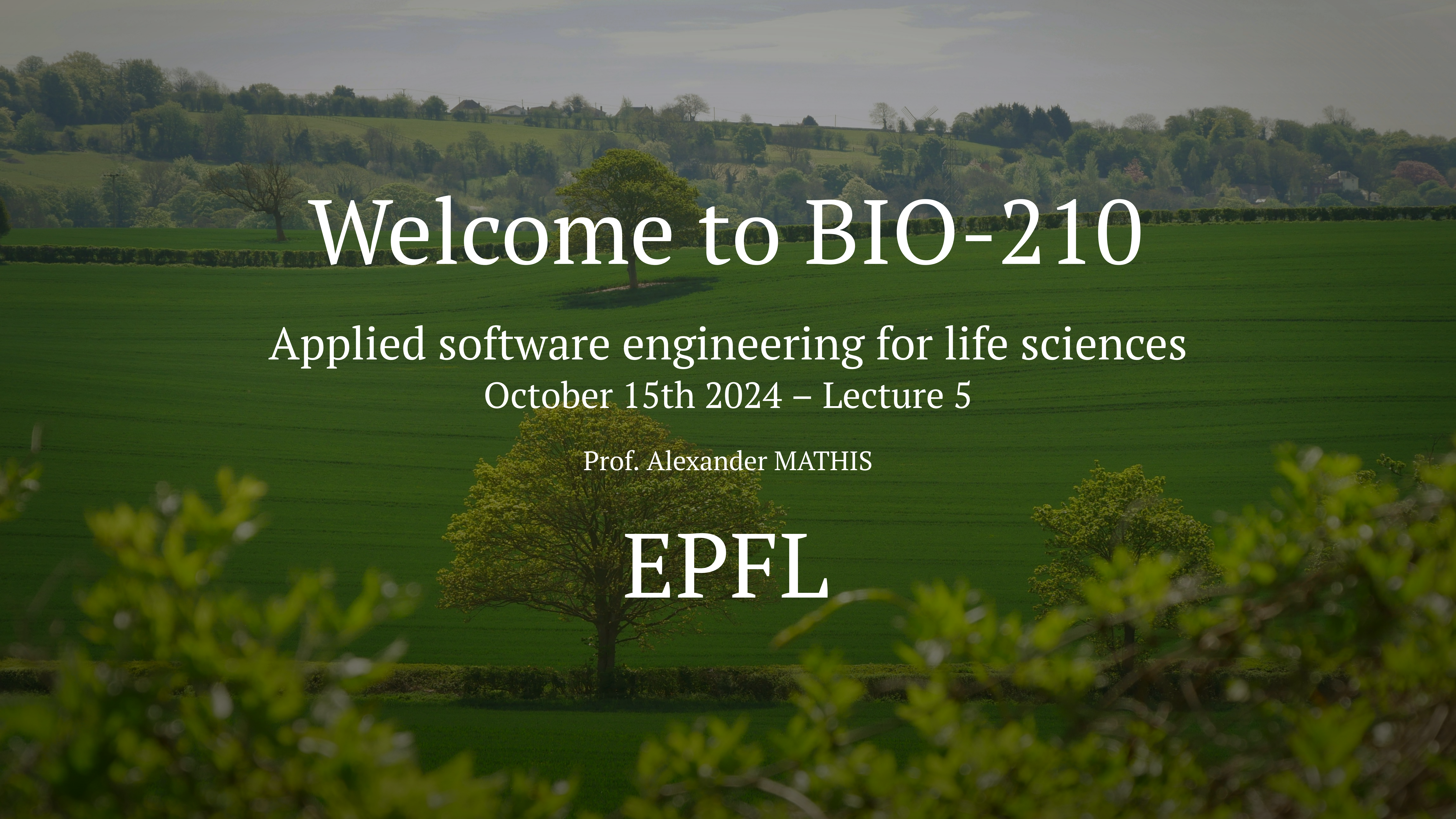


Welcome to BIO-210

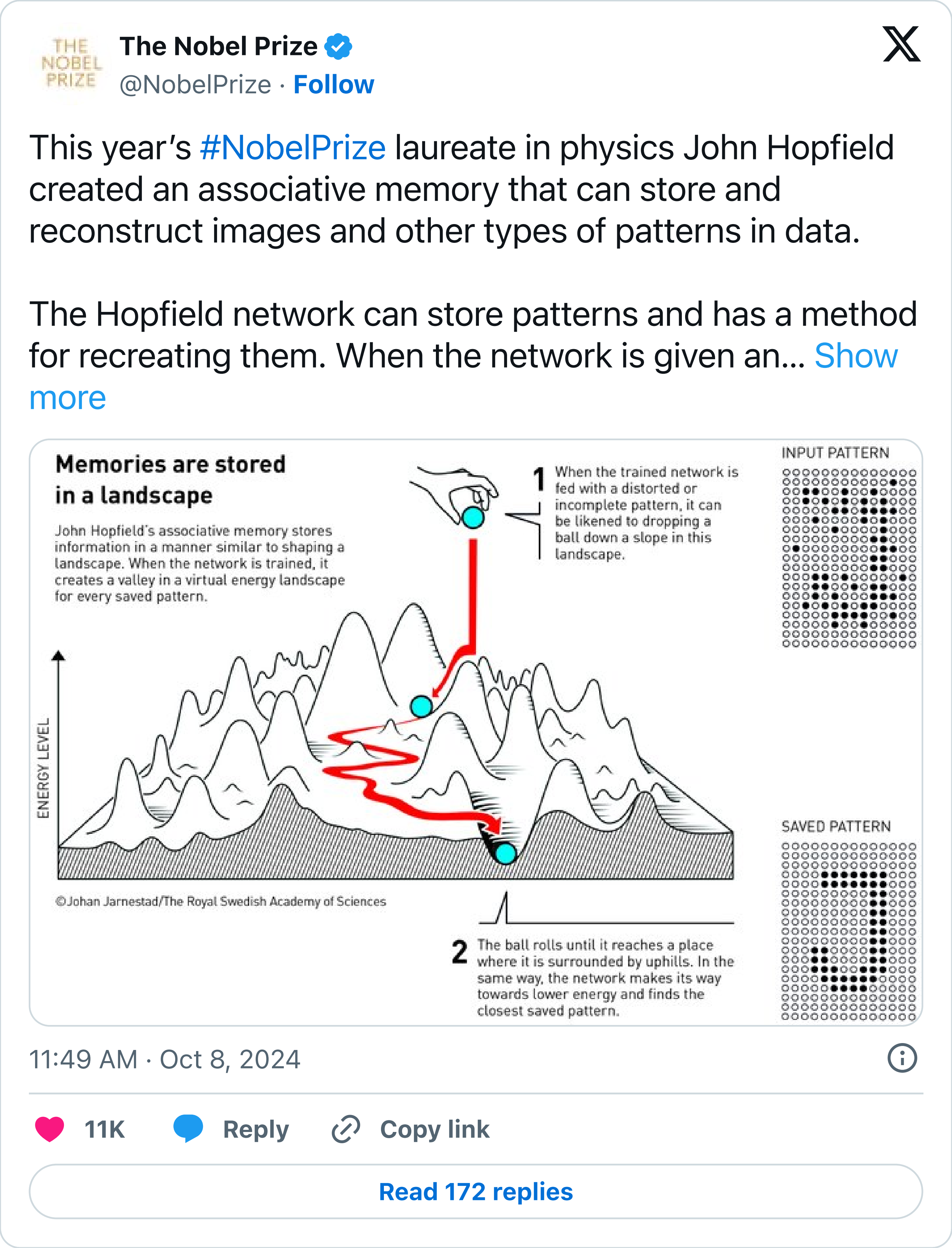
Applied software engineering for life sciences

October 15th 2024 – Lecture 5

Prof. Alexander MATHIS

EPFL

Congrats to John Hopfield!



The Nobel Prize
@NobelPrize · Follow

This year's #NobelPrize laureate in physics John Hopfield created an associative memory that can store and reconstruct images and other types of patterns in data.

The Hopfield network can store patterns and has a method for recreating them. When the network is given an... Show more

Memories are stored in a landscape

John Hopfield's associative memory stores information in a manner similar to shaping a landscape. When the network is trained, it creates a valley in a virtual energy landscape for every saved pattern.

1 When the trained network is fed with a distorted or incomplete pattern, it can be likened to dropping a ball down a slope in this landscape.

2 The ball rolls until it reaches a place where it is surrounded by uphills. In the same way, the network makes its way towards lower energy and finds the closest saved pattern.

©Johan Jarnestad/The Royal Swedish Academy of Sciences

11:49 AM · Oct 8, 2024

Announcements I

- Congrats on your great quiz results! Class average 9.7/10 (before the correction below).
- We had one typos in question 7 (most got it right anyway). However, as a consequence, everyone gets full points on this question.

Namely for the question, what is the output of the following code:

```
mygrades = {"Linear Algebra" : 3.5, "Analysis": 3, "Physics": 2.5, "SHS" :4}
for course, grade in mygrades.items():
    grade = 6
print(mygrades)
```

The "correct" answer contained an erroneous semicolon:

```
{"Linear Algebra" : 3.5; "Analysis": 3, "Physics": 2.5, "SHS" :4}
```

Announcements II

- Today is the first in person quiz: please come *in time*, there will be no extra time. Submission closes at 13:35. To start, you'll need to sign in. Bring your Camipro. No notes are allowed. If you switch to a different tab from Moodle's quiz or communicate with somebody, you'll receive 0 points.

- Monday 15:15 - 16: my office hours at SV 2811

Announcements III

- Please note that we altered the room assignment for the exercises slightly. Check here and go to the correct room from now on. NOTE: you need your EPFL login to see it!
- Did you release your code, v1?
- v1 of your project was due at 10am today (not graded/checked), check out release guide.
- Note: For learning its better if you collaborate via git. But, if multiple team members contribute to a commit, add all authors to your commit message:

```
>>> git add .           # staging all files
>>> git commit -m "Adding testing symmetry of Hopfield weights

Some more content ...
Co-authored-by: Lucas Stoffl <lucas.stoffl@epfl.ch>
Co-authored-by: Mu Zhou <use_the_email_of_the_github_accuount@epfl.ch>
```

Git tutorial video

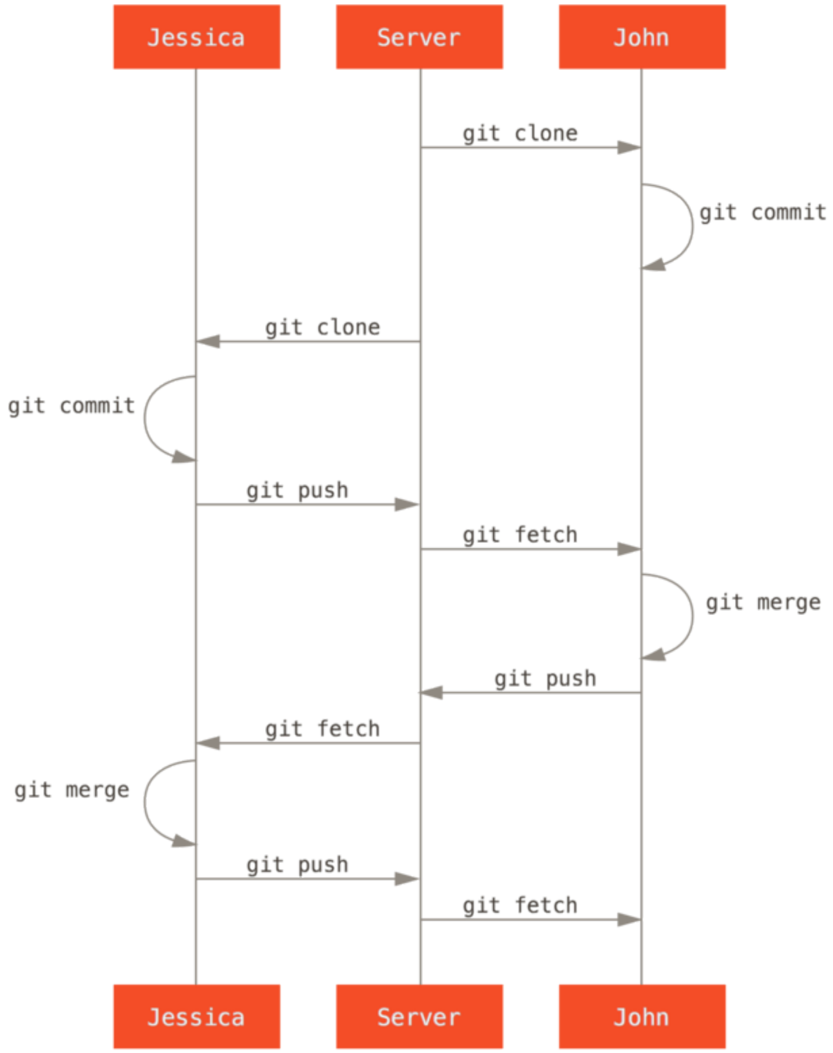


Image source: git docs

Jennifer Shan (one of the SAs) developed a video tutorial on how to use git in Visual Studio Code

Viva Berlenghi (one of the SAs) wrote a Git survival kit

How to ask a question well on the forum/ED?

It’s an important skill to ask good questions.

- (first check docs, e.g., in `` `ipython` ``: `` `range?` `` and python docs)
- (first search if the question was already asked)
- Write a title that summarizes the specific problem
  - Bad: C# Math Confusion
  - …
  - Good: Why does str == "value" evaluate to false when str is set to "value"?
- Introduce the problem before you post any code
- Help others reproduce the problem
- Proofread before posting
- Respond to feedback after posting, did it resolve your question?

Tips from Stackoverflow

Functions

- so far we have mostly written procedural Python statements/programs
- a `` `function` `` can group a set of statements so that they can be run more than once in programs
- `` `functions` `` are packaged procedures with a name
- `` `functions` `` also can compute a result based on parameters that we can specify
- Coding procedures/operations as `` `functions` `` makes them re-usable

Rule of thumb

Every time you copy/paste some statements, make a function!

Functions are one of the most basic Python structures for maximizing code-reuse

Why should you use functions?

- maximize code reuse and minimize redundancy (thus reducing maintenance effort)
- prodecural decomposition (splitting programs into well-defined roles)
- it’s easier to implement smaller tasks in isolation (rather than the whole process at once)

Function-related statements and expressions

Statement or expression	Examples
Call expression	myfunc('Seppl',175,age=22,*rest)

`def`	def printer(message):
	    print('Hello'+message)

`return`	def adder(a,b=1,*c):
	    return a+b+c[0]

`global`	x = 'outside'
	def changer():
	    global x; x= 'new'

`lambda`	`func = [lambda x: x**2, lambda x: x**3]`

Function basics

- The keyword `` `def` `` is an executable statement
- The keyword `` `def` `` creates an object and assigns it a name
- Functions *only exist, once* Python reaches `` `def` ``
- Functions really behave like other objects, they can be re-assigned, stored in lists etc.
- The keyword `` `return` `` sends a result back to the caller. When a function is called, the caller stops until the function is done and returns control to the caller. Functions that compute a value send it back to caller with a return statement (i.e., the result of the function call).
- Functions without `` `return` `` statements, return `` `None` `` (upon completion)

Def statements

```
def function_name(arg1, arg2, ..., argN):
  statements1
  statements2

statement_not_part_of_function    # Added just for illustration
```

- A function's body is indented. This code is run when the function is called.
- For a single statement, one can use `` `;` `` and place the code in one line, e.g. `` `def f(x): return x;` ``
- The keyword `` `def` `` specifies the function name and a list of zero or more `` `arguments` `` in parentheses
- The function arguments are assigned to objects passed, when the function is called (not upon definition). I.e., here `` `arg1` `` does not need to exist (see Example 0).

Return statements

```python
def name(arg1, arg2, ..., argN):
  statements
  ...
  return value
```

- Return can be anywhere in the function body (or exist multiple times, e.g. Example 2)
- If there is no return, `` `None` `` will be returned

Example 0: defs are not calls

```
In [1]: x
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-7-6fcf9dfbd479> in <module>
----> 1 x

NameError: name 'x' is not defined

In [2]: def f(x):               # Create and assign function
   ...:     return x            # Body executed when called
   ...:
 In [3]: f(2)                   # Arguments are passed in parentheses
 Out[3]: 2
 In [4]: f(x)
 ---------------------------------------------------------------------------
 NameError                                 Traceback (most recent call last)
 <ipython-input-10-f2d123ee1505> in <module>
 ----> 1 f(x)

 NameError: name 'x' is not defined
```

Note: even though `x` appears in the definition of `f`, `f` is not called.

Example 1: Functions are flexible

```
In [1]: def f1(x):          #
   ...:     return x**2
   ...:
In [2]: def f2(x):
   ...:     return x**4
   ...:
In [3]: f = [f1,f2]           # combine in a list!
In [4]: f(2)                  # f is a list, cannot be called...
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-c510dc86724b> in <module>
----> 1 f(2)

TypeError: 'list' object is not callable
In [5]: f[0](2)               # Index first element, then pass 2, returns 2**2 = 4
Out[5]: 4
In [6]: f[0](3)
Out[6]: 9
In [7]: f[1](3)               # Calling f2, via f[1] (shared object)
Out[7]: 81
```

Example 2: A strange function

```
In [1]: def strange_fun(arg1):
    ...:   if arg1>0:
    ...:     return arg1
    ...:   elif arg1<0:
    ...:     return -1*arg1
    ...:

In [2]: type(strange_fun(1.2))
Out[2]: float
In [3]: type(strange_fun(1))
Out[3]: int
In [4]: type(strange_fun(0))      # returns None, as no return exists for arg1=0
Out[4]: NoneType
```

Quiz: What is the result?

```
x=2
if x>3:
  def func(x):
    return 3*x
elif x<3:
  def func(x):
    return 2*x, 0

result = func(2)
```

The code will set result to: `(4,0)`.

Note `def` executes at run-time. You do not need to define func like in C.

Quiz: What is the result?

```
In [1]: x=5
   ...: if x>3:
   ...:   print(x)
   ...:
   ...: elif x<3:
   ...:   def func(x):
   ...:     return 2*x, 0
   ...:
   ...: result = func(2)
```

```
5
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[1], line 9
      6   def func(x):
      7     return 2*x, 0
----> 9 result = func(2)

NameError: name 'func' is not defined
```

Quiz: what is printed when this program runs?

```python
x = 'abc'
def func():
    x = 'xyz'

func()
print(x)
```

It prints `` `abc` ``, as `` `x` `` inside func() is a local variable.

This local variable, thus does not affect the global variable x=`` `abc` `` (see scoping, later in this lecture)

Functions are typeless, and general!

Defintion:

```
In [1]: def times(x,y):
   ...:     return x*y
   ...:
```

Calls:

```
In [2]: times(2,3)                  # arguments in parentheses
Out[2]: 6
In [3]: times(1.0,4)
Out[3]: 4.0                         # results are casted (type converted)
In [4]: times("La",3)               # functions are "typeless"!
Out[4]: 'LaLaLa'                    # polymorphism
In [5]: times([1,2],4)
Out[5]: [1, 2, 1, 2, 1, 2, 1, 2]
In [6]: my_list = times([1,2],4)    # save the result object
```

Scope

- when you use a name in a program, Python creates, or looks up the name in the `` `namespace` ``
- scope refers to a `` `namespace` ``. Where you assign a name, determines the scope of a name’s visibility
- apart from packaging code for reuse, functions add an extra namespace layer to your programs
- Python has four levels of namespaces:
  - builtins
  - global
  - local
  - enclosing

The built-in namespace

The built-in namespace contains the names of all of Python’s built-in objects.

```
 In [1]: print(dir(__builtins__))      # you can list them like this!
...:
['ArithmeticError', 'AssertionError', 'AttributeError',
 'BaseException','BlockingIOError', 'BrokenPipeError', 'BufferError',
 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError',
 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError',
 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
 'Exception', 'False', 'FileExistsError', 'FileNotFoundError',
 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError',
 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError',
 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt',
 'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None',
 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError',
 'OverflowError', 'PendingDeprecationWarning', 'PermissionError',
 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning',
 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration',
 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError',
 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError',
 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError',
 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError',
 'Warning', 'ZeroDivisionError', '_', '__build_class__', '__debug__',
 '__doc__', '__import__', '__loader__', '__name__', '__package__',
```

The global namespace

- the global namespace contains all names defined at the level of the main program
- it is created when the main program starts (and exists until the interpreter terminates)

```
(base) alex@mac Code % python
Python 3.8.8 (default, Apr 13 2021, 12:59:45)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> globals()      # list the global variables
{'__name__': '__main__', '__doc__': None, '__package__': None,
'__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
 '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>}
>>> a=3
>>> globals()      # a was added
{'__name__': '__main__', '__doc__': None, '__package__': None,
'__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
'__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, 'a': 3}
```

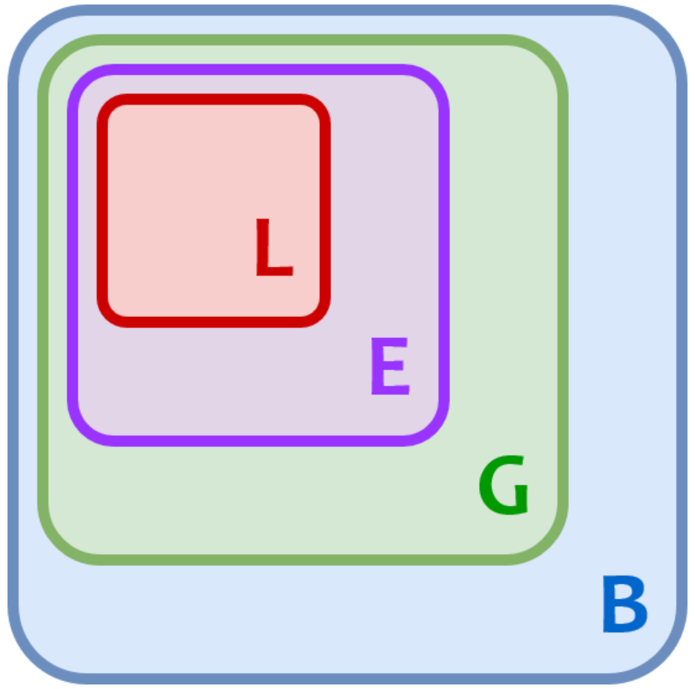
Local and enclosing namespaces

```
In [2]: def f():      # f, the enclosing function of g
   ...:     print('Start f()')
   ...:
   ...:     def g():  # definition of enclosed function g
   ...:         print('Start g()')
   ...:         print('End g()')
   ...:         return
   ...:     g()       # call g()
   ...:
   ...:     print('End f()')
   ...:     return
   ...:
   ...: f()         # Calling f()
Start f()           # Now Python creates a namespace for f()
Start g()           # A new namespace for g() is created
End g()
End f()
```

Here g’s namespace is called local namespace, and f’s namespace is called enclosing namespace (as f is the enclosed function). Each of these namespaces remains in existence until its respective function terminates.

Variable scope: LEGB rule

- **Local:** If you refer to `x` inside a function, then the interpreter first searches for it in the innermost scope that's local to that function.
- **Enclosing:** If `x` is not in the local scope, but appears in a function that resides inside another function, then the interpreter searches in the enclosing function's scope.
- **Global:** If neither of the above searches is fruitful, then the interpreter looks in the global scope next.
- **Built-in:** If the interpreter cannot find `x` anywhere else, then the interpreter tries the built-in scope.



Source: Python docs

Questions?

Quiz: what does this code print and why?

```python
x = 'abc'
def func():
    x = 'xyz'
    print(x)

func()
print(x)
```

It prints `` `xyz` ``, then `` `abc` ``, as the reference in func() returns the value `` `xyz` `` and the reference at the end returns the value of the variable in the global namespace.

Global statement

The `global` statement is one of the only statements, that remotely resembles a declaration statement in Python. However, `global` is not a size-declaration, but a namespace declaration.

```
In [1]: x = 11      # global x
   ...: def func():
   ...:   global x
   ...:   x=99      # global x, within function namescape assignment
   ...:
   ...: func()
   ...: print(x)
99
```

See more, incl. on `nonlocal`, which can be required for enclosed functions, in docs.

Quiz: what is printed when this program runs?

```python
x = 'abc'
def func():
    print(x)

func()
print(x)
```

It prints `abc`, then `abc`, as the reference in func() looks up x in the global scope and the last print refers to the global `x`.

Quiz: what does this code print and why?

```python
x = 'abc'
def func():
    global x
    x = 'xyz'
    print(x)

func()
print(x)
```

It prints `` `xyz` ``, then `` `xyz` ``.

As the reference in func() returns the value `` `xyz` `` and when func is called, the value of `` `x` `` is overwritten.

Quiz: what does this code print and why?

```python
x = 'abc'
def func():
    global x
    x = 'xyz'
    print(x)

print(x)
```

It just prints `` `abc` `` – nothing will be overwritten, as `` `func()` `` is never called.

However, try to avoid globals...

Consider this example:

```python
X = 99
def f():
  global X
  X = 77

def g():
  global X
  X = 33

```

Why, should you avoid globals?

Here, the value of X is timing dependent, it depends on which function was called last.

Now imagine you want to modify and reuse this code...

Conclusion of scope

Keep in mind that

where you define a name, determines much of its meaning (in functions, modules, etc.)

A simple function factory

Factory functions (a.k.a. closures) are sometimes used to generate handlers on the fly in response to some condition at runtime.

```
In [1]: def maker(N):
   ...:   def action(X):      # Make and return action
   ...:     return X**N       # action retains N from enclosing scope
   ...:   return action
   ...:
```

```
In [2]: square = maker(2)     # Pass 2 to N
   ...: cube = maker(3)       # Pass 3, note cube remembers 3; square 2!
In [3]: square(2)             # 2**2
Out[3]: 4

In [4]: cube(3)               # 3**3
Out[4]: 27
In [5]: maker(5)(2)           # 2**5
Out[5]: 32
```

The created function, retains the state of N.

Argument passing

- arguments are passed by automatically assigning objects to local variables (because references are implemented as pointers, arguments are passed by pointers)
- assignments to argument names in a function, do not affect the caller
- however, changing a mutable object in a function, may impact the caller

Arguments and shared references

```
In [1]: def f(a):               # a is assigned to (references; i.e. the passed obj.)
   ...:   a = 99                # Changes local variable a only
   ...: b = 88
   ...: f(b)                    # a and b both reference 88 (initially)
   ...: print(b)                # b is unchanged
88
```

However, if mutable objects (such as lists, dicts,...) are passed, aliasing can happen!

```
In [1]: def f(a,b):   # arguments assigned references to objects
   ...:   a=99        # changes local name's value only
   ...:   b[0]=22     # changes shared object in place!
   ...: A = 1
   ...: B = ['hello',2]
   ...: f(A,B)        # caller
   ...: print(A,B)    # A is unchanged, B is different
1 [22, 2]
```

Note this is the *canonical* Python behavior!

```
In [1]: x = 1
   ...: a = x     # x and a share the same object
   ...: a = 2     # resets `a` only, `x` is still 1.
   ...: print(x)
1
```

```
In [2]: x = [1,2]
   ...: a = x        # x and a share the same object
   ...: a[0] = 2     # in-place change to a; x affected!
   ...: print(x)
   ...:
[2, 2]
```

Avoiding mutable argument changes

Method 1 (pass a copy):

```
In [3]: def f(a,b):
    ...:    a, b[0]=99,22        #  You can assign in parallel
    ...:    print("inside f", a,b)
    ...: A, B = 1, ['hello',2]
    ...: f(A,B.copy())           # Caller (pass a copy), also B[:]
    ...: print("outside",A,B)    # A and B are unchanged
inside f 99 [22, 2]
outside 1 ['hello', 2]
```

Method 2 (copy input):

```
In [4]: def f(a,b):
    ...:    b = b[:]            # copy input to not impact caller
    ...:    a, b[0]=99,22
    ...:    print("inside f", a,b)
    ...: f(A,B)
    ...: print("outside",A,B)
inside f 99 [22, 2]
outside 1 ['hello', 2]
```

Argument-matching modes

Python functions allow highly flexible calling patterns for functions

- *Positionals*: matched left to right (standard mode seen so far)
- *Keywords*: matched by argument name; `` `name = value` `` syntax
- *Defaults*: specify values for optional arguments (that do not need to be passed)
- *Varargs collecting*: pass arbitrarily many positional or keyword arguments

Let’s look at some examples … and discuss *varargs* collecting later in the course.

Keyword examples

```
>>> def f(x,y,z): print(x,y,z);
>>> f(1,2,3)        # passing by position
(1,2,3)

# Using keywords
>>> f(z=3,y=2,x=1)  # match by name
(1,2,3)

# Mixed type
>>> f(1,z=3,y=2)    # x gets assigned 1 by position, others by name
(1,2,3)
```

Why using the keyword mode?

To better document code, which goes hand in hand with better variable names, e.g. `` `process_user(name='Franz',age=22,job='EPFL student')` ``

gives a good idea what this code might do.

Default examples

```
>>> def f(x,y=2,z=3): print(x,y,z);    # x required, y and z optional!
>>> f(1)         # using defaults
(1,2,3)

>>> f(1,4)       # overwriting defaults by positional variable
(1,4,3)
>>> f(1,4,5)
(1,4,5)
# Mixed keyword and default example:
>>> f(1,z=55)    # x gets 1 by position, others by name
(1,2,55)
>>> f(y=2)       # positional arguments need to be passed
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[3], line 1
----> 1 f(y=2)

TypeError: f() missing 1 required positional argument: 'x'
```

Default, is a very flexible, core Python feature. We have seen it in use for many functions, e.g.: `range`, `np.arange`, `np.linspace`, …

Quiz: How do you get 11 equidistant numbers from 0 to 1?

Variant 1:

```
In [1]: import numpy as np
In [2]: X = np.linspace(0,1,11)
In [3]: X
Out[3]: array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

Variant 2:

```
In [1]: import numpy as np
In [2]: X = 1./10*np.arange(11)
```

...

Quiz: What will be printed?

```python
def f():
    x = 20

    def g():
        global x
        x = 40

    g()
    print(x)


f()
```

It will print `20`, as x refers to the enclosing namespace in `f()`, not the global namespace!

Questions?

Project week I passed …

- Last week, as part of version I (released this morning) you implemented a discrete dynamical simulation for either the Turing/Hopfield project (with some parameters)
- this week the task is to re-factor your code by creating an interface that we specify in the problem set
- release this v2 by on Monday after the fall break at 10 am (Oct 27).
- we will then review your code and give feedback (this is not graded)

Today’s summary

- deeper dive into functions: `` `def` ``, `` `return` ``
- scoping, namespaces, LEGB rule, global, globals()
- discussion of Python’s argument matching modes

Try out the commands in the Python shell/notebooks!

After lunch:

- Please note that we altered the room assignment for the exercises slightly. Check here and go to the correct room from now on. NOTE: you need your EPFL login to see it!
- Arrive early for the quiz (so you can start at 13:15)
- Monday 15:15 - 16: my office hours at SV 2811