Programming in Python[1]

Mattia Monga

Dip. di Informatica
Università degli Studi di Milano, Italia
mattia.monga@unimi.it

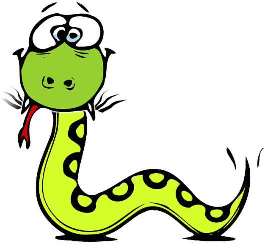
Academic year 2023/24, I semester

Lecture I: Programming in Python for quantitative biologists

Programming in Python (for quantitative biologists)

The course introduces imperative programming by referring to the Python language.

1. Python3 and its object-oriented features;
2. Python3 libraries that can be useful in scientific computation and data analysis, in particular `NumPy` and `pandas`.



Everything will be available on:
`https://mameli.docenti.di.unimi.it/pyqb`

Course schedule

- Tuesdays: 10:30 Aula Jommi, Thursday: 8:30 109, Fridays: 8:30 Lab Lambda Scheduling is complex: check the website
- Lectures: 40h, Labs: 16h
- Labs always on Friday
- We will explore different setups: (1) a "scaffolded" one for the first steps, (2) the plain python interpreter, and finally (3) the notebooks popular in scientific practice
- Tutor: TBD (computer science master student)
- Text: every Python3 reference/book/tutorial is ok, you can access freely to the book linked on the website
- Final test: write (small) python programs without help

Why Python?

Programming can be approached in many "languages", the fundamental skills are general. . . but you cannot learn without referring to a specific language.

- A precise requirement of the teaching committee
- Very popular in the scientific landscape
- Easy to learn, many useful libraries, free software
- Alternatives: Fortran, C, Matlab, Mathematica, R, Julia, . . .
- Python is slower, but it is considered easier to understand and manage

Which Python?

We will use Python3 (current version is 3.11): be careful when looking around, Python2 is still very common (but deprecated) and incompatible. Python supports different "paradigms", we will focus on:

- Imperative programming: programs describe changes in *registers* and the *executing environment*;
- Object-oriented: complex (imperative) programs are organized around objects in order to hide and isolate complexity.

This is a programming course: I will try to propose example that I believe could be useful in your daily practice, but I'm not a biologist.

Programming

Programming in science can serve two (almost opposite) goals:

1. Understanding every detail of a computational process;
2. Compose computational process by assembling powerful build blocks of which you understand very little.

Most of the current popularity of programming is related to goal 2. . . with many *sorcerer's apprentices*. But this course will focus mainly on goal 1. In the last part of the course we will bend towards 2, hopefully with a solid background.

Programming can be both hard and addictive: Teach Yourself Programming in Ten Years

Fundamental concepts of Python

The programmer describes computational processes in terms of:

objects : all the entities manipulated by the program, each has an identity (can be distinguished) and a value, that is an element in a specific type (a set of values together with the operations that make sense on them)

basic types : integers (`int`), floats, strings (`str`), functions; they can be composed in more complex types

variables : names used to refer to objects; the same name can refer to different objects during the same process

special commands : the only way to change the execution environment (i.e., the "virtual machine" provided by the operating system) is to use system calls; syscalls change from system to system (e.g., Linux vs. Windows), but Python wraps them and they appear like the functions written by the programmers (e.g., `print`), even if they could not be programmed in Python.

Let's try!

https://python.di.unimi.it/

You can use it without any personal account, but if you want support you must create one, putting me as the "guru": `mmonga`

This platform will be used for the first lessons, since it requires no setup at all: everything happens in the browser (and the server).
(Thanks to the University of Waterloo, Canada for providing the CS Circles)

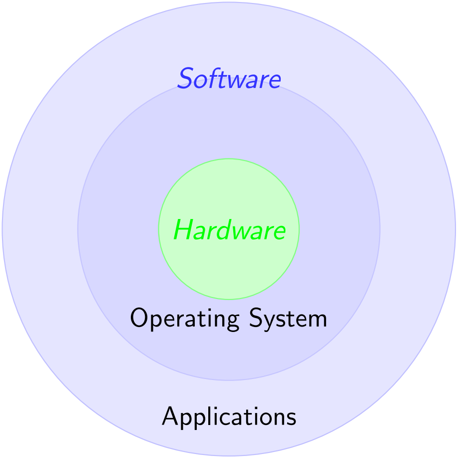
Lecture II: Fundamentals

Fundamental concepts of Python

The programmer describes computational processes in terms of:

- **objects**: all the entities manipulated by the program, each has an identity (can be distinguished) and a value, that is an element in a specific type (a set of values together with the operations that make sense on them)
- **basic types**: integers (`int`), floats, strings (`str`), functions; they can be composed in more complex types
- **variables**: names used to refer to objects; the same name can refer to different objects during the same process
- **special commands**: the only way to change the execution environment (i.e., the “virtual machine” provided by the operating system) is to use system calls; syscalls change from system to system (e.g., Linux vs. Windows), but Python wraps them and they appear like the functions written by the programmers (e.g., `print`), even if they could not be programmed in Python.

The onion model



- Operating System: it is the only program interpreted directly by the hardware; other pieces of software get interpreted by the virtual machine provided by it.
- Applications: programs (e.g., the python interpreter or python programs) executed within the protected environment created by the operating system.

What we want to do

- Programming means to instruct an (automatic) interpret with a precise description of a computational process.
- (In fact, the only way to make a description precise is to specify exactly the interpreter)
- We use a software interpreter, itself a program interpreted by the operating system (the stack of interpreters can be much deeper).
- Our interpret (Python3) manipulates objects taken from types (that define which manipulations are possible), referred by variables, with special commands to ask the services provided by the operating system.

Assignment

This is the fundamental statement for imperative programming:

- A name, known as variable, is needed to refer to objects.
  `professor = "Mattia"`
- = is not symmetrical, read it as becomes: Left-hand-side is always a variable, right-hand-side is an object, that can be either a literal or anything referred by another variable.
- A variable can change its value with another, following, assignment. Thus, the same variable may refer to different objects.
  `professor = "Violetta"`
- Basic objects (numbers, strings, Boolean values) are immutable (the variable change, not the object; different objects have always different identity)
- Tracking a program means to track the values of all the variables of a program during its execution.

Type hints

Since Python 3.4 it is possible (and indeed desirable, especially for novices) to hint any reader of a program about the type of a variable.

- A variable has always a type (a string in this case) `professor = 'Mattia'`
- Type hints make clear the intention of the programmer (can be checked by external programs) `professor: str = 'Mattia'`
- Assigning to an object of another type is still possible (there is no syntax error raised), but it should be regarded with suspicion `professor = True`

Basic operations

- Binary operators: `5 + 2`, they compute a new object by using the two objects on which they apply;
- Unary operators: `-(-5)`;
- Functions: `max`, they compute a new object by using an arbitrary number of objects (in general 0–. . . , `max` takes at least 1) passed as parameters (or arguments) when the function is called (`max(3, 6, something_else)`); sometimes the object computed is `None`;
- Syntactically appear as functions, but *commands* like `print("Hello!")` are actually used to request side effects in the executing environment.

Official Python docs (3.11)

Different approaches

Problem: exchange the name of two objects (Chapter 1, last exercise).

- Know the basic syntax of variables and assignment =
- Know the semantics of what you write: assigning an object to a variable delete any previous assignment;

Different approaches

Problem: exchange the name of two objects (Chapter 1, last exercise).

- Know the basic syntax of variables and assignment =
- Know the semantics of what you write: assigning an object to a variable delete any previous assignment;
- Natural strategy: use a temporary name to “save” the value during the exchange;

Different approaches

Problem: exchange the name of two objects (Chapter 1, last exercise).

- Know the basic syntax of variables and assignment =
- Know the semantics of what you write: assigning an object to a variable delete any previous assignment;
- Natural strategy: use a temporary name to "save" the value during the exchange;
- "Fox" strategy: know language or library tricks For example Python has a "multiple assignment" construct `x, y = y, x`, or a special library function `swap(x, y)` could exist;

Different approaches

Problem: exchange the name of two objects (Chapter 1, last exercise).

- Know the basic syntax of variables and assignment =
- Know the semantics of what you write: assigning an object to a variable delete any previous assignment;
- Natural strategy: use a temporary name to "save" the value during the exchange;
- "Fox" strategy: know language or library tricks For example Python has a "multiple assignment" construct `x, y = y, x`, or a special library function `swap(x, y)` could exist;
- "Hedgehog" strategy: study the problem in depth, e.g., if objects are numbers you can exploit arithmetic.

```
x = x + y
y = x - y
x = x - y
```

Basic types

`bool` `False`, `True` Logical operations

`int` `1`, `-33`, `1_000_000_000` ... Arithmetic operations, no upper or lower limit

`float` `1.0`, `.1`, `1.2e34` ... Arithmetic operations, limited but you have `float('infinity')` (and `float('nan')`)

```
sys.float_info(max=1.7976931348623157e+308,
↪   max_exp=1024, max_10_exp=308,
↪   min=2.2250738585072014e-308,
↪   min_exp=-1021, min_10_exp=-307, dig=15,
↪   mant_dig=53,
↪   epsilon=2.220446049250313e-16, radix=2,
↪   rounds=1)
```

`str` `'aaaa\nthis is on a new line'`, `"bbb'b\"b"` ... Concatenation, alphabetical ordering, replication, ...

Homework

Finish chapters 1, 1E, 2, 2X, 3, 4.
It shouldn't take more than a couple of hours, but exercising continuously is crucial.

Lecture III: Control flow

Basic types

- **bool** `False`, `True` Logical operations
- **int** `1`, `-33`, `1_000_000_000` … Arithmetic operations, no upper or lower limit
- **float** `1.0`, `.1`, `1.2e34` … Arithmetic operations, limited but you have `float('infinity')` (and `float('nan')`)

```
sys.float_info(max=1.7976931348623157e+308,
↪   max_exp=1024, max_10_exp=308,
↪   min=2.2250738585072014e-308,
↪   min_exp=-1021, min_10_exp=-307, dig=15,
↪   mant_dig=53,
↪   epsilon=2.220446049250313e-16, radix=2,
↪   rounds=1)
```

- **str** `'aaaa\nthis is on a new line'`, `"bbb'b\"b"` … Concatenation, alphabetical ordering, replication, …

Sequence of operations

```
x = 1 + 2 * 3
x = x + 1
```

The 2 lines of code translate to at least 5 “logical” instructions (maybe more, for example adding two big numbers require multiple instructions):

1. `2 * 3`
2. `1 + 6`
3. `x = 7`
4. `7 + 1`
5. `x = 8`

Flow of control

It is normally not very useful to write programs that do just one single computation. You wouldn't teach a kid how to multiply 32×43, but the general algorithm of multiplication (the level of generality can vary).

To write programs that address a family of problems we need to be able to select instructions to execute according to conditions.

```
if x < 0:
  x = -x

y = 2 * x
```

```
if x == -1:
  x = x + 1
else:
  x = 3 * x

y = 2 * x
```

In Python the indentation is part of the syntax and it is mandatory.

Input (special command needed)

- A special command to ask to the operating system (same as `print`)
- `input()` or `input("Prompt the user:")`
- The operating system (or the operating environment as in cscircle) collect the input data (from keyboard/console or the network in cscircles) and returns them to Python as a `str`.
  - `s = input() ## read a string`
  - `i = int(input()) ## read a string, convert to int`
- Input on cscircles seems strange, but when one understands the need of the mediation, the machinery is rather straighforward

Lecture IV: Repetitions

Repetitions

It is also useful to be able to repeat instructions: it is very convenient, but it also opens a deep Pandora's box...
There are two ways of looping in Python:

Repeat by iterating on the elements of a collection (similar to math notation $\sum_{i \in \{a,b,c\}} f(i)$)

```
for i in range(0, 5):
   # 0 1 2 3 4
   print(i)
```

Repeat while a (variable) condition is true

```
i = 0
while i < 5:
  print(i)
  i = i + 1
```

Euclid's GCD

Two unequal numbers being set out, and the less being continually subtracted in turn from the greater, if the number which is left never measures the one before it until an unit is left, the original numbers will be prime to one another. [. . .] But, if CD does not measure AB, then, the less of the numbers AB, CD being continually subtracted from the greater, some number will be left which will measure the one before it. [. . .]

[Euclid *"Elements"*, Book VII, Prop. I, II (c. 300 BC)]

```
a: int = 420
b: int = 240

while a != b:
  if a > b:
    a = a - b
  else:
    b = b - a

print(a)
```

Loops can be difficult to understand

When you have loops, understanding the code can be a difficult task and the only general strategy is to track the execution.

```
# This is known as Collatz's procedure
n = ...
while n > 1:
  if n % 2 == 0:
    # if the remainder of division by 2 is 0, i.e. n is
    ↪  even
    n = n / 2
  else:
    n = 3*n + 1
```

We know (by empirical evidence) that it ends for all $n < 2^{68} \approx 10^{20}$, nobody is able to predict the number of iterations given any `n`.
With loops it is also hard to exploit parallel execution.

Learn to write loops can be hard

When you write a loop, you should have in mind two related goals:

1. the loop must terminate: this is normally easy with `for` loops (when the finite collection ends, the loop ends also), but it can be tricky with `while`s (remember to change something in the condition);
2. the loop repeats something: the programmer should be able to write the "repeating thing" in a way that makes it equal in its form (but probably different in what it does).

The second part (technically known as loop invariant) is the hardest to learn, since it requires experience, creativity, and ingenuity.

Homework

- Create an account on https://github.com/ (if you don't have one) and send me the name.

Lecture V: Functions

Summary

In Python3

- Variables are names to refer to objects;
- Objects are elements of types, which define the operations that make sense on them;
- Therefore, the basic instructions are the assignment (bind a name to an object), the proper operations for each object, and the commands to ask the services of the operating system;
- One can alter the otherwise strictly sequential execution of instruction with control flow statements: `if`, `for`, `while`.

Remember that in python3, indentation matters (it is part of the syntax).

Proper operations

- On objects one can apply binary and unary operators: `2 * 3 -(-5.0) not True 'foo' + 'bar'`...
- There also built-in functions like `max(8,5,6)`, the full list is here: https://docs.python.org/3/library/functions.html
- (syntactically, commands like `print` or `input` cannot be distinguished from other built-in functions)
- Every object has `methods` that can be applied with the so called dot notation: `(3.2).is_integer()` `'foo'.upper()` `'xxx'.startswith('z')`; the list of which methods an object has is given by `dir(object)`.

Definition of functions

As variables are names for objects, one can also name fragments of code:

```
def cube(x: int) -> int:
    square = x * x
    return square * x
```

Now we have a new operation `cube`, acting on `int`s: `cube(3)`. Type hints are optional (and ignored, you can call `cube(3.2)` or `cube('foo')`), but very useful for humans (and tools like `mypy`).

```
# Equivalent
def cube(x):
    square = x * x
    return square * x
```

A function computes a result

- Returns a useful result

```
def concat_with_a_space(string1: str, string2: str) -> str:
   return string1 + ' ' + string2

# string1 is the _formal_ parameter
# 'foo' is the _actual_ parameter (like an assigment string1 =
↪  'foo')
print(concat_with_a_space('foo','bar'))
```

- Return None

```
def repeated_print(string: str, repetitions: int) -> None:
   for i in range(0, repetitions):
     print(string)

repeatedPrint('Hello, world!', 3)
```

- Recursive call:

```
def repeatedPrint(string: str, repetitions: int) -> None:
   if repetitions > 0:
     print(string)
     repeatedPrint(string, repetitions - 1)

repeatedPrint('Hello, world!', 3)
```

Functions are objects too

One can assign functions to variables:

```
def cube(x: int) -> int:
    square = x * x
    return square * x

mycube = cube

print(mycube(3))
print(type(mycube))
```

And short functions can even be expressed as literal expressions (lambda expressions)

```
cube = lambda y: y*y*y
```

Naming helps solving

The tower of Hanoi

https://www.mathsisfun.com/games/towerofhanoi.html

Describe the moves for a solution

Recursive thinking is a powerful problem solving technique and it can be translated to Python thanks to recursive calls.
Hanoi moves $A \to C$:

- In A there is just one disk: move it to C
- Otherwise in A there are n disks (> 1):
  - leap of faith! I suppose to know the moves needed to move $n - 1$ disk; then
    - apply this (supposed) solution to move $n - 1$ disks from A to B (leveraging on C, empty, as the third pole)
    - move the last disk from A to C
    - apply the (supposed) solution to move $n - 1$ disks from B to C (leveraging on A, now empty, as the third pole)

This implicit description solve the problem! Finding a non-recursive solution is possible but not that easy.

In Python

```
def hanoi(n: int, a_from: str, c_to: str,
↪  b_intermediate: str) -> None:
  if n == 1:
    print('Move 1 disk from ' + a_from + ' to ' + c_to)
  else:
    hanoi(n - 1, a_from, b_intermediate, c_to)
    print('Move 1 disk from ' + a_from + ' to ' + c_to)
    hanoi(n - 1, b_intermediate, c_to, a_from)

hanoi(3, 'A', 'C', 'B')
```

Lecture VI: Using the “naked” interpreter

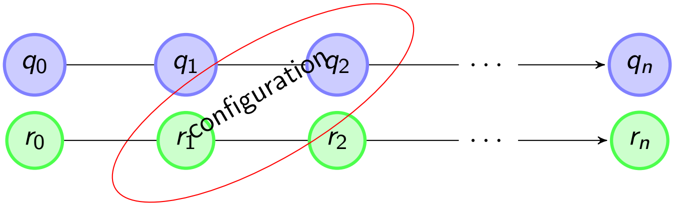
The pieces of software

- Python 3.10+, with `pip` and the `IDLE` editor (on MS Windows they are bundled together): https://www.python.org/downloads/
- Git 2.30+ https://git-scm.com/downloads
- (optional, Win and Mac only) Github desktop https://desktop.github.com/

Homework assigments will be available via Github Classroom (you will need a Github account).
When you push (hand in) your solution, a suite of tests is run.

Software Configuration Management

Software Configuration Management like `git` are tools designed to track all the revisions of some set of software artifacts (files).



The system configuration itself evolves in different versions. One can have multiple branches of evolution.

A motivating talk on why you should use tools like these in your scientific work.

Git

`git` is a powerful tool to manage all this complexity in a very efficient (and distributed) way. It is not an easy tool, however. A good tutorial is here. But for this course we use a very simplistic workflow:

1. Clone (copy) on your machine a repository `git clone ...`;
2. Work on the artifacts
3. Add the modified artifacts to the changeset you want to "publish" `git add ...`
4. Commit the changeset `git commit -m"message"` providing a comment about what have you done
5. Push the changeset on Github `git push`
6. (If someone else is working on the same artifacts you can sync with `git pull`)

All these steps are very easy (almost hidden, especially authentication) if you use Github desktop.

IDLE

Programs are data! File extension is conventionally `.py`

- To edit Python programs you need a text editor: something like `Notepad`, not `Word` (a word processor)
- `IDLE` is the "standard" one provided by the Python distribution itself: it is easy to use and it provides an easy way for executing programs without getting to the command line
- Other good choices: `VS Code` `Atom` `Notepad++` or any other universal text editor like `EMACS` or `vi`

Exercise

https://classroom.github.com/a/I3pCS400

Lecture VII: Composite objects

Simple and composite objects

- `int`s `float`s `bool`s are simple objects: they have no "parts"
- Strings are an example of composite objects since it is possible to consider also the characters: a `str` is a sequence of single characters; an important (simplifying) property: they are immutable
- Generic immutable sequences (with elements of any type) are called tuples (`tuple`): `(1, 2, 'foo')` `(1,)`
- Generic mutable sequences (with elements of any type) are called lists (`list`): `[1, 2, 'foo']` `[1]` `[1,2].append(3)`

Mutability

Immutable objects are simpler to use:

```
x = (1, 2, 3)
y = x

x = (10, 20, 30) # x refers to a new object, since the
↪ old cannot be changed
print(x, y)
```

Mutable ones require some caution:

```
x = [1, 2, 3]
y = x

x[0] = 10  # both x and y refer to a changed object
print(x, y)

x = [100, 200, 300]
print(x, y)
z = x.copy() # a copy not the same object
```

Exercises

- Write a function `middle(L: list[int])` which takes a list *L* as its argument, and returns the item in the middle position of *L*. (In order that the middle is well-defined, you should assume that *L* has odd length.) For example, calling `middle([8, 0, 100, 12, 1])` should return 100, since it is positioned exactly in the middle of the list. (`assert` is a useful tool to check assumptions — known as preconditions — are indeed true)
- Define a function `prod(L: list[int])` which returns the product of the elements in a list *L*.

Dictionaries

A composite type dict that implements a mapping between immutable keys and values.

```
d = {'key': 'foo', 3: 'bar'}

print(d['key']) # 'foo'
print(d[3])     # 'bar'
print(d[2])     # error!
```

Notation is similar to lists/tuples, but dicts are not sequences indexed by numbers, you must use only the existing keys (`d.keys()`).

```
if x in d.keys():
   print(d[x])
```

A sequence of values can be obtained with `d.values`. A sequence of 2-tuples `(key, value)` with `d.items()`.

Lecture VIII: Other Composite Objects

Sets

A `set` is a composite object with no duplicate (non mutable) elements. Common set operations are possible.

- Set literals: `{1,2,3}` `set()`
- `{1,2,3}.union({3,5,6})`
  `{1,2,3}.intersection({3,5,6})`

Comprehensions

Comprehensions are a concise way to create lists, sets, maps... It resembles the mathematical notation used for sets $A = \{a^2 | a \in \mathbb{N}\}$.

```
squares = [x**2 for x in range(10)]

# equivalent to:
squares = []
for x in range(10):
  squares.append(x**2)

# filtering is possible
odds = [x for x in range(100) if x % 2 != 0]

# with a set
s = {x for x in range(50+1) if x % 5 == 0}

# with a dict
d = {x: x**2 for x in range(10)}
```

Make a program readable

You never write a program only for a machine! You, others, tools will *read* the program for different purposes. Every minute spent in making a program more understandable pays off hours saved later.

- Type hinting makes clear what a function needs to work properly, and what it produces
- Documentation helps understanding without the need to read implementation details
- Examples of use make easy to remember how to use a function and can be used for verification

Example

```
from typing import Union

Num = Union[int, float]

def cube(x: Num) -> Num:
    """Return the cube of x.

    >>> cube(-3)
    -27

    >>> abs(cube(0.2) - 0.008) < 10e-5
    True
    """
    return x * x * x
```

Examples can be tested by:
`python -m doctest filename.py`.

Lecture IX: Files

Files

A file is an abstraction the operating system uses to preserve data among the execution of programs. Data must be accessed sequentially. (Italian reading people might enjoy this)

- We need commands to ask to the OS to give access to a file (`open`).
- It is easy to read or write data sequentially, otherwise you need special commands (`seek`) to move the file "cursor"
- The number of open files is limited ($\approx$ thousands), thus it is better to `close` files when they are not in use

Files contain bits (normally considered by group of bytes, 8 bits), the interpretation ("format") is given by the programs which manipulate them. However, "lines of printable characters" (plain text) is a rather universal/predefined interpretation, normally the easiest to program.

File read access

```
f = open('filename.txt', 'r') # read only

# iterating on a file reads (all) the lines
for i in f:
    print(i)

# End of file already reached, result is ''
f.readline()

f.close()

# File closed, error!
f.readline()
```

To avoid remembering to close explicitly, Python provides the context manager syntax.

```
with open('filename.txt', 'r') as f:
  for i in f:
    print(i)
```

Lecture X: Encapsulation

Procedural abstraction

Procedural abstraction is key for our thinking process (remember the power of recursion, for example): giving a name to a procedure/function enhances our problem solving skills.

```
def sum_range(a: int, b: int) -> int:
    """Sum integers from a through b.

    >>> sum_range(1, 4)
    10

    >>> sum_range(3, 3)
    3
    """
    assert b >= a
    result = 0
    for i in range(a, b+1):
        result = result + i
    return result
```

Another "sum"

This is very similar...

```
def sum_range_cubes(a: int, b: int) -> int:
    """Sum the cubes of the integers from a through b.

    >>> sum_range_cubes(1, 3)
    36

    >>> sum_range_cubes(-2, 2)
    0

    """
    assert b >= a
    result = 0
    for i in range(a, b+1):
        result = result + cube(i) # cube(i: int) -> int
         ↪  defined elsewhere
    return result
```

Another "sum"

This is also very similar...

$$\frac{1}{a\cdot(a+2)} + \frac{1}{(a+4)\cdot(a+6)} + \frac{1}{(a+8)\cdot(a+10)} + \cdots + \frac{1}{(b-2)\cdot(b)}$$

(Leibniz: $\frac{1}{1\cdot 3} + \frac{1}{5\cdot 7} + \frac{1}{9\cdot 11} + \cdots = \frac{\pi}{8}$)

Another "sum"

This is also very similar...

$$\frac{1}{a\cdot(a+2)} + \frac{1}{(a+4)\cdot(a+6)} + \frac{1}{(a+8)\cdot(a+10)} + \cdots + \frac{1}{(b-2)\cdot(b)}$$

(Leibniz: $\frac{1}{1\cdot 3} + \frac{1}{5\cdot 7} + \frac{1}{9\cdot 11} + \cdots = \frac{\pi}{8}$)

```
def pi_sum(a: int, b: int) -> float:
    """Sum 1/(a(a+2)) terms until (a+2) > b.

    >>> from math import pi
    >>> abs(8*pi_sum(1, 1001) - pi) < 10e-3
    True

    """
    assert b >= a
    result = 0.0
    for i in range(a, b+1, 4):
        result = result + (1 / (i * (i + 2)))
    return result
```

Can we abstract the similarity?

```python
from typing import Callable

Num = int | float # same as Num = Union[int, float]

def gen_sum(a: int, b: int, fun: Callable[[int], Num], step: int = 1) -> Num:
    """Sum terms from a through b, incrementing by step.

    >>> gen_sum(1, 4, lambda x: x)
    10

    >>> gen_sum(1, 3, lambda x: x**3)
    36

    >>> from math import pi
    >>> abs(8*gen_sum(1, 1000, lambda x: 1 / (x * (x + 2)), 4) - pi) < 10e-3
    True

    """

    assert b >= a
    result = 0.0
    for i in range(a, b+1, step):
        result = result + fun(i)
    if isinstance(result, float) and result.is_integer():
        return int(result)
    return result
```

The huge value of procedural abstraction

It is worth to emphasize again the huge value brought by procedural abstraction. In Python it is not mandatory to use procedures/functions: the language is designed to be used also for *on the fly* calculations.

```
x = 45
s = 0
for i in range(0, x):
  s = s + i
```

This is ok, but it is not encapsulated (in fact, since encapsulation is so important you can at least consider it encapsulated in file which contains it)

- the piece of functionality is not easily to distinguish

  it could be intertwined with other unrelated code

```
x = 45
a = 67 # another concern
s = 0
for i in range(0, x):
  s = s + i
print(a) # another concern
```

- the goal is not explicit, which data are needed, what computes
- it's hard to reuse even in slightly different contexts

Encapsulate the functionality

```
def sum_to(x: int) -> int:
    assert x >= 0
    r = 0
    for i in range(0, x):
        r = r + i
    return r

s = sum_to(45)
```

- It gives to our mind a "piece of functionality", the interpreter we are programming is now "able" to do a new thing that can be used without thinking about the internal details
- It makes clear which data it needs (an integer, ≥ 0 if we add also an assertion or a docstring)
- It makes clear that the interesting result is another integer produced by the calculation
- It can be reused easily and safely

Lecture XI: OOP

Object Oriented encapsulation

Encapsulation is so important that it is used also at a higher level: a collection of related procedures.

```
x = 666

def increment():
  x = x + 1

def decrement():
  x = x - 1
```

Again: this is correct Python code, but it has problems:

- Both the functions depends on `x` but this is not clear from their signature: a user must look at the internal details
- The two functions cannot be reused individually, but only together with the other (and `x`)

Classes

A class is a way to package together a collection of related functions. The class is a “mold” to instance new objects that encapsulated the related functionalities.

```
class Counter:
   def __init__(self, start: int):
     self.x = start

   def increment(self):
     self.x = self.x + 1

   def decrement(self):
     self.x = self.x - 1

c = Counter(666)
c.decrement()
d = Counter(999)
d.increment()
```

Lecture XII: Lab 4

Status

23 rostered students:

	subscribed	done
One triangle	17	
Triangle kinds	16	4
DNA Hamming	21	10
Newton square root	15	6
Pythagorean triplets	8	4
DNA files	15	2
flatten list	3	1

The figures didn't change that much since last lab!

Exercises

- Eels https://classroom.github.com/a/p3UKOtXC
- Flatten list (not new) https://classroom.github.com/a/L8_e5QiN
- DNA forensics https://classroom.github.com/a/j5nL7_Ef

Lecture XIII: Random numbers

Random numbers

Pseudorandomness: the sequence of numbers is not predictable. . .

```
from random import randint

# To get a random integer x in the set [1..10]
x = randint(1, 10)
```

```
from random import randint

for _ in range(0,10):
   print(randint(1, 100))
```

unless you know the seed.

```
from random import seed, randint

seed(292)
for _ in range(0,10):
   print(randint(1, 100))
```

Exercise

Write a Python program which chooses an integer 1–10 and asks to the user to guess it

- if the number given by the user is not 1–10, it prints "Invalid";
- if the number is the chosen one, it prints "Yes!";
- otherwise "You didn't guess it…".

Exercise

Write a Python program which chooses an integer 1–10 and asks to the user to guess it

- if the number given by the user is not 1–10, it prints "Invalid";
- if the number is the chosen one, it prints "Yes!";
- otherwise "You didn't guess it...".

Evolve the program: it should now ask until the user guess the number correctly, giving hints ("higher...", "lower...").

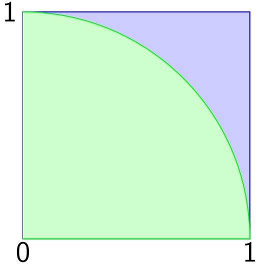
Exercise

Write a Python program which chooses an integer 1–10 and asks to the user to guess it

- if the number given by the user is not 1–10, it prints "Invalid";
- if the number is the chosen one, it prints "Yes!";
- otherwise "You didn't guess it...".

Evolve the program: it should now ask until the user guess the number correctly, giving hints ("higher...", "lower...").
How many tries in the worst case? Can you write a program guessing a number between 1 and `int(1e32)`

Example



- Blue square: 1
- Green area: $\frac{\pi}{4}$

The Monte Carlo method consists of choosing sample experiments at random from a large set and then making deductions on the basis of the probabilities estimated from frequency of occurrences.

Lecture XIV: Random numbers

Example

```
from random import random

def approx_pi(tries: int) -> float:
    """Return an approximation for pi.

    >>> from math import pi
    >>> from random import seed
    >>> seed(7897)  # Tests should be reproducible
    >>> abs(4*approx_pi(1000) - pi) < 10e-2
    True

    >>> abs(4*approx_pi(100000) - pi) < abs(approx_pi(1000) - pi)
    True
    """
    assert tries > 0
    within_circle = 0
    for i in range (0, tries):
        x = random() # range [0,1)
        y = random()
        if x**2 + y**2 < 1:
            within_circle += 1
    return within_circle / tries
```

Example

It's easy to extend to make this work for any function on $[0, 1)$.

```
from random import random
from typing import Callable

def approx_fun(predicate: Callable[[float, float], bool], tries:
↪   int) -> float:
    """Return an approximation for pi.

    >>> from math import pi
    >>> from random import seed
    >>> seed(7897)  # Tests should be reproducible
    >>> within_circle = lambda x, y: x**2 + y**2 < 1
    >>> abs(4*approx_fun(within_circle, 1000) - pi) < 10e-2
    True
    """
    assert tries > 0
    true_cases = 0
    for i in range (0, tries):
        x = random() # range [0,1)
        y = random()
        if predicate(x, y):
            true_cases += 1
    return true_cases / tries
```

Simulations

Random number are useful also for *simulation*: for example, we could simulate evolutionary drift.

```
from random import seed, randint, getstate, setstate

class DriftSimulation:
    def __init__(self, sim_seed: int = 232943) -> None:
        self.population = ['\N{MONKEY}', '\N{TIGER}', '\N{BUTTERFLY}', '\N{LIZARD}',
        ↪  '\N{SNAIL}']
        seed(sim_seed)
        self.r_state = getstate()

    def offspring(self) -> None:
        setstate(self.r_state)
        new = self.population[randint(0, len(self.population)-1)]
        self.population[randint(0, len(self.population)-1)] = new
        self.r_state = getstate()

    def simulate(self, generations: int) -> None:
        for i in range(0, generations):
            self.offspring()

a = DriftSimulation()
b = DriftSimulation()
a.simulate(2)
b.simulate(2)
```

Lecture XV: Using Third-party libraries

Third-party libraries

Python is “sold” *batteries included* (with many useful built-in libraries). Moreover, like many modern programming environments, it has standard online package directories that list libraries produced by independent developers.
`https://pypi.org/`
The Python package index currently lists almost 300K libraries!

Installing a library

The details are explained here: https://packaging.python.org/tutorials/installing-packages/

- In most cases it is very easy, the `pip` program does all the magic
- It is very important to understand the difference between a system-wide and a project-specific installation.

System-wide vs. Project-specific

If you don't take special precautions, a package is installed in a way that makes it available to your Python system: every Python interpreter you launch sees them.

- In many cases, this is not what you want
- Different projects/programs might depend on different versions of the libraries
- Libraries themselves depend on other libraries, you want to understand exactly which packages your program is using in order to reproduce the settings on other machines

Virtual environments

Python provides the idea of virtual development environments (`venv`)

- You can create one with: `python -m venv CHOOSE_A_NAME`
- You must activate it (syntax depends on your OS): `CHOOSE_A_NAME\Scripts\activate.bat`
- In an active virtual environment all the installation are confined to it
- You can get the list of installed packages with `pip freeze`

Simplified venv administration

Virtual environments are key to avoid messing up your system.
Many tools simplify their administration.

- `pipenv` (my preferred one, we will use this)
- `poetry` (similar to `pipenv`, currently less popular, but it has a better dependency control, a bit more complex)
- `conda` (uses its own package index, great flexibility and complexity, manage different python versions)

Virtual environments caveats

When you are working in a Python virtual environment, remember to launch all your development tools "inside" the virtual space.
For example, to use Thonny you have to activate the proper virtual environment each time you launch the application.

Lecture XVI: NumPy arrays

NumPy

NumPy is a third-party library very popular for scientific/numerical programming (https://numpy.org/).

- Features familiar to matlab, R, Julia programmers
- The key data structure is the array
  - 1-dimension arrays: vectors
  - 2-dimension arrays: matrices
  - n-dimension arrays

In some languages array is more or less synonym of list: Python distinguishes: lists (mutable, arbitrary elements), arrays (mutable, all elements have the same type), tuples (immutable, fixed length, arbitrary elements).

NumPy arrays

The most important data structure in `NumPy` is `ndarray`: a (usually fixed-size) sequence of same type elements, organized in one or more dimensions.
https://numpy.org/doc/stable/reference/arrays.ndarray.html
Implementation is based on byte arrays: accessing an element (all of the same byte-size) is virtually just the computation of an 'address'.

Why?

- using NumPy arrays is often more compact, especially when there's more than one dimension
- faster than lists when the operation can be vectorized
- (slower than lists when you append elements to the end)
- can be used with element of different types but this is less efficient

A `ndarray` has a `dtype` (the type of elements) and a `shape` (the length of the array on each dimensional axis). (Note the jargon: slightly different from linear algebra)

- Since appending is costly, normally they are pre-allocated (`zeros`, `ones`, `arange`, `linspace`, . . .)
- vectorized operations can simplify code (no need for loops) and they are faster with big arrays
- vector indexing syntax (similar to R): very convenient (but you need to learn something new)

All the elements must have the same size

This is actually a big limitation: the faster access comes with a price in flexibility.

```
>>> np.array(['','',''])
array(['', '', ''], dtype='<U1')
>>> np.array(['a','bb','ccc'])
array(['a', 'bb', 'ccc'], dtype='<U3')
>>> np.array(['a','bb','cccxxxxxxxxxxxxxxxxxx'])
array(['a', 'bb', 'cccxxxxxxxxxxxxxxxxxx'], dtype='<U21')
```

Usually the length is not changed

The best use of arrays is to avoid a change in their length, that can be costly. Thus, they are normally preallocated at creation:

- `np.array([1,2,3])`
- `np.zeros(2)`, `np.zeros(2, float)`, `np.ones(2)`
- `np.empty((2,3))` six not meaningful float values
- `np.arange(1, 5)` be careful with floats:

```
>>> np.arange(0.4, 0.8, 0.1)
array([0.4, 0.5, 0.6, 0.7])
>>> np.arange(0.5, 0.8, 0.1)
array([0.5, 0.6, 0.7, 0.8])
```

- `np.linspace(0.5, 0.8, 3)` with this the length is easier to predict

You can concatenate arrays with `np.concatenate` (be careful with the shapes!)

Lecture XVII: NumPy arrays

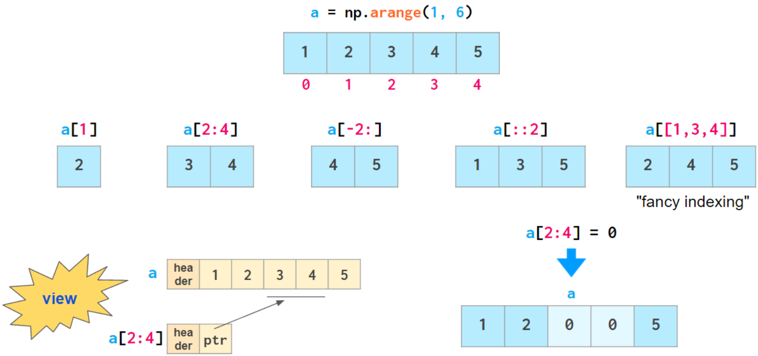
Don't remove, select

In general you don't remove elements but select them. Be careful: if you don't make an explicit copy you get a "view" and possibly side-effects.

```
>>> a = np.ones((2,3))
>>> a
array([[1., 1., 1.],
       [1., 1., 1.]])
>>> x = a[:, 1]
>>> x
array([1., 1.])
>>> x[0] = 0
>>> x
array([0., 1.])
>>> a
array([[1., 0., 1.],
       [1., 1., 1.]])
```

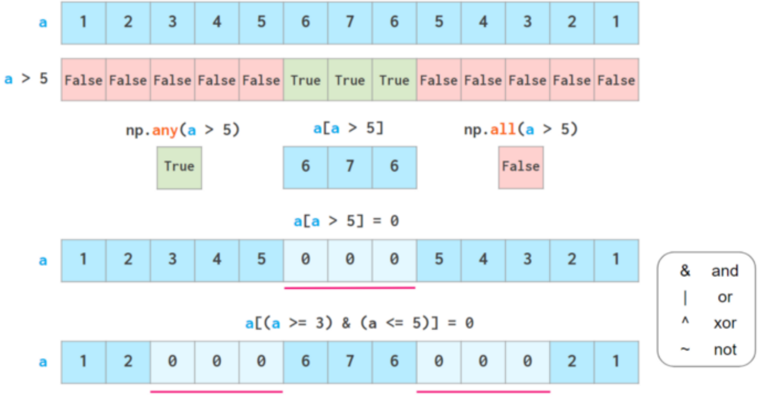
```
>>> x = a[:, 1].copy()
>>> x[1] = 100
>>> x
array([  0., 100.])
>>> a
array([[1., 0., 1.],
       [1., 1., 1.]])
```

Indexing is powerful



Picture from “NumPy Illustrated: The Visual Guide to NumPy”, highly recommended

Indexing is powerful



Picture from "NumPy Illustrated: The Visual Guide to NumPy", highly recommended

Warning! Assignment works differently from lists

```
>>> np = np.array([1,2,3,4,5])
>>> lst = [1,2,3,4,5]
>>> np[2:4] = 0
>>> np
array([1, 2, 0, 0, 5])
>>> lst[2:4] = 0 # Error!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only assign an iterable
>>> lst[2:4] = [0,0]
>>> lst
[1, 2, 0, 0, 5]
>>> lst[2:4] = [0,0,0]
>>> lst
[1, 2, 0, 0, 0, 5]
>>> np[2:4] = [0,0]
>>> np[2:4] = [0,0,0] # Error!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not broadcast input array from shape (3,) into
↪ shape (2,)
```

The highest power: vectorization

Most of the basic mathematical function are vectorized: no need for loops! This is both convenient and faster!

```
>>> a = np.array([1,2,3,4])
>>> a + 1
array([2, 3, 4, 5])
>>> a ** 2
array([ 1,  4,  9, 16])
>>> np.exp(a)
array([ 2.71828183,  7.3890561 , 20.08553692,
↪  54.59815003])
```

Array operations

On arrays you have many "aggregate" operations.

```
>>> a
array([1, 2, 3, 4])
>>> a.sum()
10
>>> a.max()
4
>>> a.argmin()
0
>>> a.mean()
2.5
```

Remember to look at `dir` or the online documentation.

Lecture XVIII: Matplotlib

Matplotlib

When you have arrays with many data it is useful to have a way to display them graphically.

- The most popular is `matplotlib` https://matplotlib.org/
- Many other graphical frameworks (e.g., `seaborn`) based on it
- Many, many possibilities to tune your graphics! It's hard to master every detail.
- Be careful: it can be used with two different styles.
  1. The (preferred) object-oriented way: clean and rational, but a bit more verbose
  2. The procedural way: mostly useful only for "throw-away" scripts, but for this reason more common in the examples you can find online

Graphical output is an operating system service

- Output is a service provided by the operating system: *textual* output is very standardized even across different platform, graphics is not so stable
- When you deal with graphical programs: expect installation headaches, portability glitches, etc.

The OO style

- You need always to objects: a `Figure` and a `Axes`
- plotting happens on axes, framed in a figure
- very flexible: you can add plots on the same axis, or you can have many axes collected in a single figure

Basic example

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

x = np.linspace(-2*np.pi, 2*np.pi, 100)

fig, ax = plt.subplots()

ax.plot(x, np.sin(x))

fig.show()
```

Many different types of charts

If `ax` is a `Axes`

- Scatter-plots `ax.scatter`
- Bar-plots `ax.bar`
- Histograms `ax.hist`
- 2D `ax.imshow`

Tweaks

- add labels, legends, titles
- add a grid
- combine multiple plots on the same axis
- combine multiple axes on the same figure

Save your pictures!

A `Figure` can be saved in a file with `savefig`. You should keep in mind the difference between:

- bitmap formats (`png` `jpg` ...): the file is matrix of pixels
- vector formats (`svg` `pdf` ...): the file is a set of instructions to reproduce the picture, less portable but it can be magnified

Lecture XIX: A game of life

Using the notebook in a virtual environment

Since we are now interested in graphics, Jupyter notebooks can be very convenient to see pictures together with the code.

1. We set up a virtual environment as usual
2. With `pip install notebook` we have the Jupyter notebook machinery available
3. I normally want to have also a clean `.py` file, since `.ipynb` do not play well with configuration management (git) and other command line tools like the type checker or doctest: thus I suggest to install `jupytext`; it needs a `jupytext.toml` text file telling `.ipynb` and `.py` files are paired, *i.e.*, they are kept synchronized.

   ```
   # Always pair ipynb notebooks to py files
   formats = "ipynb,py:percent"
   ```
4. lunch the notebook with `jupyter notebook`

A game of life

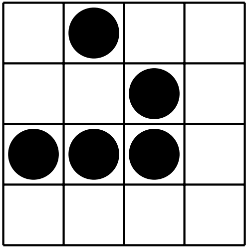
In 1970, J.H. Conway proposed his Game of Life, a simulation on a 2D grid:

1. Every cell can be *alive* or *dead*: the game start with a population of alive cells (*seed*)
2. any alive cell with less of 2 alive neighbours dies (*underpopulation*)
3. any alive cell with more than 3 alive neighbours dies (*overpopulation*)
4. any dead cell with exactly 3 alive neighbours becomes alive (*reproduction*)

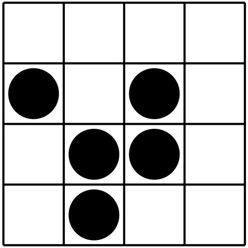
The game is surprisingly rich: many mathematicians, computer scientists, biologists. . . spent their careers on the emerging patterns!

Life forms

There are names for many "life forms": *still lifes*, *oscillators*, *starships*. . .
A famous starship is the glider:



1	1	2	1
3	5	3	2
1	3	2	2
2	3	2	1



The glider repeats itself in another position after 4 generations.

Python implementation

To implement a Game of Life simulation in Python, we can:

- use a `ndarray` for the grid
- each cell contains 0 (dead) or 1 (alive)
- for simplicity we can add a "border" of zeros

0	0	0	0	0
0	1	1	1	0
0	1	0	1	0
0	1	1	0	0
0	0	0	0	0

Avoiding loops

For a 1-D array `X`

0	**1**	**1**	**0**	**1**	0

All the neighbours on the right `X[2:]`

0	**1**	**1**	**0**	**1**	0

All the neighbours on the left `X[:-2]`

What does `X[2:] + X[:-2]` represent? The sum is (yellow) element by (yellow) element, the result is: `[1,1,2,0]`
Can you think to a similar solution for the 2-D case?

Avoiding loops

0	0	0	0	0	0
0	**0**	**1**	**0**	**0**	0
0	**0**	**0**	**1**	**0**	0
0	**1**	**1**	**1**	**0**	0
0	**0**	**0**	**0**	**0**	0
0	0	0	0	0	0

```
X[1:-1, 2:]
```

Avoiding loops

0	0	0	0	0	0
0	**0**	**1**	**0**	**0**	0
0	**0**	**0**	**1**	**0**	0
0	**1**	**1**	**1**	**0**	0
0	**0**	**0**	**0**	**0**	0
0	0	0	0	0	0

`X[2:,2:]`

Avoiding loops

0	0	0	0	0	0
0	**0**	**1**	**0**	**0**	0
0	**0**	**0**	**1**	**0**	0
0	**1**	**1**	**1**	**0**	0
0	**0**	**0**	**0**	**0**	0
0	0	0	0	0	0

```
X[2:,1:-1]
```

Avoiding loops

0	0	0	0	0	0
0	**0**	**1**	**0**	**0**	0
0	**0**	**0**	**1**	**0**	0
0	**1**	**1**	**1**	**0**	0
0	**0**	**0**	**0**	**0**	0
0	0	0	0	0	0

`X[2:,1:-1]`

And other 5 matrices. . .

Avoiding loops

X

0	0	0	0	0	0
0	**0**	**1**	**0**	**0**	0
0	**0**	**0**	**1**	**0**	0
0	**1**	**1**	**1**	**0**	0
0	**0**	**0**	**0**	**0**	0
0	0	0	0	0	0

N

0	0	0	0	0	0
0	1	1	2	1	0
0	3	5	3	2	0
0	1	3	2	2	0
0	2	3	2	1	0
0	0	0	0	0	0

`X == 1` `N > 3`

Death by overpopulation: `X[(X == 1) & (N > 3)] = 0` (empty in this case!)

Lecture XX: Laplacian operator

Gray-Scott systems

Systems driven by the Gray-Scott's equation exhibit Turing patterns (D_u, D_v, f, k are constants).

$$\frac{\partial u}{\partial t} = D_u \nabla^2 u - uv^2 + f \cdot (1 - u)$$
$$\frac{\partial v}{\partial t} = D_v \nabla^2 v + uv^2 - (f + k) \cdot v$$

- These give the change of u and v chemicals over time
- The diffusion term can be approximated on a grid by computing the discrete Laplacian

Discrete Laplacian

$$\nabla^2 = \nabla \cdot \nabla = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$$

- Change on a grid (1-D):

$$\nabla f[n] = f[n+1] - f[n]$$
$$\nabla f[n] = f[n] - f[n-1]$$

- Second order change (1-D):

$$\begin{aligned}\nabla(\nabla f[n]) &= \nabla(f[n+1]) - \nabla(f[n]) \\ &= (f[n+1] - f[n]) - (f[n] - f[n-1]) \\ &= f[n-1] - 2f[n] + f[n+1]\end{aligned}$$

- In 2-D we do this independently on the 2 dimensions n, m:

$$\begin{aligned}\nabla(\nabla f[n,m]) &= f[n-1,m] - 2f[n,m] + f[n+1,m] + \\ &\quad f[n,m-1] - 2f[n,m] + f[n,m+1] \\ &= f[n-1,m] + f[n+1,m] + f[n,m-1] + f[n,m+1] - 4f[n,m]\end{aligned}$$

Vectorization

0	0	0	0	0	0
0	13	14	15	16	0
0	9	10	11	12	0
0	5	6	7	8	0
0	1	2	3	4	0
0	0	0	0	0	0

-29	-18	-19	-37
-8	0	0	-13
-4	0	0	9
3	2	1	-5

Same trick we used for "life", but we need to compute the *5-point stencil* with these weights (see previous derivation):

0	1	0
1	-4	1
0	1	0

This way one can compute the Laplacian matrix using only vectorized plus.

Vectorization

0	0	0	0	0	0
0	13	14	15	16	0
0	9	10	11	12	0
0	5	6	7	8	0
0	1	2	3	4	0
0	0	0	0	0	0

`X[1:-1, 2:]`

-29	-18	-19	-37
-8	0	0	-13
-4	0	0	9
3	2	1	-5

Same trick we used for "life", but we need to compute the *5-point stencil* with these weights (see previous derivation):

0	1	0
1	-4	1
0	1	0

This way one can compute the Laplacian matrix using only vectorized plus.

Vectorization

0	0	0	0	0	0
0	13	14	15	16	0
0	9	10	11	12	0
0	5	6	7	8	0
0	1	2	3	4	0
0	0	0	0	0	0

`X[2:, 1:-1]`

-29	-18	-19	-37
-8	0	0	-13
-4	0	0	9
3	2	1	-5

Same trick we used for "life", but we need to compute the *5-point stencil* with these weights (see previous derivation):

0	1	0
1	-4	1
0	1	0

This way one can compute the Laplacian matrix using only vectorized plus.

Vectorization

0	0	0	0	0	0
0	13	14	15	16	0
0	9	10	11	12	0
0	5	6	7	8	0
0	1	2	3	4	0
0	0	0	0	0	0

`X[1:-1, :-2]`

-29	-18	-19	-37
-8	0	0	-13
-4	0	0	9
3	2	1	-5

Same trick we used for "life", but we need to compute the *5-point stencil* with these weights (see previous derivation):

0	1	0
1	-4	1
0	1	0

This way one can compute the Laplacian matrix using only vectorized plus.

Vectorization

0	0	0	0	0	0
0	13	14	15	16	0
0	9	10	11	12	0
0	5	6	7	8	0
0	1	2	3	4	0
0	0	0	0	0	0

-29	-18	-19	-37
-8	0	0	-13
-4	0	0	9
3	2	1	-5

`X[:-2, 1:-1]`

Same trick we used for "life", but we need to compute the *5-point stencil* with these weights (see previous derivation):

0	1	0
1	-4	1
0	1	0

This way one can compute the Laplacian matrix using only vectorized plus.

0	0	0	0	0	0
0	13	14	15	16	0
0	9	10	11	12	0
0	5	6	7	8	0
0	1	2	3	4	0
0	0	0	0	0	0

-29	-18	-19	-37
-8	0	0	-13
-4	0	0	9
3	2	1	-5

`X[1:-1, 1:-1]`

Same trick we used for "life", but we need to compute the *5-point stencil* with these weights (see previous derivation):

0	1	0
1	-4	1
0	1	0

This way one can compute the Laplacian matrix using only vectorized plus.

Consider also the diagonals

Another approximation which takes into account also the "diagonals" is the *9-point stencil*.

1	1	1
1	-8	1
1	1	1

Experimental evidence

Turing proposed his model on a pure theoretical basis, but we have now also some experimental evidence:

> *Economou, A. D., Ohazama, A., Porntaveetus, T., Sharpe, P. T., Kondo, S., Basson, M. A., Gritli-Linde, A., Cobourne, M. T., Green, J. B. (2012). Periodic stripe formation by a Turing mechanism operating at growth zones in the mammalian palate. Nature genetics, 44(3), 348–351.* https://doi.org/10.1038/ng.1090

Lecture XXI: Tabular data

Tabular data

Data are often given/collected as tables: matrices with rows for individual records and columns for the fields of the records. This is especially common in statistics, R has a built-in type for this: the dataframe.

pandas

`pandas` (Python for data analysis) brings the `DataFrame` type to Python. It is based on `numpy`.

- `Series`: a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the index.
- `DataFrame`: a 2-dimensional labeled data structure with columns of potentially different types. You can think of it like a spreadsheet, or a `dict` of `Series` objects.

Series

```
import pandas as pd
s = pd.Series(np.random.randn(5), index=["a", "b", "c",
↪  "d", "e"])
```

s is a numpy array of floats, each one has a label.

```
d = {"b": 1, "a": 0, "c": 2}

s = pd.Series(d)
```

The ordering depends on Python and pandas version. . . The current ones takes the insertion order, but you can provide explicitly the index.

```
d = {"b": 1, "a": 0, "c": 2}

s = pd.Series(d, index=['a', 'b', 'c'])
```

Series

A `Series` is convenient because it is a `ndarray` (and can be vectorized) but also a `dict`.

Dataframes

```
d = { "one": pd.Series([1.0, 2.0, 3.0], index=["a",
↪  "b", "c"]),
      "two": pd.Series([1.0, 2.0, 3.0, 4.0],
       ↪  index=["a", "b", "c", "d"]),
    }

df = pd.DataFrame(d)
```

A `DataFrame` has an `index` and a `columns` attribute.
There are many ways of creating `DataFrames`, see the docs.

From csv or spreadsheets

A famous example: Fisher's Iris flowers dataset.
150 records, "sepal length","sepal width","petal length","petal width","class"

```
iris = pd.read_csv('iris.csv')
# with a url
iris = pd.read_csv('https://tinyurl.com/iris-data')
```

Two ways of indexing

- `.loc[]` "label based"
- `.iloc[]` "position based"

For both you can use: a single value, a list of values, a boolean array. Two notable things:

1. If you use a slice notation with `.loc` (`'a':'f'`) the last value is included! (different from plain python and from `.iloc`)
2. Can be also a callable function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above)

Lecture XXII: More pandas

Group by

Data can be grouped with `groupby`, then *summary* function (`sum`, `mean`, ...) can be applied to each group at the same time.

```
iris = pd.read_csv('https://tinyurl.com/iris-data')
```

```
iris.groupby('variety').mean()
```

Groups are special lazy types which generate data only when needed for the summary operation.

Iterators

Object can be iterable. Python defines the iterator protocol as:

- `iterator.__iter__()` Return the iterator object itself. This is required to allow both containers and iterators to be used with the for and in statements.
- `iterator.__next__()` Return the next item from the container. If there are no further items, raise the StopIteration exception.

Notable iterators

Built-in lists, tuples, ranges, sets, dicts are iterators.

- Numpy arrays
- Pandas Series and DataFrames

Generators

```
def mygenerator() -> int:
    for i in [1, 6, 70, 2]:
        yield i
    print('Ended')  # Just to see when it reaches this
    ↪  point

g = mygenerator()

print(g)  # not useful
print(next(g))
print(next(g))
print(next(g))
print(next(g))
print(next(g))     # Exception
```

Pandas DataFrame

Be careful: the default iteration is on column names (similar to dicts, which iterate on keys).

- `iterrows()`: Iterate over the rows of a DataFrame as `(index, Series)` pairs. This converts the rows to Series objects, which can change the dtypes and has some performance implications.
- `itertuples()`: Iterate over the rows of a DataFrame as namedtuples of the values. This is a lot faster than `iterrows()`, and is in most cases preferable to use to iterate over the values of a DataFrame.

Iterating is slow: whenever possibile try to use vectorized operation or function application.

Pandas function application

```
# apply the function to each column
df.apply(lambda col: col.mean() + 3)

# apply the function to each row
df.apply(lambda row: row + 3, axis=1)
```

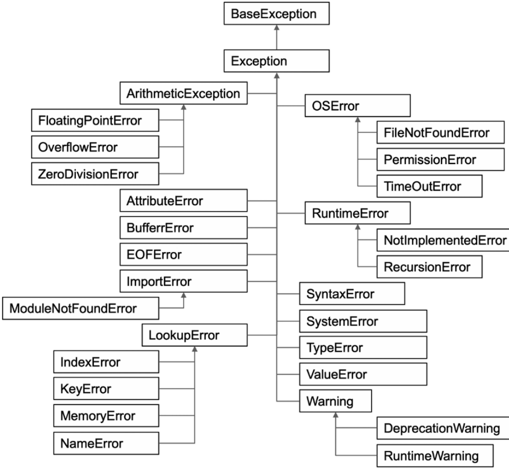
Pandas query


```
df[df['A A'] > 3]

# equivalent to this (backticks because of the space)
df.query('`A A` > 3')

# query can also refer to the index
df.query('index >= 15')

# same as
df[15:]
```

Lecture XXIII: Exception handling

Exceptions

- Exceptions and Errors are object raised (or thrown) in the middle of an anomalous computation.
- Exceptions change the control flow: the control passes to the "closer" handler, if it exists: otherwise it aborts.



Exception handling

Exceptions can be handled: the strategy is normally an "organized panic" in which the programmer tidies up the environment and exits.

```
danger()
# An exception in danger
# aborts the program
```

```
try:
  danger()
except:
  # An exception in danger
  # it's handled here
```

```
try:
  danger()
except OverflowError as e:
  # An exception in danger
  # it's handled here
  # The object is referred by
  ↪  e
finally:
  # This is executed in any
  ↪  case
```

Raising an exception

To explicitly raise an exception, use the `raise` statement

```
if something == WRONG:
    raise ValueError(f'The value {something} is wrong!')
```

Assertions are a disciplined way to raise exceptions.

Lecture XXIV: Inheritance

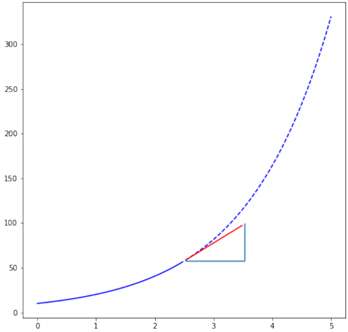
Destructuring a bound computation

```
def approx_euler(t: np.ndarray, f0: float, dfun:
↪  Callable[[float], float]) -> np.ndarray:
    """Compute the Euler approximation of a function on times
    ↪  t, with derivative dfun.
    """
    res = np.zeros_like(t)
    res[0] = f0

    for i in range(1, len(t)):
        res[i] = res[i-1] + (t[i]-t[i-1])*dfun(res[i-1])

    return res
```

Since we approximate the solution of a differential equation $p' = f(p, t)$, we used the trick of writing `dfun` as a function of p: this is why we call it by passing a point of `res` (and not of pyt). This trick makes it possible to compute it *together* with res itself (given the initial condition).

Two things together

A good way to keep two things separate (thus they can be changed independently), but together is the object-oriented approach: a `class` is a *small world* in which several computations are bound together, they share data and can depend one on each other.



OOP approach

```python
class EulerSolver:
    """An EulerSolver object computes the Euler approximation of a differential equation p'
    ↪  = f(p, t).

    Create it by giving the f function, then set the initial condition P0.
    The approximate solution on a given time span is computed by the method solve.
    """

    def __init__(self, f: Callable[[float, float], float]):
        self.f = f

    def set_initial_condition(self, P0: float):
        self.P0 = P0

    def solve(self, time: np.ndarray) -> np.ndarray:
        """Compute p for t values over time."""
        self.t = time
        self.p = np.zeros_like(self.t)
        # ....

    def _diff(self, i: int) -> float:
        """Compute the differential increment at time of index i."""

        assert i >= 0
        # ...
```

How to use it

```
time = np.linspace(0, 5, 100)

solver = EulerSolver(lambda p, t: 0.7*p)
solver.set_initial_condition(10)
euler = solver.solve(time)
```

What we have gained

Conceptual steps are separated (but kept together by the class). We can decide to change one of them independently. Object-oriented programming has a feature to make this easy: inheritance

```
class RKSolver(EulerSolver):
    def _diff(self, i: int) -> float:
        """Compute the differential increment at time
        ↪  of index i."""

        assert i >= 0
        # use Runge-Kutta now!
        # overridden functionality is available with
        # super()._diff(i)
```

RKSolver inherits the methods of EulerSolver and it overrides the method _diff.

Substitution principle

If inheritance is done properly (unfortunately not trivial in many cases), the new class can be used wherever the old one was.

```
solver = RKSolver(lambda p, t: 0.7*p)
solver.set_initial_condition(10)
rk = solver.solve(time)
```

Overridden methods must be executable when the old ones were and their must produce at least the "same effects" (Liskov's principle).

Lecture XXV: Probabilistic programming

How science works

Describing one single "scientific method" is problematic, but a schema many will accept is:

1. Imagine a hypothesis
2. Design (mathematical/convenient) models consistent with the hypothesis
3. Collect experimental data
4. Discuss the fitness of data given the models

It is worth noting that the falsification of models is not *automatically* a rejection of hypotheses (and, more obviously, neither a validation).

The role of Bayes Theorem

In this discussion, a useful relationship between data and models is Bayes Theorem.

$$P(M, D) = P(M|D) \cdot P(D) = P(D|M) \cdot P(M)$$

Therefore:

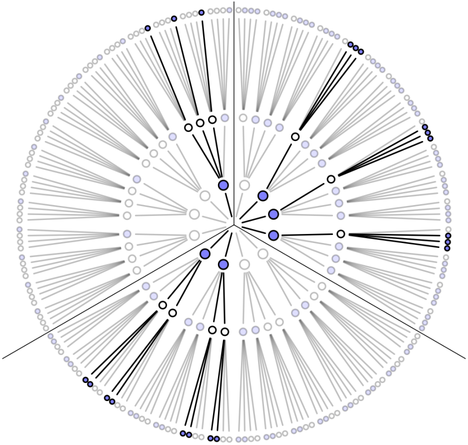
$$P(M|D) = \frac{P(D|M) \cdot P(M)}{P(D)}$$

The plausibility of the model given some observed data, is proportional to the number of ways data can be *produced* by the model and the prior plausibility of the model itself.

Simple example

- Model: a bag with 4 balls in 2 colors `B/W` (but we don't know which of `BBBB`, `BBBW`, `BBWW`, `BWWW`, `WWWW`)
- Observed: `BWB`
- Which is the plausibility of `BBBB`, `BBBW`, `BBWW`, `BWWW`, `WWWW`?

Bayes Theorem is counting



Picture from: R. McElreath, Statistical Rethinking

A computational approach

This Bayesian strategy is (conceptually) easy to transform in a computational process.

1. Code the models
2. Run the models
3. Compute the plausibility of the models based on observed data

Classical binomial example

- Which is the proportion p of water covering Earth? The models are indexed by the float $0 < p < 1$
- Given p, the probability of observing some `W`,`L` in a series of independent random observations is: $P(W, L|p) = \frac{(W+L)!}{W!\cdot L!} p^W \cdot (1-p)^L$ (binomial distribution).
- Do we have an initial (prior) idea?
- Make observations, apply Bayes, update prior!

A conventional way of expressing the model

$$\begin{aligned} W &\sim Binomial(W + L, p) \\ p &\sim Uniform(0, 1) \end{aligned}$$

Probabilistic programming is systematic way of coding this kind of models, combining predefined statistical distributions and Monte Carlo methods for computing the posterior plausibility of parameters.

In principle you can do it by hand

```python
def dbinom(success: int, size: int, prob: float) -> float:

    fail = size - success
    return math.factorial(size)/(math.factorial(success)*math.factorial(fail))*prob**succ
    ↪ ess*(1-prob)**(fail)
```

Then,

```python
W, L = 7, 3    # for example 'WWWLLWWLWW'
p_grid = np.linspace(start=0, stop=1, num=20)
prior = np.ones(20)/20

likelihood = dbinom(W, n=W+L, p=p_grid)

unstd_posterior = likelihood * prior

posterior = unstd_posterior / unstd_posterior.sum()
```

Unfeasible with many variables!

PyMC

```
import pymc as pm

W, L = 7, 3
earth = pm.Model()
with earth:
    p = pm.Uniform("p", 0, 1)  # uniform prior
    w = pm.Binomial("w", n=W+L, p=p, observed=W)
    posterior =  pm.sample(2000)

posterior['p']
```

Lecture XXVI: Behind pymc

Behind PyMC

The probabilistic programming approach of `PyMC` is built on two "technologies":

1. A library that mixes numerical and symbolic computations (Theano, Aesara, currently a new implementation called PyTensor)
2. Markov Chain Monte-Carlo (MCMC) algorithms to estimate posterior densities

PyTensor

It bounds numerical computations to its symbolic structure ("graph")

```
import aesara as at


a = at.tensor.dscalar()
b = at.tensor.dscalar()

c = a + b**2

f = at.function([a,b], c)

assert f(1.5, 2) == 5.5
```

Symbolic manipulations

Variables can be used to compute values, but also symbolic manipulations.

```
d = at.tensor.grad(c, b)

f_prime = at.function([a, b], d)

assert f_prime(1.5, 2) == 4.
```

Note you still need to give an a because the symbolic structure needs it.

Markov Chain Monte-Carlo

It's way of estimating (relative) populations of "contiguous" states.

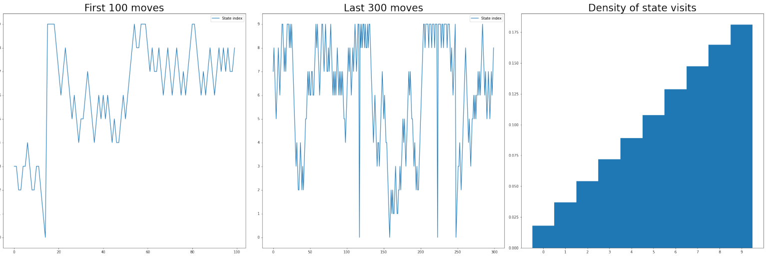
- It needs the capacity of evaluate the population/magnitude of any two close states (but a global knowledge of all the states *at the same time*)
- It's useful to estimate *posterior* distribution *without explicitly computing* $P(D)$: $P(M|D) = \frac{P(D|M) \cdot P(M)}{P(D)}$

Metropolis

The easiest MCMC approach is the so-called Metropolis algorithm (in fact appeared as Metropolis, N., **Rosenbluth, A., Rosenbluth, M.**, Teller, A., and Teller, E., 1953)

```
steps = 100000
positions = np.zeros(steps)
populations = [1,2,3,4,5,6,7,8,9,10]
current = 3

for i in range(steps):
    positions[i] = current
    proposal = (current + np.random.choice([-1,1])) %
    ↪  len(populations)
    prob_move = populations[proposal] /
    ↪  populations[current]
    if np.random.uniform(0, 1) < prob_move:
        current = proposal
```

Convergence



Eventual convergence is guaranteed, but it can be painful slow (and you dont't know if you are there...). Many algorithms try to improve: Gibbs, Hamiltonian-MC, NUTS...

Putting them together

```
import pymc as pm

linear_regression = pm.Model()

with linear_regression:
    # PyTensor variables
    sigma = pm.Uniform('sigma_h', 0, 50)
    alpha = pm.Normal('alpha', 178, 20)
    beta = pm.Normal('beta', 0, 10)
    mu = alpha + beta*(adult_males['weight'] -
    ↪  adult_males['weight'].mean())
    # Observed!
    h = pm.Normal('height', mu, sigma,
    ↪  observed=adult_males['height'])

    trace = pm.sample() # MCMC sampling
```