LEARN CODING

ale66

github.com/ale66/learn-coding

LAMBDA ETC.

MAPPING FUNCTION ON SEQUENCES

```
my_values = [1, 2, 3, 4, 5]
```

Compute the squares of the first five non-zero natural numbers

```
l = len(my_values)

squared_values = []

for i in range(l):
    s = my_values[i]**2

    squared_values.append(s)

print(squared_values)

[1, 4, 9, 16, 25]
```

Compute the squares of those integers inside `my_values`

```
squared_values = list()

for val in my_values:
    squared_values.append(val**2)

print(squared_values)

[1, 4, 9, 16, 25]
```

A local, *list processing* activity

FUNCTORS

```python
def my_square(base: int)->int:

    return base**2
```

here `my_square` is a *functor*: the name of the function itself

MAPPING FUNCTIONS TO LISTS

Given a function name, we can build an iterable

that will return the results of applying such function to a given list (mostly)

```
my_iterable = map(my_square, my_list)
```

We 'query' `my_iterable` we get a sequence with 1, 4, 9 etc.

```
my_iterable = map(my_square, my_list)

for el in my_iterable:
    print(el)
```

Maps can be easily *materialised* into proper lists

```
# a map object needs to be wrapped into a data structure
list(map(my_square, my_values))

[1, 4, 9, 16, 25]
```

Essentially, we are pushing a (very standard) `for` cycle down into the Python runtime management

LIST COMPREHENSIONS

Define lists by the properties of their members, not by explicit membership

Styles:

- *extensional*: name the items that make up the list
- *intensional*: give the formula/condition that items must satisfy to be into the list

In Mathematics:

$$\{1, 4, 9, 16, 25\} = \{x : \exists y \in \mathcal{N} . x = y^2 \wedge 0 < y \leq 5\}$$

list comprehension brings list definition closer to natural language specification

```python
extensional = [1, 2, 3, 4, 5]

# create a new list
intensional = [x**2 for x in extensional]
```

A shorter notation, closer to Mathematics

(Data Science code tends to avoid nesting of `for` and `while` cycles)

EXAMPLES

with strings, looking for names beginning by ‘A’

```
names = ['Alessandro','Alberto', 'Erin', 'Nicola']

A_names = [n for n in names if n[0] == 'A']

['Alessandro', 'Alberto']
```

List comprehension can be nested, iterating twice in the same line of code

Here we look for names containing ‘o’

```
o_in_name = [n for n in names for ch in n if ch == 'o']

['Alessandro', 'Alberto', 'Nicola']
```

this nested list comprehension ‘saves’ us from two nested cycles

LAMBDA DEFINITIONS

TRADITIONAL FUNCTION DEFINITION

Example: compute the $V_\circ = \frac{4}{3}\pi r^3$ formula from school

```python
def my_vol(radius: int) -> float:
    '''Traditional function definition
       to compute the volume of a sphere'''

    import math

    volume = 4/3 * math.pi * radius**3

    return volume
```

`my_vol()` can be applied anywhere, and several times.

But for a one-time application, possibly deep down the code, the `my_vol` functor may not be needed after all

IN-LINE, ANONYMOUS FUNCTIONS

There is no functor, the name of the function is the function definition itself

```
lambda x: x**2
```

- `lambda` defines the input
- the output is implicit with the evaluation of the formula`

```
(lambda x: x**2)(3)
```

will return (or be substituted with) 9.

Look at the (non-) differences:

```
print(my_square(3))

print((lambda x: x**2)(3))
```

MAPPING LAMBDAS

The intended application of lambdas

```
my_values = [1, 2, 3, 4, 5]

new_squares = map(lambda x: x**2, my_values)
```

maps the lambda definition onto each element of the list

n.b. `new_squares` is an iterator, not a real list

PUTTING IT ALL TOGETHER

```
list(map(lambda x: x**2, my_values))
```

makes the results into a list.

```
new_squares = map(lambda x: x**2, my_values)

print(list(new_squares))
```

compare the lambda mapping to non-lambda solutions for succintness

```python
def allSquares(input_list: list) -> list:
    '''Squares all values of a given list'''

    squares = list()

    for el in input_list:
        squares.append(el**2)

    return squares

print(allSquares(my_values))
```

DEFAULT AND VARIABLE ARGS

Normally, argument passing is positional:

$$\log_{10} 1000 = 3$$

$$\log_2 1024 = 10$$

```python
def my_log(base: float, exponent: float)->float:
    ''' a wrap around the standard log function'''
    import math

    return math.log(base, exponent)

my_log(1000, 10)
my_log(1024, 2)
3
10
```

GREAT FLEXIBILITY

P. functions allow calls with a variable number of parameters

Example: by *log* often the logarithm in base 10 is intended: we save time and have better clarity by

- assuming that 10 is the default base
- allow calls like *mylog(1000)*

Rule: positional argument first, then arguments with default

DEFAULT VALUES

Assumed values are described in the `def` part

Positional argument must be defined before default argument

```python
def mylog(argument, base = 10):
    import math

    return math.log(argument, base)
```

```python
print(mylog(1000))

3

print(mylog(1024))

3.0103
```

Note: for **numpy** the default base is e: $\log_e x = \ln x$.

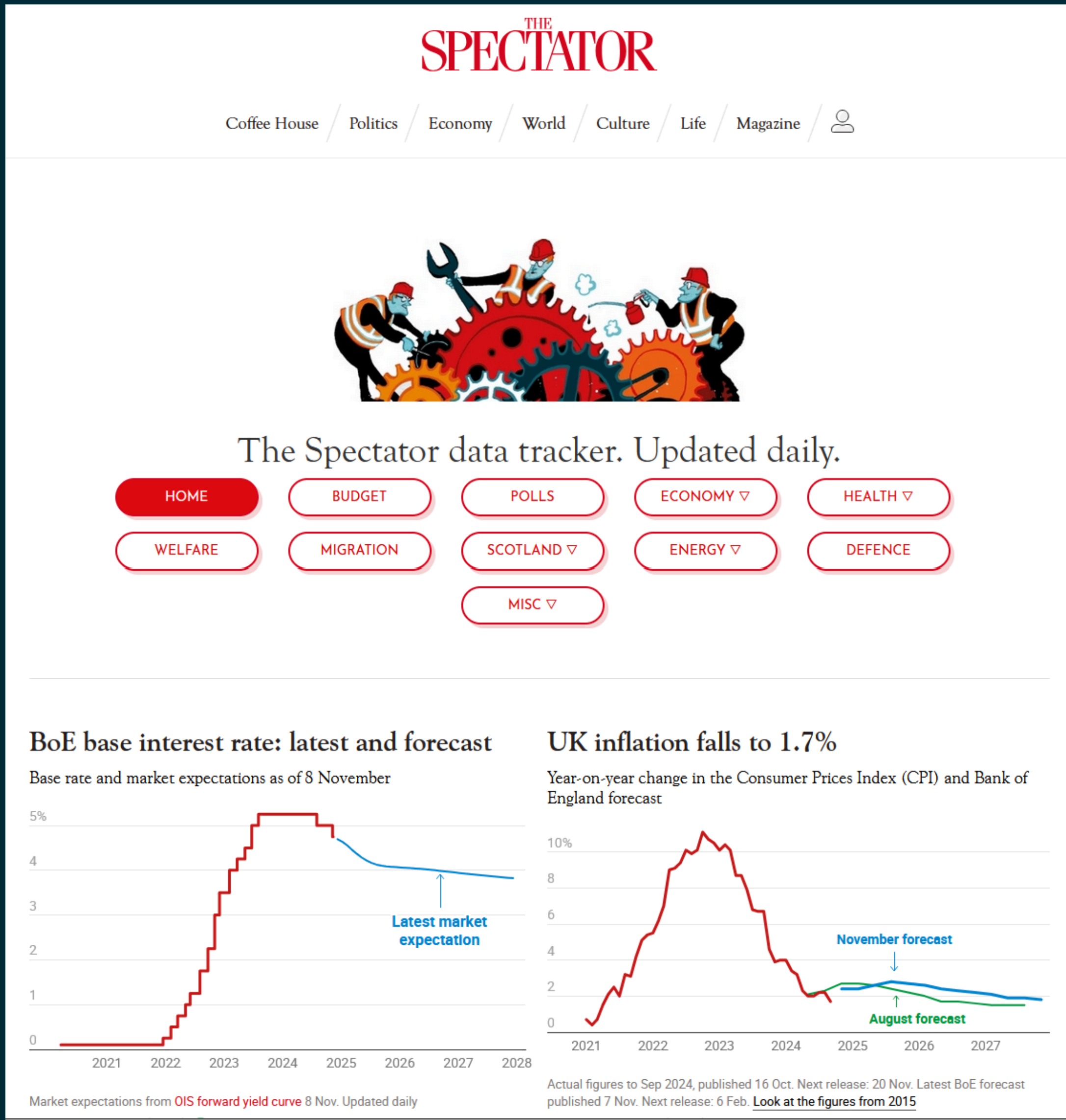
INSPECTING THE PASSED VALUES

A function can examine the values received with the call and decide a course of action

```python
def myfunc(*args):
    ''' Study the received values '''

    for a in args:
        print(a)
```

LEARN WITH DATA CHALLENGES



can you get fresh data and display it?

github.com/ale66/learn-coding