(i)
   ...:
0
1
2
3
4
```

Introduction to Computational Techniques in Physics: `for` Loops and Comprehension

```
L = ['a', 'b', 'c', 'd', 'e']
```

iterating over **content and index** of an object

```
for i, j in enumerate(L):
        print(str(i) + str(j))
```

```
0a
1b
2c
3d
4e
```

Introduction to Computational Techniques in Physics: `for` Loops and Comprehension

```
L = ['a', 'b', 'c', 'd', 'e']
```

iterating over **two** objects **simultaneously**

```
R = range(0,10,2)
```

```
for i, j in zip(R, L):
        print(str(i) + str(j))
```

```
0a
2b
4c
6d
8e
```

note: Even works, if objects have different lengths.
Just stops with the shortest object

Introduction to Computational Techniques in Physics: `for` Loops and Comprehension

```
L = ['a', 'b', 'c', 'd', 'e']
```

```
R = range(0,10,2)
```

iterating over **two** objects **simultaneously** and over **indices and content**

```
for i, (j, k) in enumerate(zip(R, L)):
        print(str(i) + str(j) + k)
```

```
00a
12b
24c
36d
48e
```

The more pythonic way is using ***comprehension***

```
from math import *

N = 10
Factorial = [None]*N

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

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

comprehension

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

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

note: very common in the Python community

shorter, often faster

only if loops are not too complex

The more pythonic way is using *comprehension*

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

```
Dict = {}

for code, nt in zip(Code, NT):
        Dict[nt] = code
```

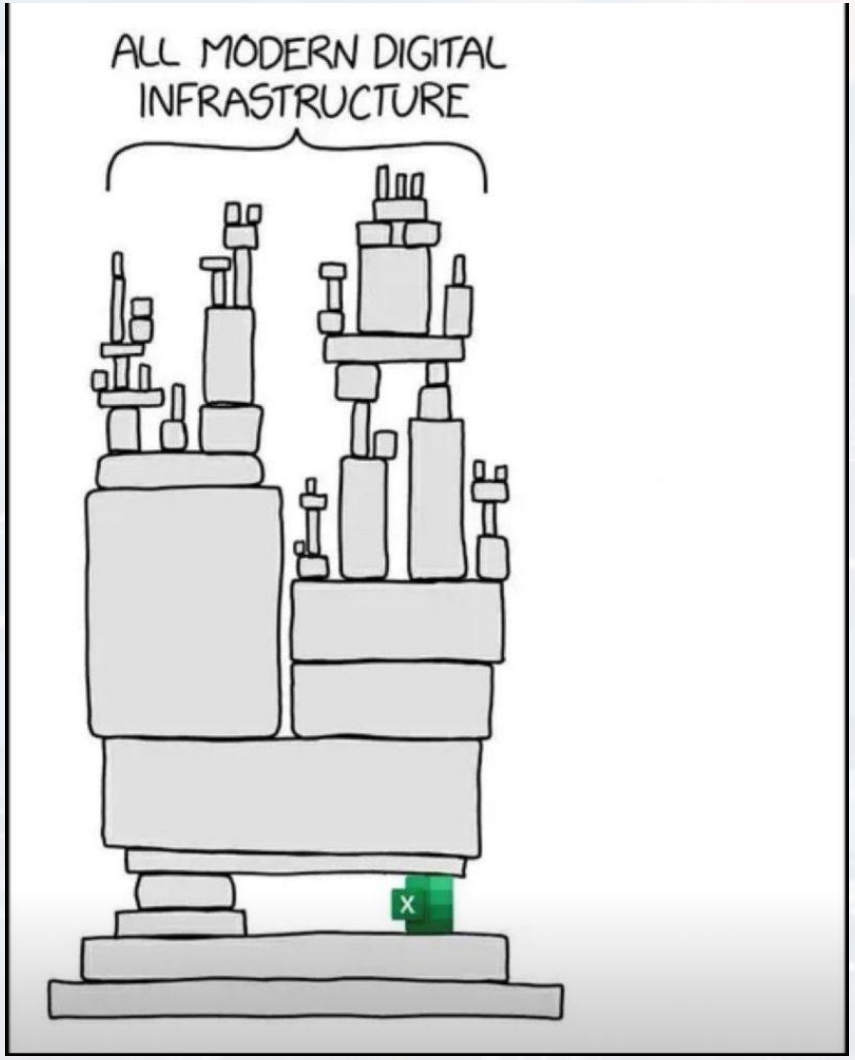
```
Dict['A']
```

```
Dict['A']
[1, 0, 0, 0]
```

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

```
Dict['A']
```

```
Dict['A']
[1, 0, 0, 0]
```



Outline

Control Structures

- General Idea and Structure
- `for` Loops and Comprehension
- **`if`, `else` and `elif`**
- `while`
- `break`, `continue` and `pass`

Functions

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

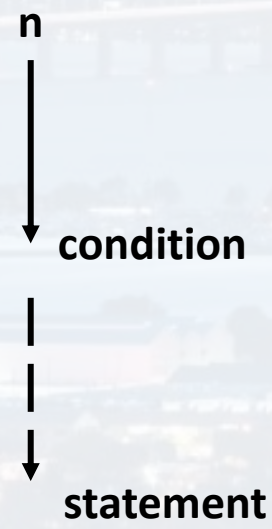
Introduction to Computational Techniques in Physics: `if`, `else` and `elif`

```
for n in range(10):
    if n%2 == 0:
        print('even number: ' + str(n))
```

applies if a condition is true

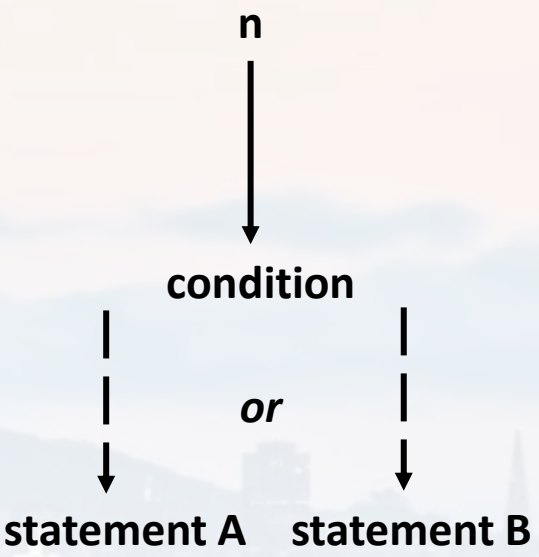
a logical condition (therefore ==)

```
even number: 0
even number: 2
even number: 4
even number: 6
even number: 8
```



Introduction to Computational Techniques in Physics: `if`, `else` and `elif`

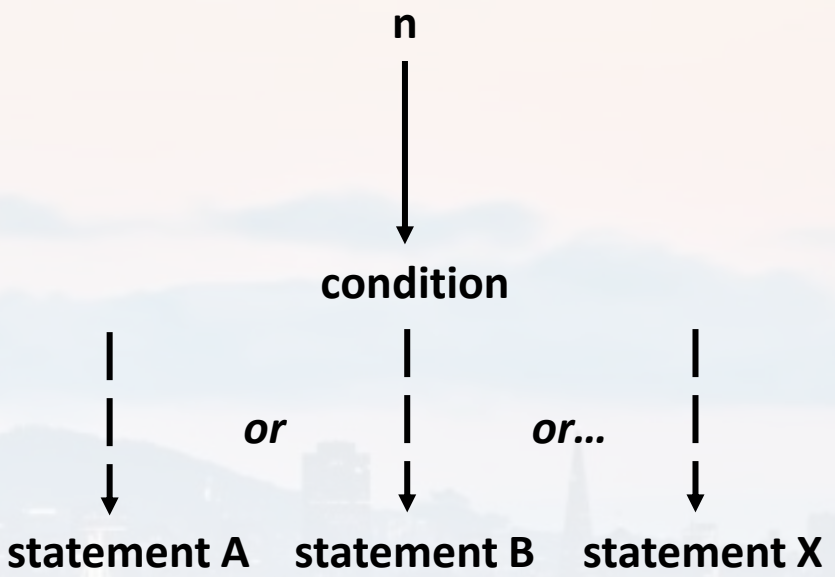
```
for n in range(10):

    if n%2 == 0:
        print('even number: ' + str(n))

    else:
        print('odd number: ' + str(n))
```




```
even number: 0
odd number: 1
even number: 2
odd number: 3
even number: 4
odd number: 5
even number: 6
odd number: 7
even number: 8
odd number: 9
```

Introduction to Computational Techniques in Physics: `if`, `else` and `elif`

```
for n in range(1,10):

    if n%2 == 0:
        print('even number: ' + str(n))

    if n%3 == 0:
        print('multiple of 3: ' + str(n))

    if n%5 == 0:
        print('multiple of 5: ' + str(n))
```

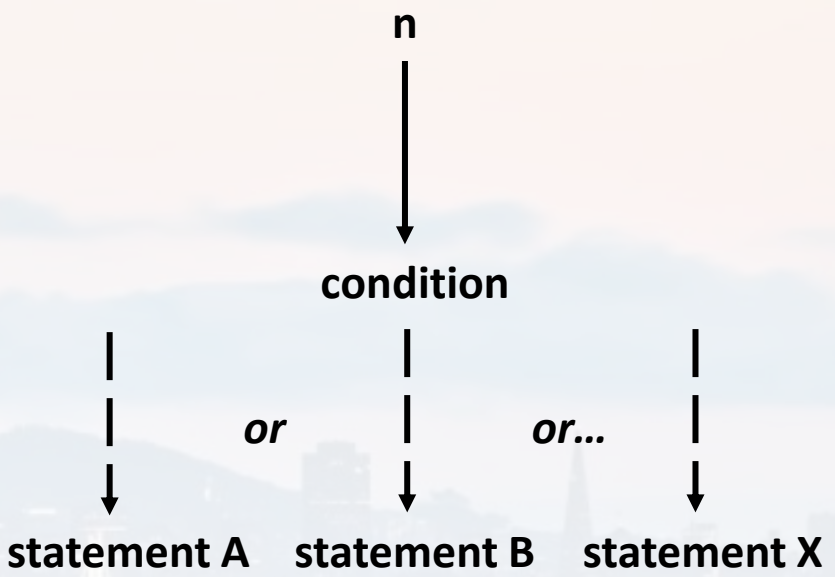



```
even number: 2
multiple of 3: 3
even number: 4
multiple of 5: 5
even number: 6
multiple of 3: 6
even number: 8
multiple of 3: 9
```

two conditions are true → **not** exclusive

Introduction to Computational Techniques in Physics: `if`, `else` and `elif`

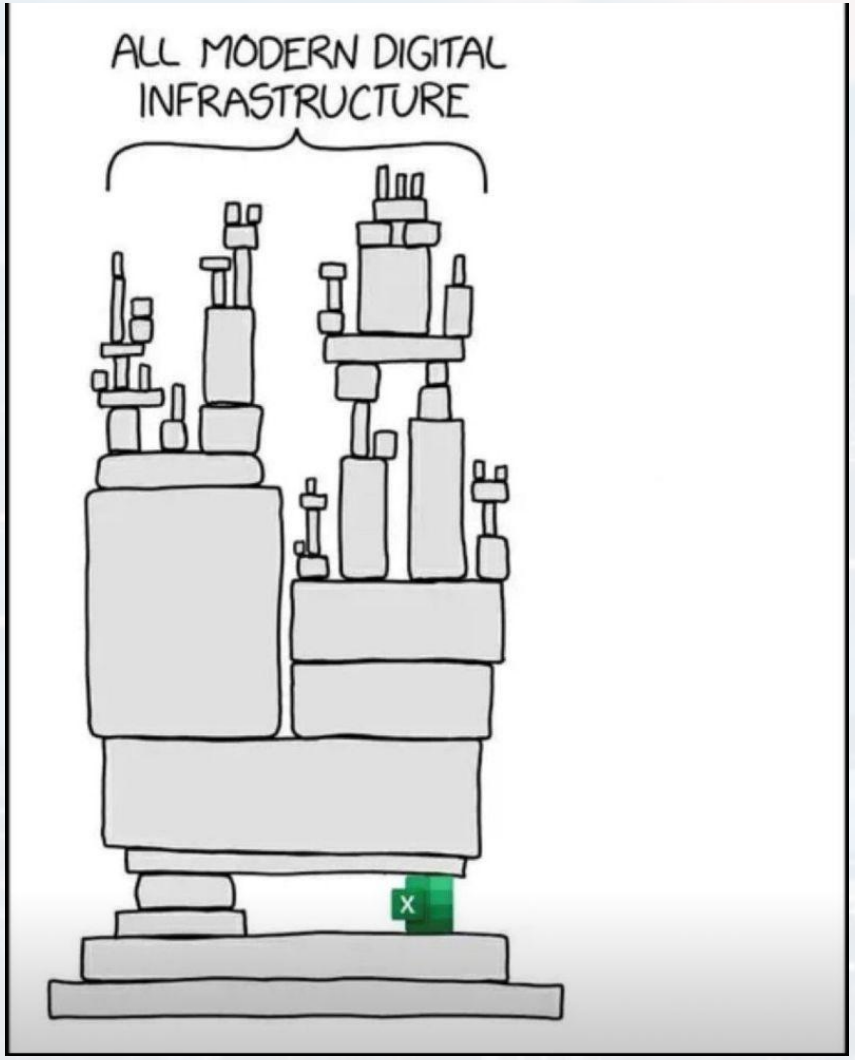
```
for n in range(1,10):

    if n%2 == 0:
        print('even number: ' + str(n))

    elif n%3 == 0:
        print('multiple of 3: ' + str(n))

    elif n%5 == 0:
        print('multiple of 5: ' + str(n))
```




```
even number: 2
multiple of 3: 3
even number: 4
multiple of 5: 5
even number: 6
even number: 8
multiple of 3: 9
```

once a condition applies → runs statement **exclusively**

Outline



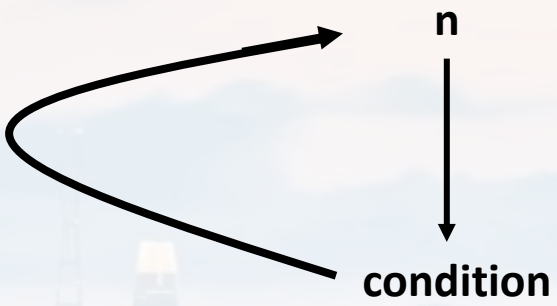
Control Structures

- General Idea and Structure
- `for` Loops and Comprehension
- `if`, `else` and `elif`
- **`while`**
- `break`, `continue` and `pass`

Functions

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

```
n = 0

while n < 10:

    n += 1
    print(n)
```

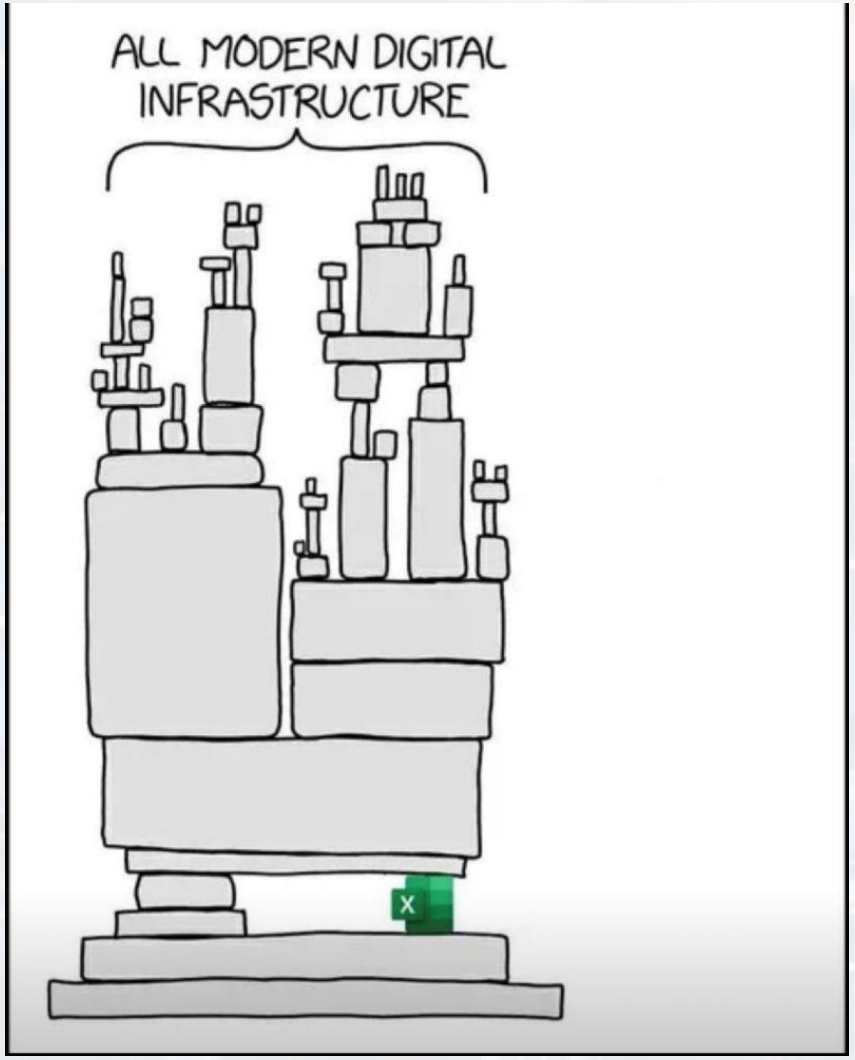


runs until condition is not true

```
1
2
3
4
5
6
7
8
9
10
```

note: make sure, that condition will be false at some point → may run infinitely → logical error

Outline



Control Structures

- General Idea and Structure
- `for` Loops and Comprehension
- `if`, `else` and `elif`
- `while`
- **`break`, `continue` and `pass`**

Functions

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

Introduction to Computational Techniques in Physics: `break`, `continue` and `pass`

checking, if integer N>3 is a prime number

```
N = 40

for n in range(3,N):

    result = N%n

    if result == 0:
        print('not prime')
```

```
not prime
not prime
not prime
not prime
not prime
not prime
```

it is sufficient to know for the first time if *N* is not prime

→ don't need to run the entire loop

Introduction to Computational Techniques in Physics: `break`, `continue` and `pass`

checking, if integer N>3 is a prime number

```
N = 40

for n in range(3,N):

    result = N%n

    if result == 0:
        print('not prime')

        break
```

interrupts loop immediately, if condition is true

```
not prime
```

checking, if integer N>3 is a prime number
Now we also want the code to print if **N is prime** too

```
N = 40

for n in range(3,N):

    result = N%n

    if result == 0:
        print('not prime')

        break

    else:
        print('is prime')
```

```
is prime
not prime
```

now says *'is prime'*, **each time** *N* is not dividable without remainder

Introduction to Computational Techniques in Physics: `break`, `continue` and `pass`

checking, if integer N>3 is a prime number
Now we also want the code to print if **N is prime** too

```
N = 40

for n in range(3,N):

    result = N%n

    if result == 0:
        print('not prime')

        break

    else:
        if n < N-1:
            continue

        print('is prime')
```

skips the current iteration

```
N = 39  not prime

N = 40  not prime

N = 41  is prime
```

Introduction to Computational Techniques in Physics: `break`, `continue` and `pass`

cleverer: **avoiding half of the iterations** in the first place:

```
if N > 1:

        for n in range(2, (N//2)+1):

                if (N % n) == 0:
                        print(N, 'not prime')
                        break
        else:
                print(N, 'is prime')

else:
        print(N, 'is not prime')
```

Introduction to Computational Techniques in Physics: `break`, `continue` and `pass`

```
7  N = 41
8
9  if N > 1:
10
11
12
13
```

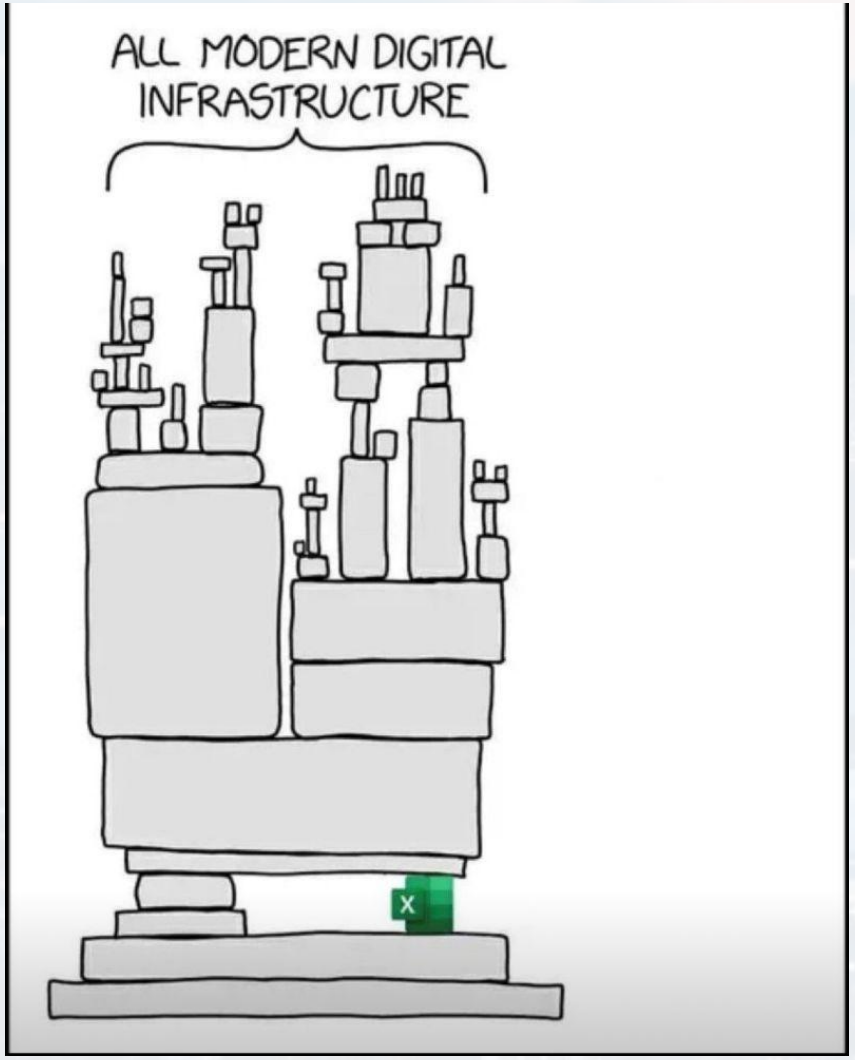
Syntax error before entering a statement.

Maybe you want to run the code without having written a statement!

```
7  N = 41
8
9  if N > 1:
10     pass
```

`pass` is a null statement: it matches the required syntax, but doesn't do anything, just a placeholder

Outline



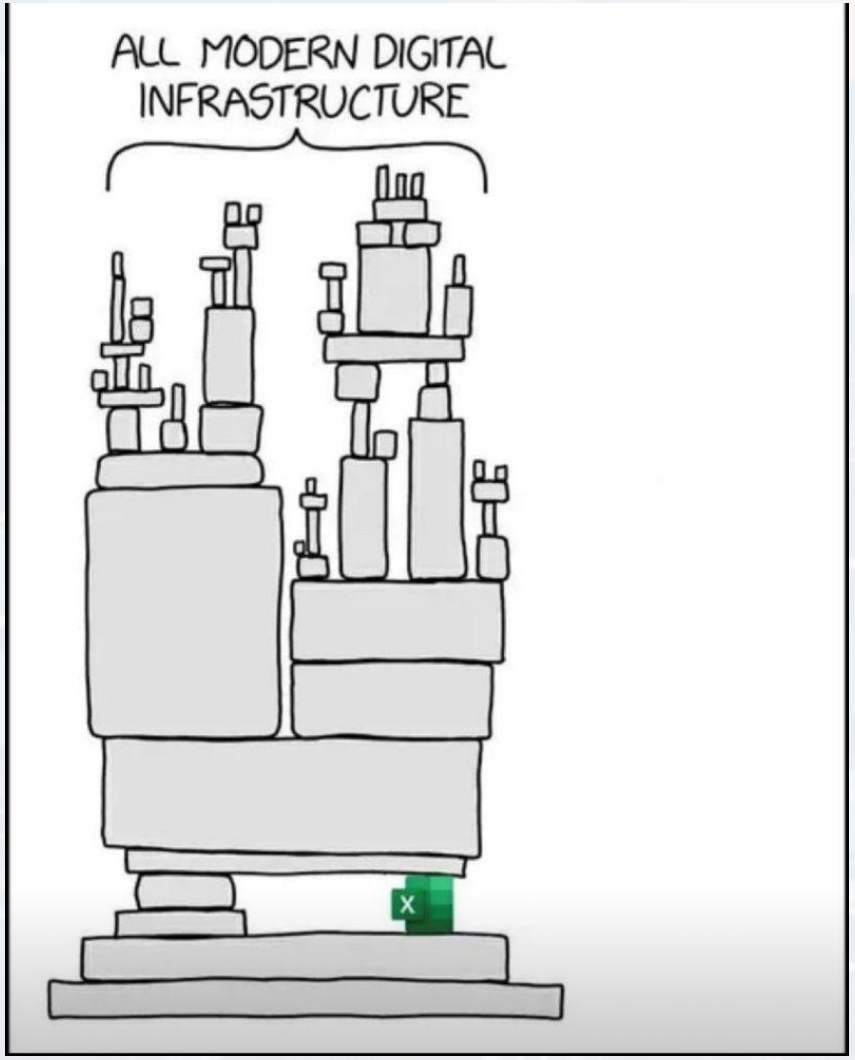
Control Structures

- General Idea and Structure
- `for` Loops and Comprehension
- `if`, `else` and `elif`
- `while`
- `break`, `continue` and `pass`

Functions

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

Outline



Control Structures

- General Idea and Structure
- `for` Loops and Comprehension
- `if`, `else` and `elif`
- `while`
- `break`, `continue` and `pass`

Functions

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

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)}
```

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

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]]
```

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

```
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

```
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**

```
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)}
```

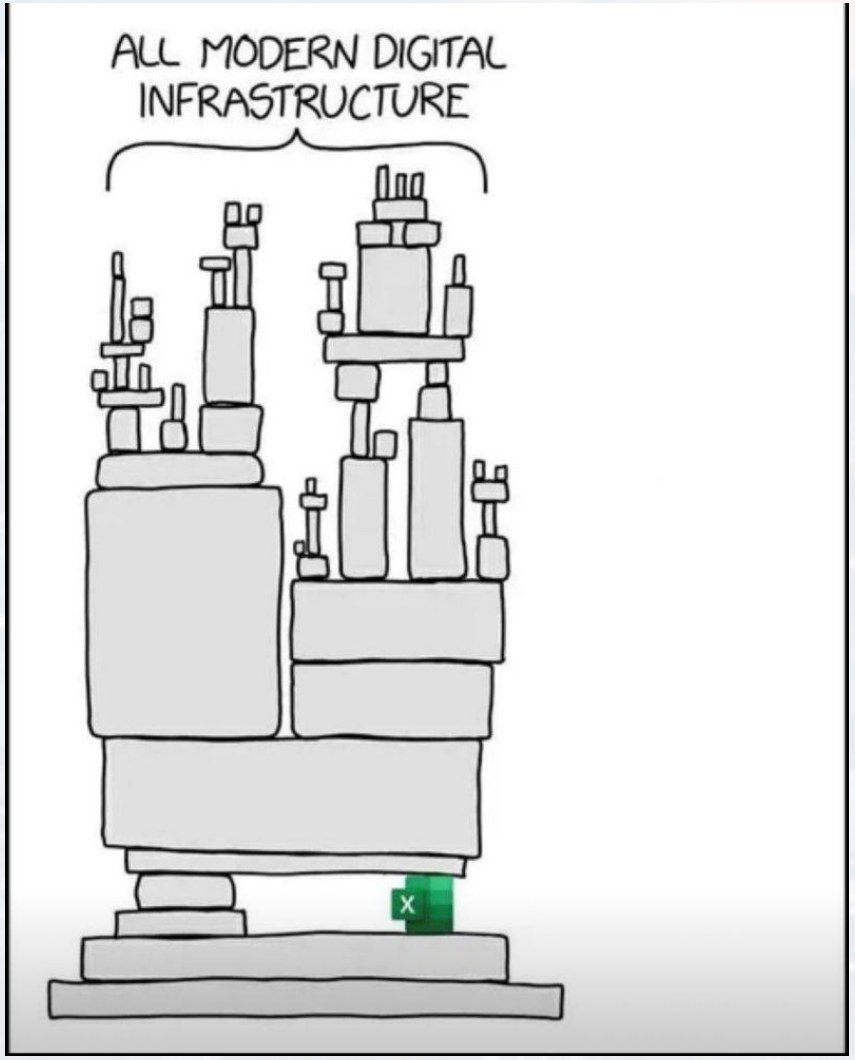
```
L = []

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



Control Structures

- General Idea and Structure
- `for` Loops and Comprehension
- `if`, `else` and `elif`
- `while`
- `break`, `continue` and `pass`

Functions

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

Introduction to Computational Techniques in Physics: map

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,…

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

Introduction to Computational Techniques in Physics: map

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`**:

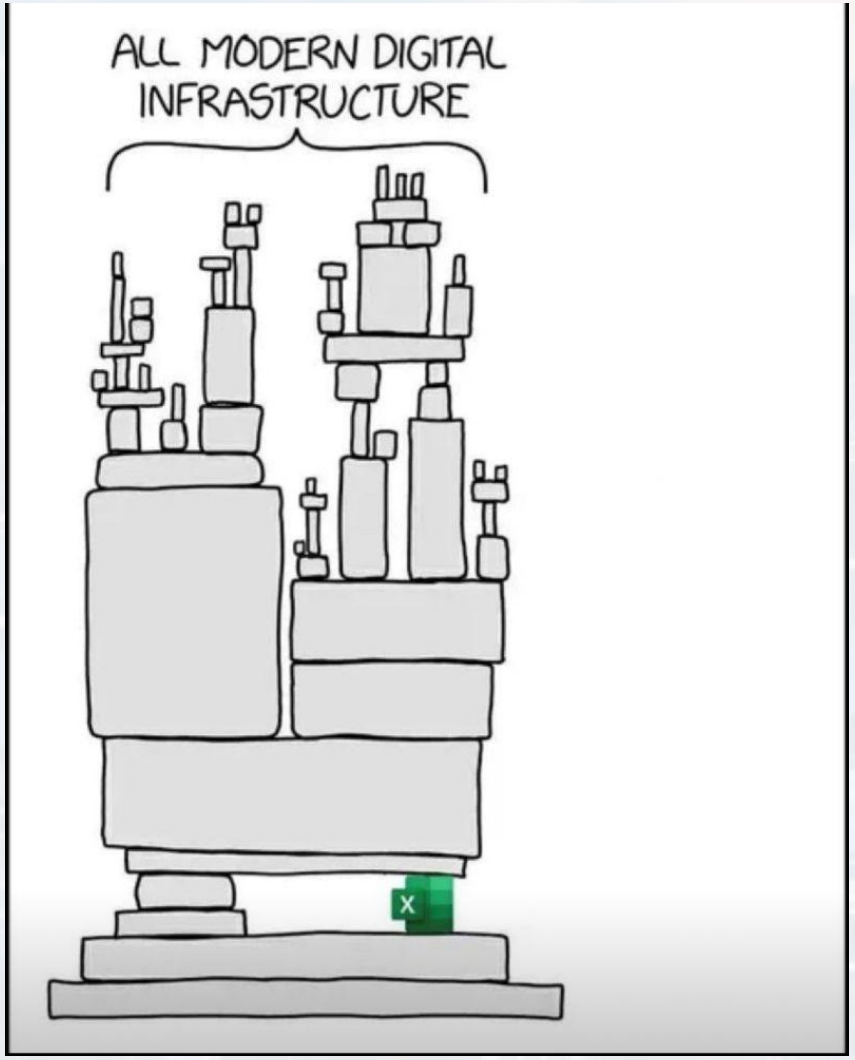
```
S1 = 'ACGGTCCGACCT'
S2 = 'TTCAGGT'
S3 = 'ATCGGCAATCTGCTTCA'
S4 = 'CGTCCGTA'

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



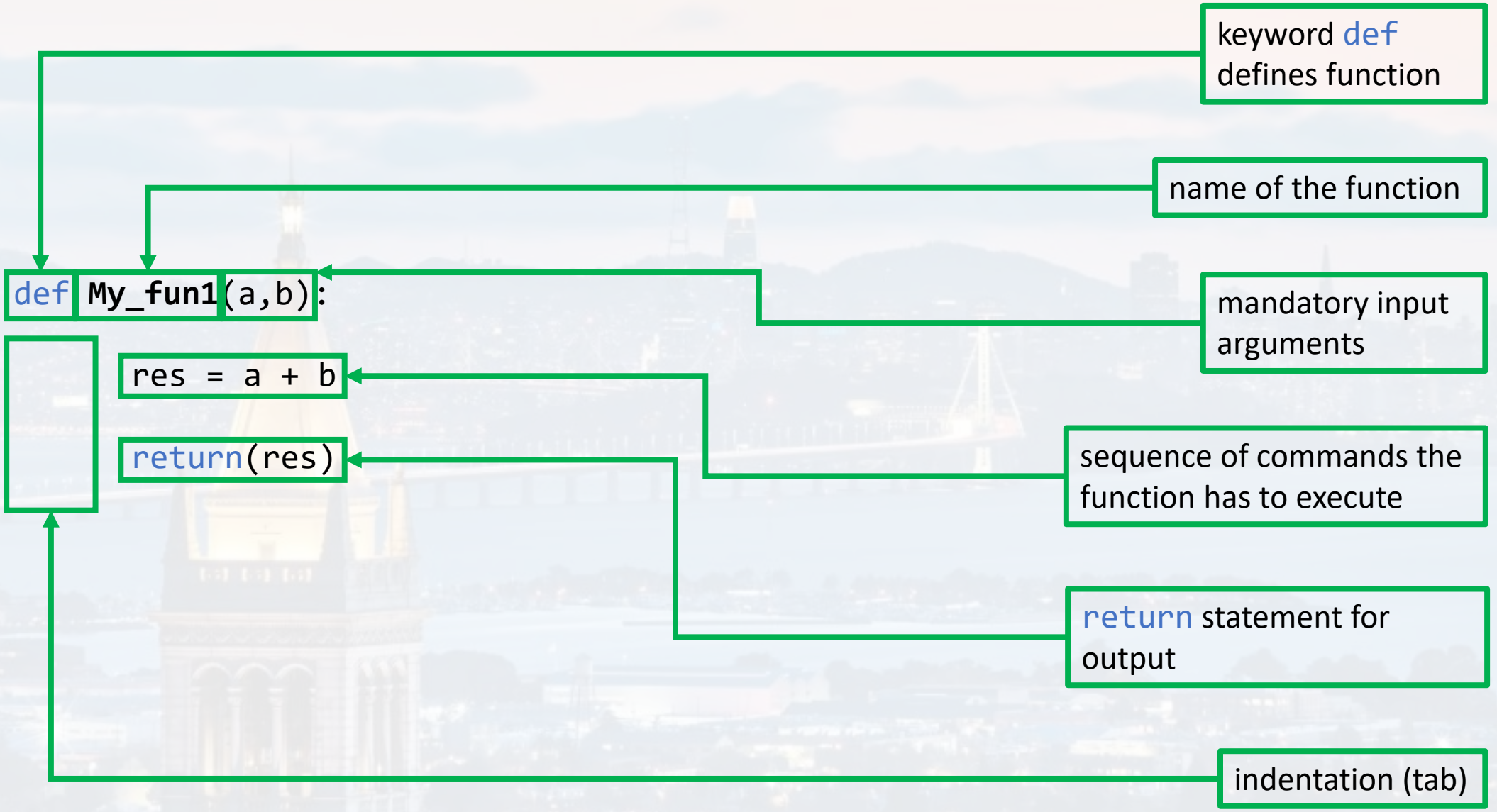
Control Structures

- General Idea and Structure
- `for` Loops and Comprehension
- `if`, `else` and `elif`
- `while`
- `break`, `continue` and `pass`

Functions

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

actual functions/methods in Python:



actual functions/methods in Python:

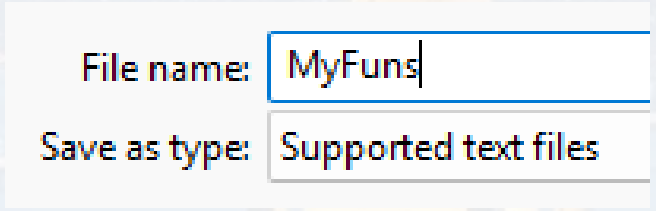
```
def My_fun1(a,b):
    res = a + b
    return(res)
```

also called **body** of the function

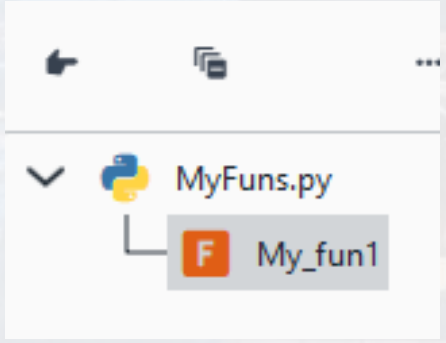
actual functions/methods in Python:

```
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)
```

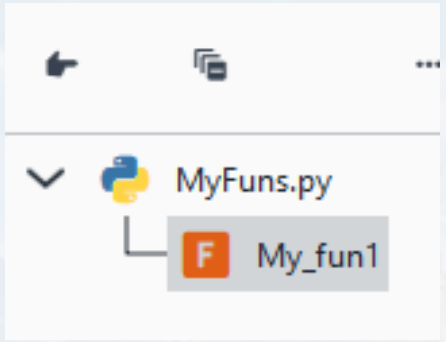


saving the `.py` script that contains `My_fun1` as `MyFuns.py`

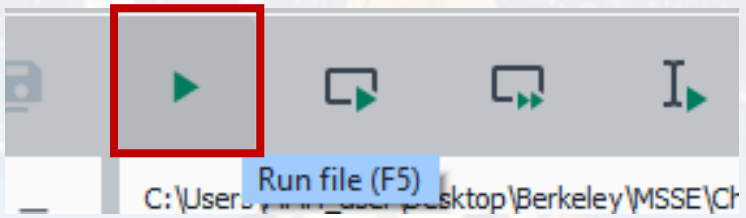


the `.py` script `MyFuns.py` contains `My_fun1`

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')
```

```
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
```

what can go wrong:

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'
```

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

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
```

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)
```

what can go wrong:

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
```

```
def My_fun1(a,b):

    res = a + b

    return(res)
```

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

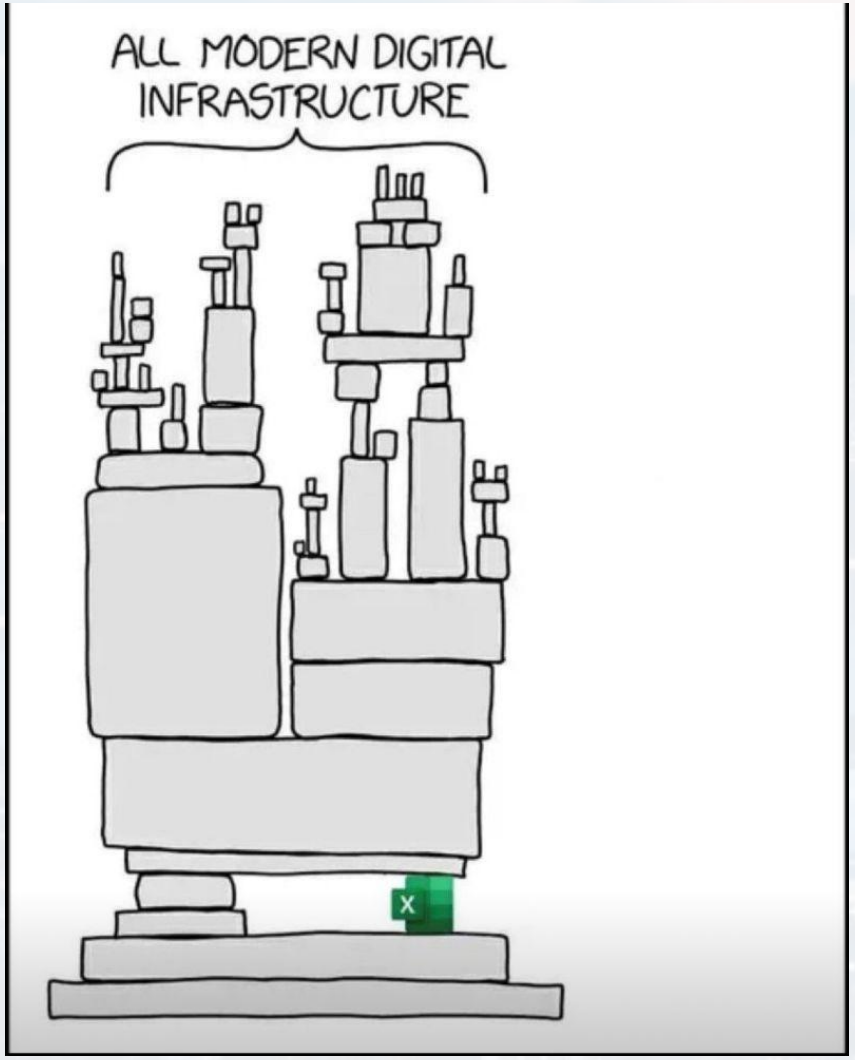
generating multiple outputs

```
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

Control Structures

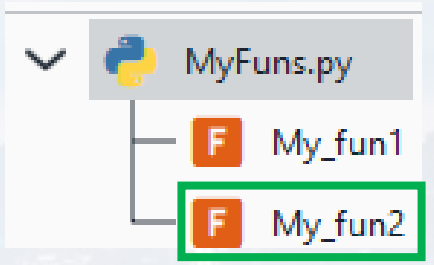
- General Idea and Structure
- `for` Loops and Comprehension
- `if`, `else` and `elif`
- `while`
- `break`, `continue` and `pass`

Functions

- `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)
```

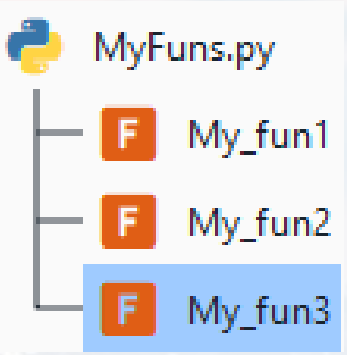
```
In [27]: My_fun2(5,3)
Out[27]: (8, 15, 125)
```

Introduction to Computational Techniques in Physics: *args and **kwargs

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'>
```

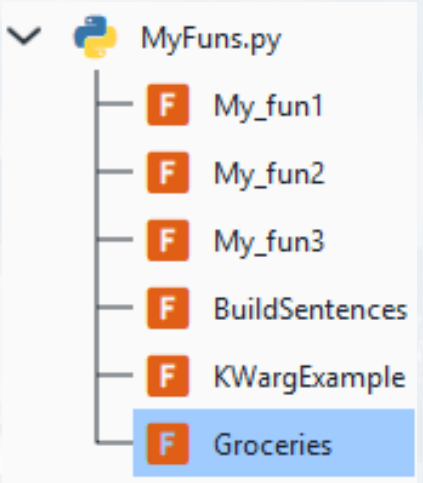
Why is it a dictionary?

Introduction to Computational Techniques in Physics: *args and **kwargs

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!