Berkeley
UNIVERSITY OF CALIFORNIA

M. Hohle:

Physics 77: Introduction to Computational Techniques in Physics

Introduction to Computational Techniques in Physics: syllabus

syllabus:

- Introduction to Unix & Python	(week 1 - 2)
- Functions, Loops, Lists and Arrays	**(week 3 - 4)**
- Visualization	(week 5)
- Parsing, Data Processing and File I/O	(week 6)
- Statistics and Probability, Interpreting Measurements	(week 7 - 8)
- Random Numbers, Simulation	(week 9)
- Numerical Integration and Differentiation	(week 10)
- Root Finding, Interpolation	(week 11)
- Systems of Linear Equations	(week 12)
- Ordinary Differential Equations	(week 13)
- Fourier Transformation and Signal Processing	(week 14)
- Capstone Project Presentations	(week 15)

```
L = [1, 2, 3, -2]
```

list: default format in Python

```
String    = 'Hello ' + ''World''
List      = [1, 2, 3, 5, ''World'' ]
Tuble     = (1, 2)
Dict      = {'A': 1, 'B': 2}
Array     = np.array([1, 2, 3, 5])
pd.DataFrame
```

type: `2*L`

```
Out[1]: [1, 2, 3, -2, 1, 2, 3, -2]
```

This is the first thing we need to do: loading packages/libraries/modules

```
import numpy as np
```

alias **you** set

name of library/module

We will use some **standard libraries** such as

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

etc…

type:

np. and use the auto complete (tab)

```
In [32]: np.
```

```
ALLOW_THREADS
AxisError
BUFSIZE
CLIP
ComplexWarning
DataSource
ERR_CALL
ERR_DEFAULT
```

```
L_arr = np.array(L)
```

Name	Type	Size	Value
L	list	4	[1, 2, 3, -2]
L_arr	Array of int32	(4,)	Min: -2 Max: 3

```
String    = 'Hello ' + ''World''
List      = [1, 2, 3, 5, ''World'' ]
Tuble     = (1, 2)
Dict      = {'A': 1, 'B': 2}
Array     = np.array([1, 2, 3, 5])
pd.DataFrame
```

try the following commands:

```
dir(L)
dir(L_arr)

L.shape()
L_arr.shape()
```

try now something like:

```
a = 2
b = 3
```

```
String    = 'Hello ' + ''World''
List      = [1, 2, 3, 5, ''World'' ]
Tuble     = (1, 2)
Dict      = {'A': 1, 'B': 2}
Array     = np.array([1, 2, 3, 5])
pd.DataFrame
```

Try to type and run the following numerical operations:

```
1) a*b
2) a = b
3) c = a - b
4) a**b
5) np.exp(a)
6) np.power(c,3)
7) np.log(a)
```

```
np.log10(10)
np.log2(2)
```

recall: `logB(x) = logA(x)/logA(B)`

Now with vectors. Type:

```
v1 = np.array([1,5,0,-3])
v2 = np.array([3,-1,2,2])
```

```
String    = 'Hello ' + ''World''
List      = [1, 2, 3, 5, ''World'' ]
Tuble     = (1, 2)
Dict      = {'A': 1, 'B': 2}
Array     = np.array([1, 2, 3, 5])
pd.DataFrame
```

Try to type and run the following numerical operations:

```
1) v1 + v2
2) v1 + 2
3) v1*2          compared to L*2 !
4) v1*v2
5) np.dot(v1,v2)
6) np.outer(v1,v2)
```

```
String    = 'Hello ' + ''World''
List      = [1, 2, 3, 5, ''World'' ]
Tuble     = (1, 2)
Dict      = {'A': 1, 'B': 2}
Array     = np.array([1, 2, 3, 5])
pd.DataFrame
```

Now with vectors. Type:

```
v1 = np.array([1,5,0,-3])
v2 = np.array([3,-1,2,2])

1) np.dot(v1,v2)
2) np.outer(v1,v2)

3) np.multiply()
```

$$\begin{pmatrix} a & b & c \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = a\alpha + b\beta + c\gamma$$

$$\begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} \begin{pmatrix} a & b & c \end{pmatrix} = \begin{pmatrix} a\alpha & \alpha b & \alpha c \\ a\beta & \beta b & \beta c \\ a\gamma & \gamma b & \gamma c \end{pmatrix}$$

```
np.outer(v1,v2)

Out[11]:

array([[ 3, -1,  2,  2],
       [15, -5, 10, 10],
       [ 0,  0,  0,  0],
       [-9,  3, -6, -6]])
```

```
np.dot(v1,v2)

Out[10]:     -8
```

```
String    = 'Hello ' + ''World''
List      = [1, 2, 3, 5, ''World'' ]
Tuble     = (1, 2)
Dict      = {'A': 1, 'B': 2}
Array     = np.array([1, 2, 3, 5])
pd.DataFrame
```

creating matrix from vectors:

```
v_tot_row      = np.array([v1,v2])
v_tot_col      = np.column_stack([v1,v2])
```

creating matrix from random numbers and zeros:

```
M              = np.random.uniform(0,1,(5,6,7))
Z              = np.zeros((5,6,7))
```

common matrix operations:

```
v_tot_col_trans = v_tot_col.transpose()
```

```
np.max(v_tot_row)
```

check also: `np.argmax(v_tot_row)`

```
np.min(v_tot_row)
np.mean(v_tot_row)
np.var(v_tot_row)
np.std(v_tot_row)
```

loops: repetitive tasks

```
for
if
else
while
...
```

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

loops: repetitive tasks

```
for
if
else
while
...
```

```
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
```

loops: repetitive tasks

```
for
if
else
while
...
```

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

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

vs

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

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

vs

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

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

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

loops: repetitive tasks

```
for
if
else
while
...
```

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

v = np.arange(0,4,2)


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

```
In [13]: for i, (j, k) in enumerate(zip(v, L)):
    ...:     print(str(i) + str(j) + k)
    ...:
00a
12b
24c
36d
48e
```

loops: Python syntax

```
for
if
else
while
...
```

```
L  = list(np.arange(0,100,1))
L2 = L


for i, l in enumerate(L):
    L2[i] = l**2
```

actual **pythonian** way:

```
L3 = [l**2 for l in L]
```

called **comprehension**

→ more compact

→ faster!

loops: Python syntax

```
for
if
else
while
...
```

```
L  = list(np.arange(0,10000,1))

t1 = datetime.now()

for i in range(10000):
    L2 = L

    for i, l in enumerate(L):
        L2[i] = l**2


t2 = datetime.now()
dt = (t2 - t1)/10000
print("Average Runtime: " + str(dt))
```

1.339 ms

```
from datetime import datetime
```

loops: Python syntax

```
for
if
else
while
...
```

```
L   = list(np.arange(0,10000,1))
```

1.339 ms

```
t1 = datetime.now()

for i in range(10000):




t2 = datetime.now()
dt = (t2 - t1)/10000
print("Average Runtime: " + str(dt))
```

loops: Python syntax

```
for
if
else
while
...
```

```
L  = list(np.arange(0,10000,1))
```

1.339 ms

```
t1 = datetime.now()

for i in range(10000):
    L2 = [l**2 for l in L]
```

0.486 ms

```
t2 = datetime.now()
dt = (t2 - t1)/10000
print("Average Runtime: " + str(dt))
```

loops: Python syntax

```
for
if
else
while
...
```

```
L   = list(np.arange(0,10000,1))

t1 = datetime.now()

for i in range(10000):




t2 = datetime.now()
dt = (t2 - t1)/10000
print("Average Runtime: " + str(dt))
```

1.339 ms

0.486 ms

loops: Python syntax

```
for
if
else
while
...
```

```
L  = list(np.arange(0,10000,1))

t1 = datetime.now()

for i in range(10000):
    L = []
    N = 10000

        for n in range(N):
            L += [n**2]

t2 = datetime.now()
dt = (t2 - t1)/10000
print("Average Runtime: " + str(dt))
```

1.339 ms

0.486 ms

4.1 ms

Always use comprehension!

functions

`lambda` → anonymous function
`map` → function/method
`def` → function/method

functions

`lambda` → **anonymous function**
`map` → function/method
`def` → function/method

```
lambda arguments: expression

Square = lambda L: [l**2 for l in L]

type(Square)

In [20]: Square([1,2,3,4])
Out[20]: [1, 4, 9, 16]

L  = np.arange(0,10000,1)

t1 = datetime.now()

for i in range(10000):
        L2 = Square(L)

t2 = datetime.now()
dt = (t2 - t1)/10000
print("Average Runtime: " + str(dt))
```

0.582 ms

functions

`lambda` → **anonymous function**
`map` → function/method
`def` → function/method

```
lambda arguments : expression
```

```
Alphabet = ['A', 'C', 'G', 'T']

Encoding = np.eye(4)

Dict     = {char: i for char, i in zip(Alphabet, Encoding)}
```

check:

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

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

check:

```
EncodeMySeq('ACTTGA')
```

functions

`lambda` → anonymous function
`map` → **function/method**
`def` → function/method

```
map(function, iterable)

Square = lambda L: [l**2 for l in L]


L   = np.arange(0,10000,1)

t1 = datetime.now()

for i in range(10000):
        L2 = list(map(Square, [L]))


t2 = datetime.now()
dt = (t2 - t1)/10000
print("Average Runtime: " + str(dt))
```

0.457 ms

Introduction to Computational Techniques in Physics: **Functions, Loops, Lists and Arrays**

functions loops & functions

`lambda` → anonymous function
`map` → **function/method**
`def` → function/method

if you **have to** use a loop: → comprehension
try to avoid loops → `map, lambda`

functions

`lambda` → anonymous function
`map` → function/method
`def` → **function/method**

*args und **kwargs

```
In [6]: plt.plot(
```

```
plot(*args, scalex=True, scaley=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)
...
```

functions

1) mandatory input arguments

```
lambda  → anonymous function
map     → function/method
def     → function/method
```

```
def fun1(a,b):

        res = a + b
        return(res)



fun1(2, 3)
```

```
In [6]: def fun1(a,b):
   ...:
   ...:         res = a + b
   ...:         return(res)
   ...:

In [7]: fun1(
        fun1(a, b)

        No documentation available
```

functions

2) default arguments

```
lambda  → anonymous function
map     → function/method
def     → function/method
```

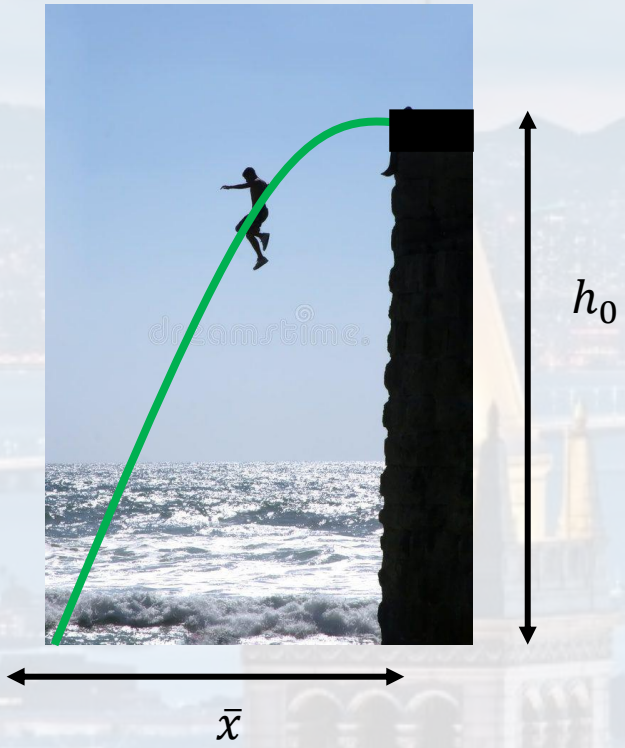
```
def fun2(a, b = 1):

        res = a + b
        return(res)



fun2(2, 3)
fun2(2)
```

```
In [8]: def fun2(a,b=1):
   ...:
   ...:     res = a + b
   ...:     return(res)
   ...:

In [9]: fun2(
```

```
fun2(a, b=1)

No documentation available
```

lets write a function that tells us how far we can jump from a height h_0

`lambda` → anonymous function
`map` → function/method
`def` → **function/method**



input: $v_0\ [m/s]$
$h_0\ [m]$

g = 9.81 m/s as default

$$y(x) = -\frac{g}{2}\frac{x^2}{(v_0 \cos\alpha)^2} + x\ tan\alpha + h_0$$

$$\bar{x} = \frac{v_0 \cos\alpha}{g}\left(v_0 \sin\alpha + \sqrt{(v_0 \sin\alpha)^2 + 2gh_0}\right)$$

lets write a function that tells us how far we can jump from a height h_0

```
lambda  → anonymous function
map     → function/method
def     → function/method
```

```
import numpy as np
import matplotlib.pyplot as plt


def Jump(v_0, h_0, g = 9.81, alpha = 0):

    alpha  = alpha * np.pi/ 180 #degrees to rad

    vsin   = v_0 * np.sin(alpha)
    vcos   = v_0 * np.cos(alpha)

    x_bar  = (vcos/g) * (vsin + np.sqrt(vsin**2 + 2*g*h_0))


    print(f'Distance is: {x_bar:.3f}m')
```

```
In [16]: Jump(10, 20)
Distance is: 20.193m

In [17]: Jump(10, 20, alpha=45)
Distance is: 20.258m

In [18]: Jump(10, 0, alpha=45)
Distance is: 10.194m
```

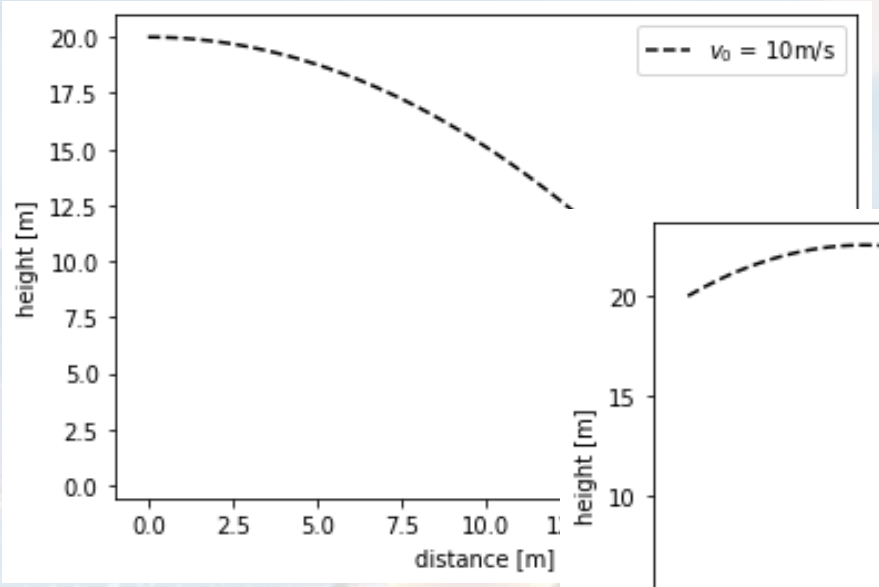
lets write a function that tells us how far we can jump from a height h_0

lambda → anonymous function
map → function/method
def → **function/method**

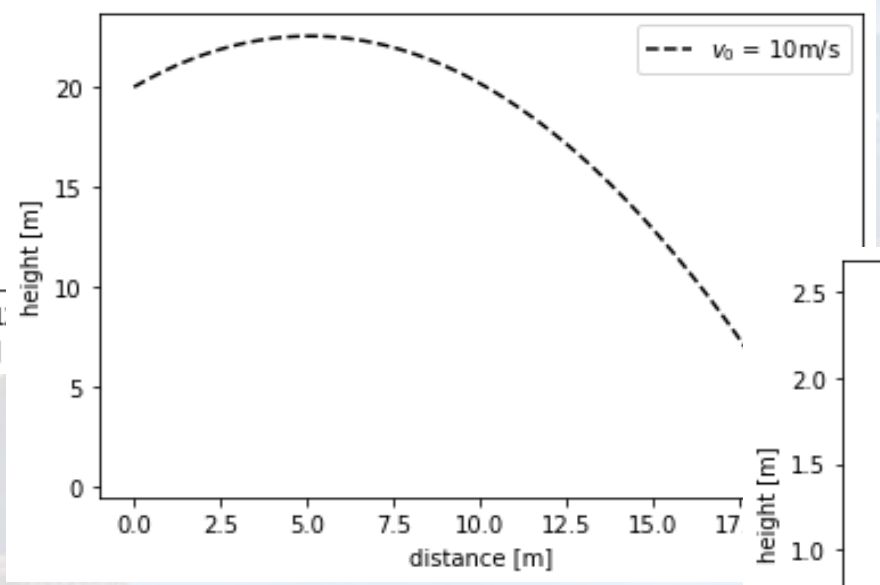
```
def Jump(v_0, h_0, g = 9.81, alpha = 0):
...

    x_bar  = (vcos/g) * (vsin + np.sqrt(vsin**2 + 2*g*h_0))
    x_plot = np.arange(0, x_bar, x_bar/100)

    y      = (-0.5 * g *x_plot**2)/ ((v_0*np.cos(alpha))**2)\
             + np.tan(alpha) * x_plot + h_0


    plt.plot(x_plot, y, '--', color = 'k',\
                        label = r'$v_{0}$ = ' + str(v_0) + 'm/s')
    plt.xlabel('distance [m]')
    plt.ylabel('height [m]')
    plt.legend()
    plt.show()

    print(f'Distance is: {x_bar:.3f}m')
```

lets write a function that tells us how far we can jump from a height h_0

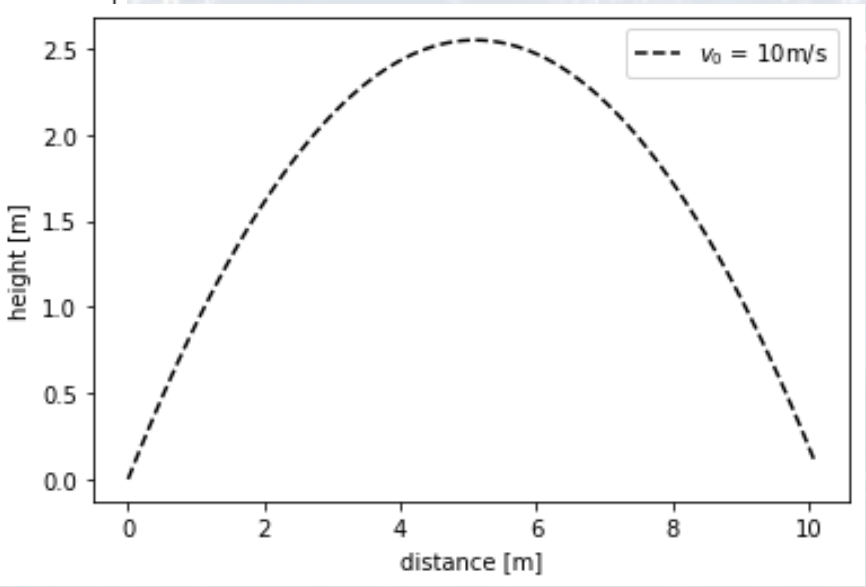
`lambda` → anonymous function
`map` → function/method
`def` → **function/method**




```
Jump(10, 20)
```




```
Jump(10, 20, alpha = 45)
```




```
Jump(10, 0, alpha = 45)
```

lets write a function that tells us how far we can jump from a height h_0

```
lambda  → anonymous function
map     → function/method
def     → function/method
```

```
def Jump(v_0, h_0, g = 9.81, alpha = 0):
...

    plt.plot(x_plot, y, '--', color = 'k',\
                        label = r'$v_{0}$ = ' + str(v_0) + 'm/s')
    plt.xlabel('distance [m]')
    plt.ylabel('height [m]')
    plt.legend()
    plt.show()

    print(f'Distance is: {x_bar:.3f}m')

    return x_bar, x_plot
```

```
In [20]:  [Xb, Xp] = Jump(10, 20)
Distance is: 20.193m

In [21]:  [Xb, _] = Jump(10, 25)
Distance is: 22.576m
```

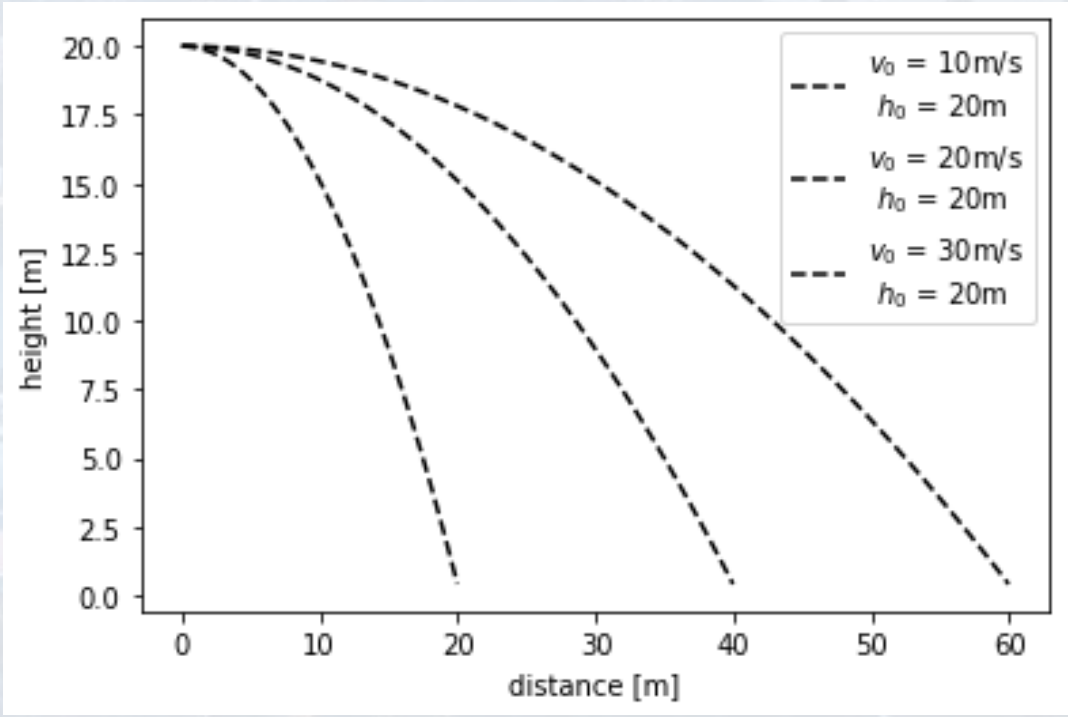
Write a function Jump2, that takes a vector of `v_0` and `h_0` and creates one plot for all trajectories

`lambda` → anonymous function
`map` → function/method
`def` → **function/method**

use a loop within the function, i.e.

```
for i, (v, h) in enumerate(zip(v_0, h_0)):
```

```
X = Jump2([10, 20, 30], [20, 20, 20])
```



functions

3) optional input arguments

```
def fun3(a, b = 1, *c):

        res = a + b

        if c:#tuple
                print(type(c))
                for c in c:
                        print(a**c)

         else:
             print('no c available')

        return(res)


fun3(2)
fun3(2, 3)
fun3(2, 3, 4)
```

lambda → anonymous function
map → function/method
def → **function/method**

```
In [11]: fun3(
         fun3(a, b=1, *c)

         No documentation available
```

c has been converted into a tuple!

```
In [29]: fun3(1,2,3)
<class 'tuple'>
1
Out[29]: 3
```

```
In [33]: fun3(2,1,1,2,3,4)
<class 'tuple'>
2
4
8
16
Out[33]: 3
```

functions

4) optional keyword arguments of any number **kwargs

lambda → anonymous function
map → function/method
def → **function/method**

```
def fun4(a, b = 1, *c, **d):

    res = a + b

    if c:
        res = res * c

    if 'hi' in d:#dict
        print(d['hi'])


    return(res)
```

```
In [13]: fun4(
         fun4(a, b=1, *c, **d)

         No documentation available
```

```
fun4(2)
fun4(2, 3)
fun4(2, 3, d = 'hi')
fun4(1, hi = 'hi')
fun4(1, 3, hi = 'abc')
fun4(1, 3, 3, hi = 'abc')
```

Interpret the results!

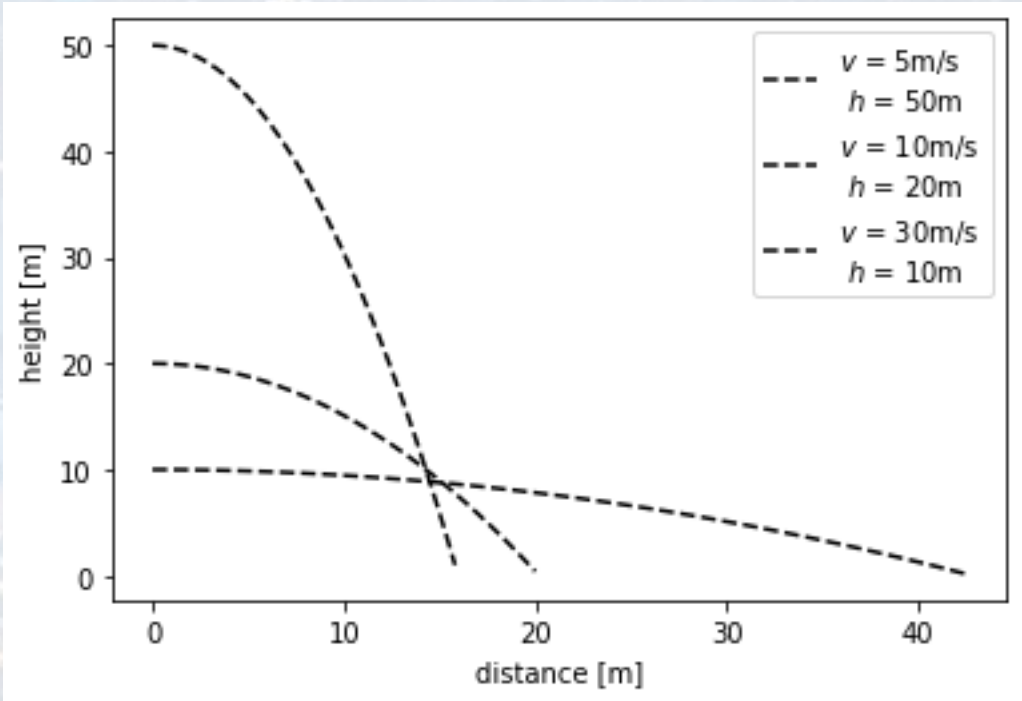
Introduction to Computational Techniques in Physics: **Functions, Loops, Lists and Arrays**

Write a function `Jump3`, that creates a plot, only if an optional keyword argument `plot = 'yes'` is given.

`lambda` → anonymous function
`map` → function/method
`def` → **function/method**

Put all three functions into a module, called `Jump`

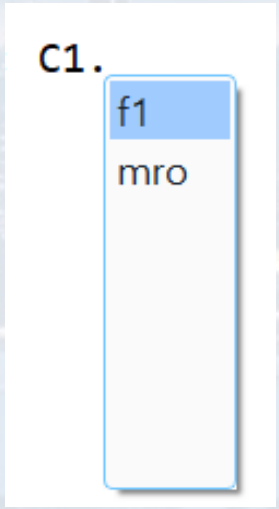
```
from Jump import *

X = Jump3([10, 20, 30], [20, 20, 40], plot = 'yes')
```

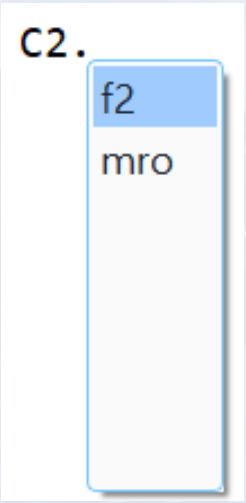


functions and classes

```
lambda  → anonymous function
map     → function/method
def     → function/method
```

```
class C1():

    def f1(a,b):
        print(a+b)


class C2():

    def f2(a,b):
        print(a*b)
```



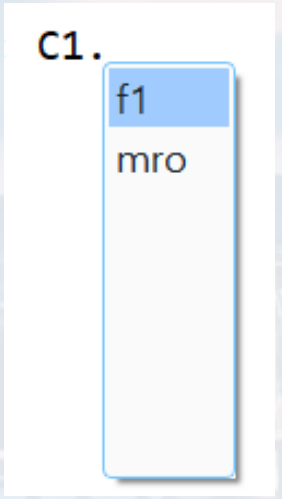


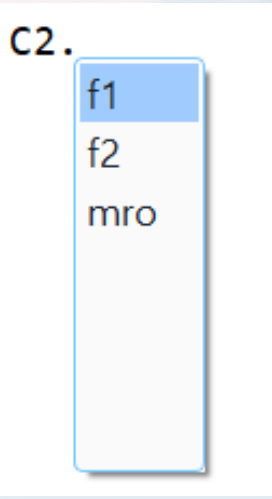

```
In [198]: C1.f1(1,2)
3

In [199]: C2.f2(1,2)
2
```

functions and classes

```
lambda  → anonymous function
map     → function/method
def     → function/method
```

```
class C1():

    def f1(a,b):
        print(a+b)


class C2(C1):

    def f2(a,b):
        print(a*b)
```






```
In [205]: C2.f1(1,2)
3
```

f1 got ***inherited*** from C1

C1 is the super class of C2
C2 is the sub class of C1

functions and `classes`

`lambda` → anonymous function
`map` → function/method
`def` → **function/method**

this…

```
class C1():

    def f1(a,b):
        print(a+b)


class C2(C1):

    def f2(a,b):
        print(a*b)
```

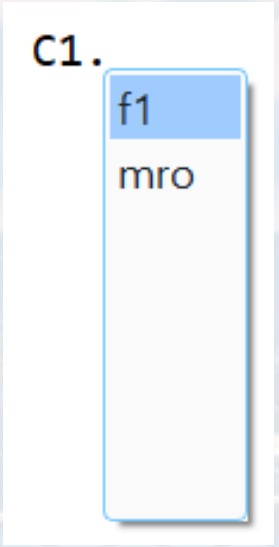
…is equivalent to

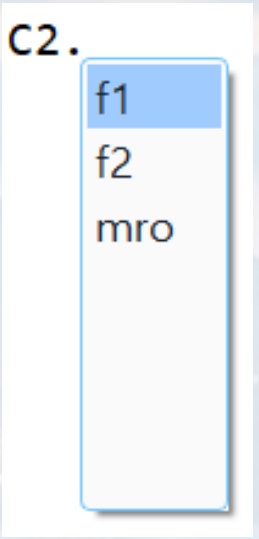
```
class C1():

    def f1(a,b):
        print(a+b)


class C2():

    def f2(a,b):
        print(a*b)

    def f1(a,b):
        C1.f1(a,b)
```

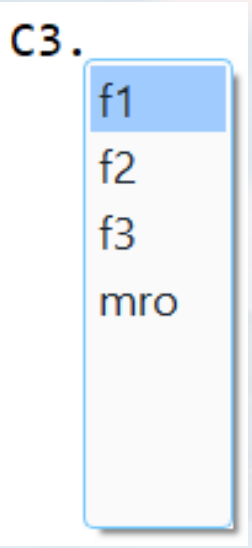
functions and **classes**

`lambda` → anonymous function
`map` → function/method
`def` → **function/method**

```
class C1():

    def f1(a,b):
        print(a+b)


class C2(C1):

    def f2(a,b):
        print(a*b)


class C3(C2):

    def f3(a,b):
        print(a**b)
```





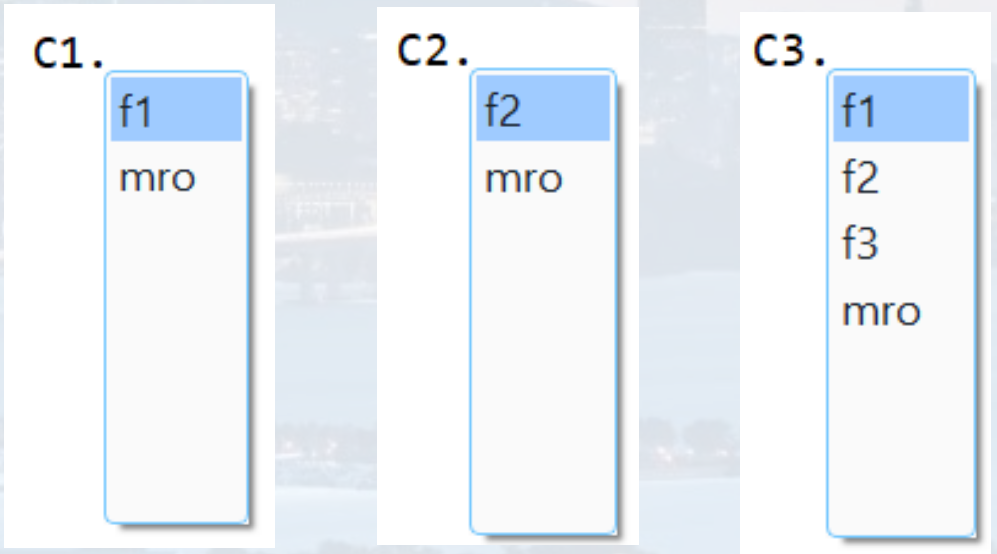



```
In [12]: C3.f1(1,2)
3

In [13]: C3.f3(1,2)
1
```

functions and **classes**

```
lambda  → anonymous function
map     → function/method
def     → function/method
```

```
class C1():

    def f1(a,b):
        print(a+b)


class C2():

    def f2(a,b):
        print(a*b)


class C3(C1, C2):

    def f3(a,b):
        print(a**b)
```



functions and **classes**

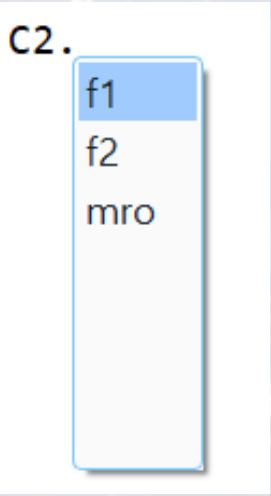
`lambda` → anonymous function
`map` → function/method
`def` → **function/method**

so far, we inherited methods from a super (aka *parent*) class to a subclass (aka *child*) via

```
class C1():

    def f1(a,b):
        print(a+b)


class C2(C1):

    def f2(a,b):
        print(a*b)
```

What if we want **f1** *directly* in **f2**

→ for example, because **f2** is an updated version of **f1** and we don't want to copy/paste code



functions and **classes**

self

```
class C1():

    def f1(a,b):
        print(a+b)


class C2():

    def f2(self, a, b):
        print(a+b)
```

`lambda` → anonymous function
`map` → function/method
`def` → **function/method**

```
In [82]: C1.f1(1,2)
3
```

```
In [84]: C2.f2(1,2)
Traceback (most recent call last):

  Cell In[84], line 1
    C2.f2(1,2)

TypeError: C2.f2() missing 1 required positional argument: 'b'
```

```
In [85]: InstC2 = C2()

In [86]: InstC2.f2(1,2)
2
```

functions and classes

lambda → anonymous function
map → function/method
def → **function/method**

```
import numpy as np

class Encoder():

    def __init__(self):

        NT      = ['A', 'C', 'G', 'T']
        Code    = np.eye(4)

        Dict    = {key: value for key, value in zip(NT, Code)}

        self.Enc = lambda Sequence: [Dict[s] for s in Sequence]

        #return(self.Enc)

    def Encode(self, Sequence):

        print(self.Enc(Sequence))
```

the **dunder method** **__init__**

we want to establish the encoder when we initialize our class

calling the encoder for any instance (see parallel programming)

functions and classes

lambda → anonymous function
map → function/method
def → **function/method**

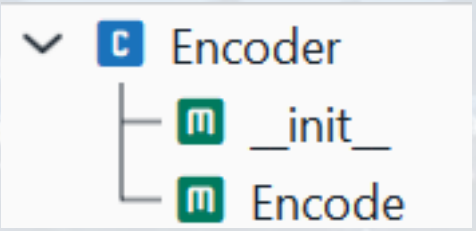
```
import numpy as np

class Encoder():

        def __init__(self):

                NT      = ['A', 'C', 'G', 'T']
                Code    = np.eye(4)

                Dict    = {key: value for key, value in zip(NT, Code)}

                self.Enc = lambda Sequence: [Dict[s] for s in Sequence]

                #return(self.Enc)

        def Encode(self, Sequence):

                print(self.Enc(Sequence))
```

functions and classes

lambda → anonymous function
map → function/method
def → **function/method**

```
In [92]: MyEncoder = Encoder()

In [93]: MyEncoder.Encode('ACCTTGGTA')
[[1, 0, 0, 0], [0, 1, 0, 0], [0, 1, 0, 0], [0, 0, 0, 1],
[0, 0, 0, 1], [0, 0, 1, 0], [0, 0, 1, 0], [0, 0, 0, 1], [1,
0, 0, 0]]
```

```
import numpy as np

class Encoder():

        def __init__(self):

                NT      = ['A', 'C', 'G', 'T']
                Code    = np.eye(4)

                Dict    = {key: value for key, value in zip(NT, Code)}

                self.Enc = lambda Sequence: [Dict[s] for s in Sequence]

                #return(self.Enc)

        def Encode(self, Sequence):

                print(self.Enc(Sequence))
```



functions and `classes` & the `super()` method

`lambda` → anonymous function
`map` → function/method
`def` → **function/method**

```
class C1():

        def f1(self, a, b):
            res = a + b

        return res


class C2(C1):

        def f2(self, a, b, c, d):
            res = super().f1(a,b)

         print(c*d + res)
```

What if we want **f1** *directly* in **f2**

→ for example, because f2 is an updated version of f1 and we don't want to copy/paste code

we actually would need *self* here too, but it is included in `super` per default

functions and classes & the super() method

lambda → anonymous function
map → function/method
def → **function/method**

```
class C1():

        def f1(self, a, b):
                res = a + b

                return res


class C2(C1):

        def f2(self, a, b, c, d):
                res = super().f1(a,b)
                print(c*d + res)
```

```
In [103]: c = C2()

In [104]: c.f2(1,2,3,4)
15
```

What if we want **f1** *directly* in **f2**

→ for example, because f2 is an updated version of f1 and we don't want to copy/paste code

1st: calling an `instance` of C2 (hence `super` is perfect for inheriting **`__init__`**

f2 is now executing **f1** too
→ code is more compact
→ less error prone

functions and classes & the super() method

```
lambda  → anonymous function
map     → function/method
def     → function/method
```

```
class C1():

    def f1(self,a,b):
        print(a+b)


class C2(C1):

    def f2(self,a,b):
        super().f1(a,b)
        print(a*b)


class C3(C2):

    def f3(self,a,b):
        super().f2(a,b)
        print(a**b)
```

We can now go on and inherit as we did before:

```
In [112]: c = C3()

In [113]: c.f3(2,3)
5
6
8
```

functions and `classes` & the `super()` method

`lambda` → anonymous function
`map` → function/method
`def` → **function/method**

```
class C1():

    def f(self,a,b):
        print(a+b)


class C2(C1):

    def f(self,a,b):
        super().f(a,b)
        print(a*b)


class C3(C2):

    def f(self,a,b):
        super().f(a,b)
        print(a**b)
```

In practice, the fs have usually the same name!

```
In [112]: c = C3()

In [113]: c.f3(2,3)
5
6
8
```

functions and classes & the super() method

lambda → anonymous function
map → function/method
def → **function/method**

```
class C1():

    def f(self,a,b):
        print(a+b)


class C2(C1):

    def f(self,a,b):
        #super().f(a,b)
        print(a*b)


class C3(C2, C1):

    def f(self,a,b):
        super().f(a,b)
        print(a**b)
```

In practice, the fs have usually the same name!

it did not inherit the first **f**!

```
In [124]: c = C3()

In [125]: c.f(2,3)
6
8
```

functions and `classes` & the `super()` method

`lambda` → anonymous function
`map` → function/method
`def` → **function/method**

```
class C1():

    def f(self,a,b):
        print(a+b)


class C2():

    def f(self,a,b):
        #super().f(a,b)
        print(a*b)


class C3(C1, C2):

    def f(self,a,b):
        super().f(a,b)
        print(a**b)
```

In practice, the fs have usually the same name!

this time it did not inherit the *second* **f**!

```
In [128]: c = C3()

In [129]: c.f(2,3)
5
8
```

functions and `classes` & the `super()` method

`lambda` → anonymous function
`map` → function/method
`def` → **function/method**

`super` refers to the **next class in line**!
Always check *mro* (**m**ethod **r**esolution **o**rder) in order to see which path inheritance went!

```
class C3(C2, C1):

    def f(self,a,b):
        super().f(a,b)
        print(a**b)
```

```
C3.mro()
[__main__.C3, __main__.C2, __main__.C1, object]
```

```
class C3(C1, C2):

    def f(self,a,b):
        super().f(a,b)
        print(a**b)
```

```
C3.mro()
[__main__.C3, __main__.C1, __main__.C2, object]
```

functions and `classes` & the `super()` method

`lambda` → anonymous function
`map` → function/method
`def` → **function/method**

```
class C1:
    def f(self, a, b):
        print(a + b)

class C2:
    def f(self, a, b):
        print(a * b)

class C3(C2, C1):

    def f(self, a, b):
        super(C2, self).f(a, b)

        super().f(a, b)

        print(a ** b)
```

```
In [112]: c = C3()

In [113]: c.f3(2,3)
5
6
8
```

Check mro!

functions and classes homework

`lambda` → anonymous function
`map` → function/method
`def` → **function/method**

```
import numpy as np

class Encoder():

        def __init__(self):

                        NT      = ['A', 'C', 'G', 'T']
                        Code    = np.eye(4)

                        Dict    = {key: value for key, value in zip(NT, Code)}

                        self.Enc = lambda Sequence: [Dict[s] for s in Sequence]

                return(self.Enc)

        def Encode(self, Sequence):

                        print(self.Enc(Sequence))
```

functions

lambda → anonymous function
map → function/method
def → **function/method**

```
class Map(Encoder):

    def __init__(self):
        self.Enc = super().__init__()

    def fun(self, s):
        NT   = ['A', 'C', 'G', 'T']
        Code = [[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,0,1]]
        Dict = {key: value for key, value in zip(NT,Code)}

        return Dict[s]

    def runboth(self, Sequence):

        print(list(map(self.Enc, Sequence))) #mapping anonymus function
        print(list(map(self.fun, Sequence))) #mapping actual function
```

We want to inherit the ***Enc*** method from ***Encoder***

That would work already!

functions

`lambda` → anonymous function
`map` → function/method
`def` → **function/method**

```
class Map(Encoder):

    def __init__(self):
        self.Enc = super().__init__()

    def fun(self, s):
        NT   = ['A', 'C', 'G', 'T']
        Code = [[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,0,1]]
        Dict = {key: value for key, value in zip(NT,Code)}

        return Dict[s]

    def runboth(self, Sequence):

        print(list(map(self.Enc, Sequence))) #mapping anonymous function
        print(list(map(self.fun, Sequence))) #mapping actual function
```

```
In [162]: M = Map()

In [163]: M.runboth('ACCTG')
[[[1, 0, 0, 0]], [[0, 1, 0, 0]], [[0, 1, 0, 0]], [[0, 0, 0, 1]], [[0, 0, 1, 0]]]
[[1, 0, 0, 0], [0, 1, 0, 0], [0, 1, 0, 0], [0, 0, 0, 1], [0, 0, 1, 0]]
```

Introduction to Unix & Python

Thank you for your attention!