SCRIPTING AND PROGRAMMING LABORATORY FOR DATA ANALYSIS

Lecture 11

Some more advanced topics & some gaming
Sampling of distribution functions

Try, except

```
try:
    f = open("myfile.txt", "r")
except:
    print("File could not be opened!")
```

```
some statements
try:
  statement1
except:
  statement2
```

This construct allows handle exceptions in a clever way.

```
some statements
try:
  statement1
except exception:
  statement2
```

The program executes the **try** block. If an exception (error) is encountered, the **except** block is executed.

```
some statements
try:
  statement1
except exception1:
  statement2
except exception2:
  statement3
except:
  statement4
```

```
try:
    f = open("myfile.txt", "r")
except FileNotFoundError:
    print("File could not be found!")
except:
    print("Another exception occurred")
```

Try, except, else, finally

The try-except construct can be also used with an **else** block. The else block is executed if no exception is raised within the try block.

```
some statements
try:
   statement1
except:
   statement2
else:
   statement2
```

```python
try:
    f = open("myfile.txt", "r")
except :
    print("File could not be found!")
else:
    print("File has been successfully opened!")
```

The try-except construct can be terminated with a **finally** clausole, which is executed no matter what. In general, this is used to release external resources.

```
some statements
try:
   statement1
except:
   statement2
finally:
   statement_end
```

```python
f = open("myfile.dat", "r")
try:
    l = f.readlines()
    w = [int(i) for i in l]
except ValueError:
    print("Lines do not contain integers!")
finally:
    f.close()
```

@DECORATORS

Remember: everything in python is an object, including functions!

Example:

```python
def first(msg):
    print(msg)
```

```python
first("Hello")
```

```
Hello
```

```python
second = first
second("Hello")
```

```
Hello
```

This implies that functions can be passed as arguments to other functions (so-called *higher-order functions*) and can *return* other functions; in addition, functions can be defined within functions, and embedded functions are not destroyed right after the main function call (closure).

Examples:

```python
[1]: def increment(x):
         return x + 1

     def decrement(x):
         return x - 1

     def operate(func, x):
         result = func(x)
         return result
```

```python
[2]: operate(increment,3)
```

```
[2]: 4
```

```python
[3]: operate(decrement,3)
```

```
[3]: 2
```

```python
[1]: def is_called():
         def is_returned():
             print("Hello")
         return is_returned
```

```python
[2]: new = is_called()
     new()
```

```
Hello
```

@DECORATORS

Functions and methods are called *callable* as they can be called. In fact, any object which implements the special `__call__()` method [see slides before] is said to be callable. So, in the most basic sense, a decorator is a callable that returns a callable.

A decorator takes in a function, adds some functionality and returns it.

Without decorators:

```
[1]: def make_pretty(func):
         def inner():
             print("I got decorated")
             func()
         return inner


     def ordinary():
         print("I am ordinary")
```

```
[2]: ordinary()
```

```
I am ordinary
```

```
[3]: pretty = make_pretty(ordinary)
     pretty()
```

```
I got decorated
I am ordinary
```

```
[4]: #reassign
     ordinary = make_pretty(ordinary)
     ordinary()
```

```
I got decorated
I am ordinary
```

Analogous with decorators:

```
[1]: def make_pretty(func):
         def inner():
             print("I got decorated")
             func()
         return inner

     #decorated function:
     @make_pretty
     def ordinary():
         print("I am ordinary")

     #equivalent to
     ## ordinary = make_pretty(ordinary)
```

```
[2]: ordinary()
```

```
I got decorated
I am ordinary
```

@DECORATION WITH PARAMETERS

Without decorators:

```
[1]: def divide(a, b):
         return a/b

[2]: divide(5,3)

[2]: 1.6666666666666667

[3]: divide(4,0)

-------------------
ZeroDivisionError
~\AppData\Local\Tem
```

Decorated, fixed num. Of parameters:

```
[4]: def smart_divide(func):
         def inner(a, b):
             print("I am going to divide", a, "and", b)
             if b == 0:
                 print("Whoops! cannot divide")
                 return

             return func(a, b)
         return inner


     @smart_divide
     def divide(a, b):
         print(a/b)

[5]: divide(4,0)

     I am going to divide 4 and 0
     Whoops! cannot divide
```

Decorated, arbitrary params:

```
[1]: def smart_divide(func):
         def inner(*args, **kwargs):
             print("I am going to divide", args)
             if args[-1] == 0:
                 print("Whoops! cannot divide")
                 return

             return func(*args, **kwargs)
         return inner


     @smart_divide
     def divide(a, b):
         print(a/b)

[2]: divide(4,0)

     I am going to divide (4, 0)
     Whoops! cannot divide
```

Note: parameters of the nested inner() function inside the decorator is the same as the parameters of functions it decorates. ***This can be generalized!*** →

Chaining decorators

Multiple decorators can be chained in Python.

A function can be decorated multiple times with different (or same) decorators. The decorators should be placed above the desired function. The last above the function call is the first to be applied.

```
[3]: def star(func):
         def inner(*args, **kwargs):
             print("*" * 30)
             func(*args, **kwargs)
             print("*" * 30)
         return inner


     def percent(func):
         def inner(*args, **kwargs):
             print("%" * 30)
             func(*args, **kwargs)
             print("%" * 30)
         return inner
```

```
[4]: @percent
     def printer(msg):
         print(msg)
```

```
[5]: printer("Hello")
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Hello
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
[6]: @percent
     @star
     def printer2(msg):
         print(msg)
```

```
[8]: printer2("Hello")
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
******************************
Hello
******************************
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Iterables

Iterable: *An object is said to be iterable if we can get an iterator from it – that is, it can return its members one at a time.*

Lists, tuples, strings, and dictionaries are all iterables

Objects that define either of the ***__iter__*** *or* ***__getitem__*** *methods are iterables (also customized ones)*

The ***iter()*** *function (which in turn calls the* ***__iter__()*** *method) returns an* **iterator** *from them.*

```
[2]: list_num = [4,5,7]
     for i in list_num:
         print(i)

4
5
7
```

```
: list_num = [4,5,7]
  print(dir(list_num))

['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

ITERATORS

Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.

Technically speaking, *a Python iterator object must implement two special methods,* ***__iter__()*** *and* ***__next__()***, collectively called the **iterator protocol**.

Note: object.__iter__() and iter(object) do nearly the same thing! Same for object.__next__() and next(object)

```
list_num = [4,5,7]
print(iter(list_num))
```

```
<list_iterator object at 0x000001B31C0163A0>
```

```
[1]: list_num = [4,5,7]
     num_it=list_num.__iter__()
     val1=num_it.__next__()
     print(val1)

     4

[2]: val2=num_it.__next__()
     print(val2)

     5

[3]: val3=num_it.__next__()
     print(val3)

     7
```

equivalent

```
[1]: list_num = [4,5,7]
     num_it=iter(list_num)
     val1=next(num_it)
     print(val1)

     4

[2]: val2=next(num_it)
     print(val2)

     5

[3]: val3=next(num_it)
     print(val3)

     7
```

ITERATORS AND LOOPS

next() manually iterates through all the items; when it reaches the end, it raises the StopIteration Exception.

for loops are more elegant ways of iterating. A for loop is in fact an infinite loop that stops when the ***StopIteration*** Exception is raised!

```
[1]: list_num = [4,5,7]
     num_it=iter(list_num)
     val1=next(num_it)
     print(val1)

4

[2]: val2=next(num_it)
     print(val2)

5

[3]: val3=next(num_it)
     print(val3)

7

[4]: val4=next(num_it)
     print(val4)

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Input In [4], in <cell line: 1>()
----> 1 val4=next(num_it)
      2 print(val4)

StopIteration:
```

```
list_num = [4,5,7]
for i in list_num:
    print(i)

4
5
7
```

Implemented as:

```
list_num = [4,5,7]
my_iterator = iter(list_num)
while True:
  try:
    i = next(my_iterator)
    #do stuff, for example:
    print(i)
  except StopIteration:
    break

4
5
7
```

BUILD YOUR CUSTOM ITERATOR

When you define your own class, you can make it iterable by defining the **__iter__()** and **__next__()** functions.

__iter__() can contain initializations and should return the iterator itself

__next__() returns the next item. When there is no next element, it should raise a **StopIteration** Exception

[1]:
```
class PowTwo:
    """Class to implement an iterator
    of powers of two"""

    def __init__(self, max=0):
        self.max = max

    def __iter__(self):
        self.n = 0
        return self

    def __next__(self):
        if self.n <= self.max:
            result = 2 ** self.n
            self.n += 1
            return result
        else:
            raise StopIteration
```

[2]:
```
numbers = PowTwo(3)
it = iter(numbers)
print(next(it))
```

```
1
```

[3]:
```
print(next(it))
print(next(it))
print(next(it))
print(next(it))
```

```
2
4
8
---------------------------
StopIteration
<ipython-input-3-3c913438779d> in <mod
      2 print(next(it))
```

[4]:
```
for i in PowTwo(5):
    print(i)
```

```
1
2
4
8
16
32
```

Infinite iterators

An iterator can be defined so that it never ends.

But: not recommended!!

```
[1]: class InfIter:
         """Infinite iterator
         to return odd numbers"""

         def __iter__(self):
             self.num = 1
             return self

         def __next__(self):
             num = self.num
             self.num += 2
             return num
```

```
[2]: oddnum = InfIter()
     it = iter(oddnum)
     print(next(it))
```

```
1
```

```
[3]: print(next(it))
     print(next(it))
     print(next(it))
```

```
3
5
7
```

GENERATORS

Generators are simple ways of generating some types of iterators without having to rely on a class.

In fact, a generator is a normal functions that returns an iterator, over which we can iterate. The function scope should contain a **yield** statement.

yield ***pauses*** the function, saving all its current states for the next successive calls

This is long and counterintuitive:

```python
class Even:
    """Returns even numbers"""
    def __init__(self, nmax):
        self.n=2
        self.max=nmax
    def __iter__(self):
        return self
    def __next__(self):
        result = self.n
        self.n += 2
        if self.n<=self.max:
            return result
        else:
            raise StopIteration
```

```python
numbers = Even(12)
print(next(numbers))
print(next(numbers))
print(next(numbers))
```

```
2
4
6
```

A generator is a much better solution (here still not optimal, see next slide):

```python
[1]: def my_gen():
         n = 2
         print('This is printed first')
         yield n

         n += 2
         print('This is printed second')
         yield n

         n += 2
         print('This is printed at last')
         yield n
```

```python
[2]: a = my_gen()
     print(next(a))
```

```
This is printed first
2
```

```python
[3]: print(next(a))
```

```
This is printed second
4
```

```python
[4]: print(next(a))
```

```
This is printed at last
6
```

Generators

Difference between a generator and a normal function:

- A generator function contains **yield** statements.
- When called, it returns an object (iterator) *but does not start execution immediately*.
- Methods like **__iter__()** and **__next__()** are implemented automatically → we can iterate through the items using next().
- Once the function yields, it is is paused and the control is transferred to the caller.
- **Local variables and their states are remembered between successive calls.**
- When the function terminates, **StopIteration** is raised automatically.

```
[1]: def EvenGen(max=0):
         n = 0
         while n < max:
             yield 2*n
             n += 1
```

```
[2]: a=EvenGen(2)
     print(next(a))

0
```

```
[3]: print(next(a))

2
```

```
[4]: print(next(a))

-----------------------
StopIteration
<ipython-input-4-94b12d
----> 1 print(next(a))

StopIteration:
```

Note: we have to create a new object to restart the generator

```
[5]: b=EvenGen(5)
     for val in b:
         print(val)

0
2
4
6
8
```

same

```
for val in EvenGen(5):
    print(val)

0
2
4
6
8
```

MORE ON GENERATORS

A more useful example:

```
[1]: def rev_str(my_str):
         length = len(my_str)
         for i in range(length - 1, -1, -1):
             yield my_str[i]


     # For loop to reverse the string
     for char in rev_str("hello"):
         print(char)
```

```
o
l
l
e
h
```

Normally, generator functions are implemented with a loop having a suitable terminating condition.

Python generator expression:

```
[1]: my_list = [1, 3, 6, 10]
     # square each term
     # using list comprehension []
     list_ = [x**2 for x in my_list]

     # using a generator ()
     generator = (x**2 for x in my_list)
```

```
[2]: print(list_)
```

```
[1, 9, 36, 100]
```

```
[3]: print(generator)
```

```
<generator object <genexpr> at 0x000001F77CFEC120>
```

```
[4]: a = (x**2 for x in my_list)
     print(next(a))
     print(next(a))
     print(next(a))
     print(next(a))
```

```
1
9
36
100
```

```
[5]: for val in (x**2 for x in my_list):
         print(val)
```

```
1
9
36
100
```

Generator expressions work very similarly to *list comprehension*, but they are surrounded by parentheses

List comprehension immediately produces all elements, generator expressions are “lazy”: they produce an element at a time

→ **Memory efficient!**

Note: generators can be used as arguments for functions!

```
sum(x**2 for x in my_list)
```

```
146
```

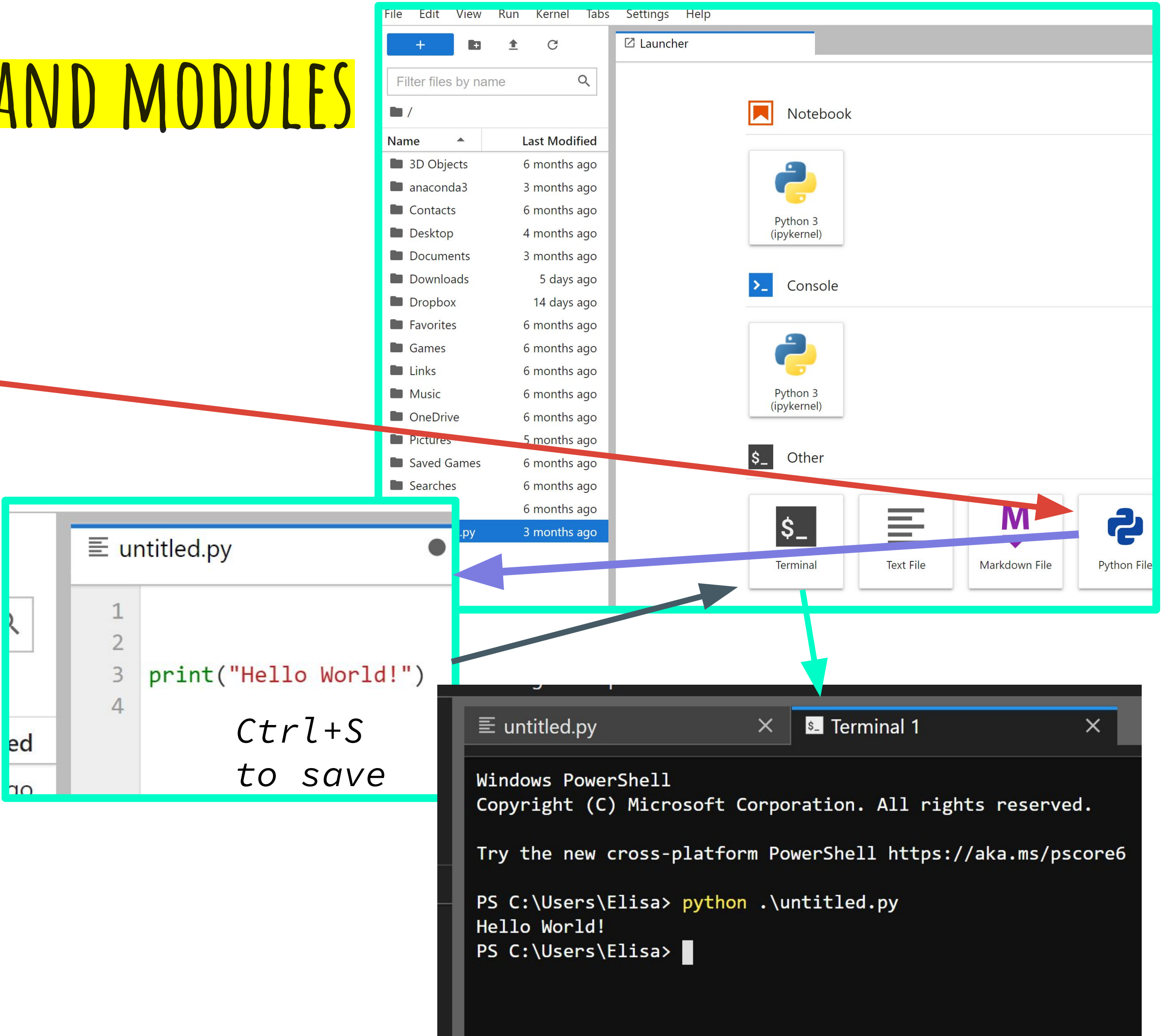
WRITE YOUR OWN SCRIPT AND MODULES

Any file with extension **.py** can be a module (and your own script!)

1. From the launcher: create a file .py in a directory, put there the code to be executed. Save it!
2. Again from the launcher: open a **Terminal** in the same directory (or navigate there) and execute the program with:

   python name.py

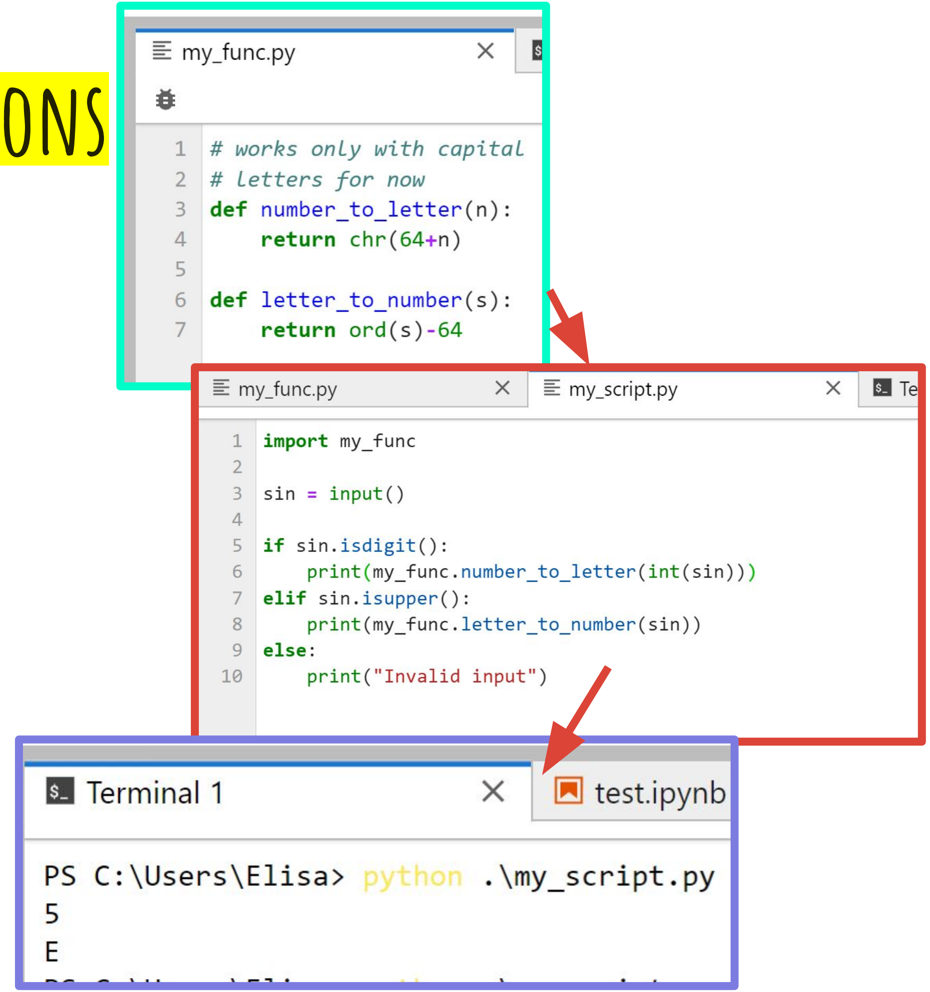
   Then press Enter



Create a module with functions

Knowing this, **you can create your own module to be imported if needed**. A module can contain the definitions of classes, functions and so forth.

The name of the module should then be imported in the notebook or .py file in which we want to use it.



Note: the **input()** command takes as input the string from the prompt!

MONTE CARLO METHODS: GENERATE YOUR DISTRIBUTION FUNCTIONS

Broadly speaking, *Monte Carlo methods are methods that rely on the generation of random numbers*. They are non-deterministic, but they can give approximate results in a number of situations. Monte Carlo methods are frequently adopted for the solution of problems where the complexity of the model makes a direct solution method infeasible.

Problem: Let's assume that we want to generate a set of data between a certain minimum and maximum value, that follow a given distribution function $f(x)$:

How can we do it?

Important notions

Distribution function:

$f(x)\,\mathrm{d}x$ represents the probability $\mathscr{P}$ for x to fall in the range $\{x, x+\mathrm{d}x\}$. This means that the probability for x to assume a value in the interval $\{a, b\}$ is given by

$$\mathscr{P}\{a,b\} = \int_a^b f(x)\,\mathrm{d}x.$$

Must be normalized to 1

Cumulative distribution function:

cumulative distribution function (CDF) $F(X)$, which represents the probability for x to have a value minor or equal to X:

$$F(X) = \int_{-\infty}^{X} f(x)\,\mathrm{d}x = \mathscr{P}\{x < X\}.$$

It is an increasing function between 0 and 1

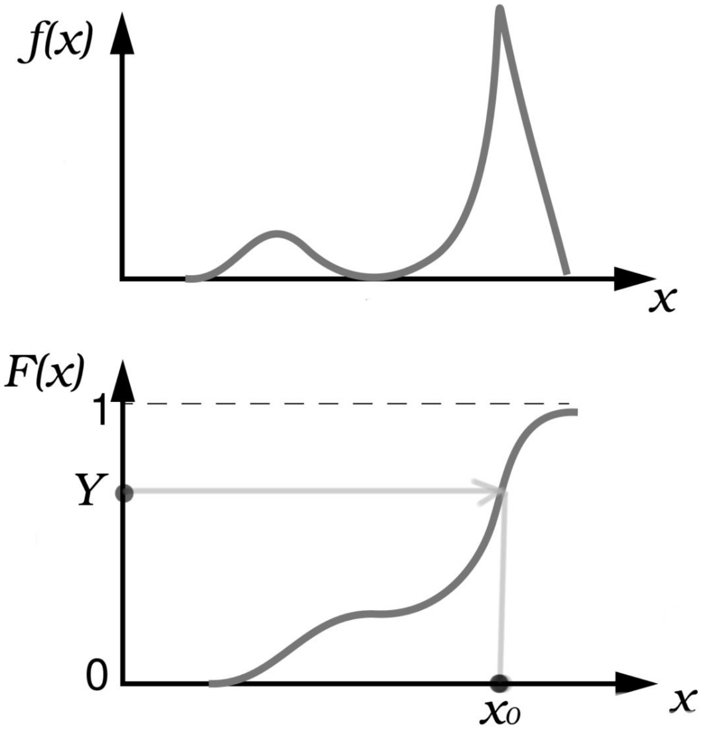
It follows that:

$$\frac{\mathrm{d}F(x)}{\mathrm{d}x} = f(x);$$

$$\mathscr{P}\{a,b\} = \int_a^b f(x)\,\mathrm{d}x = F(b) - F(a).$$

Let's also recall the fundamental law of probability:

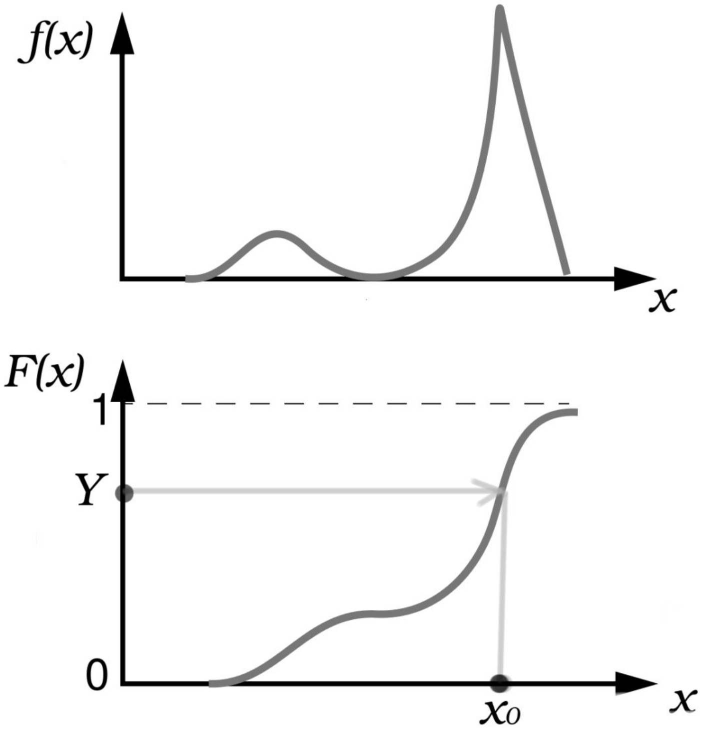
Monte Carlo transformation method



Basic idea: if ***F(x) = Y*** is a value assumed by the cumulative distribution function generated from ***f(x)***, it must be uniformly distributed between 0 and 1.

This notion can be used to generate values distributed as f(x). NOTE: this method relies on the assumption that we are able to compute the inverse of the cumulative *F(x)!!* [not necessarily analytically]

Monte Carlo transformation method



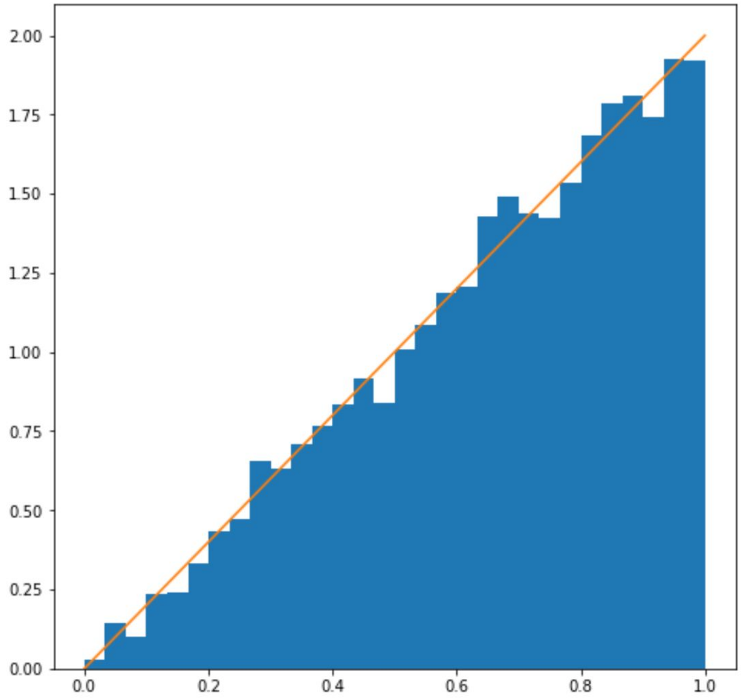
1. Take a probability distribution function f(x) of the quantity x you want to sample
2. Compute its cumulative distribution function (analytically if possible, otherwise numerically, but be careful!)
3. Extract a random number for Y = F(x) of the cumulative distribution function between 0 and 1
4. Invert the function F(x) to get $X=F^{-1}(Y)$; X is distributed according to f(x)

REPEAT 3 & 4 as many times as you need to get your complete sample

Monte Carlo transformation method

An example from jupyter notebook

```
# actual execution of the monte carlo sampling
# f(x) = 2*x
# F(x) = x^2
# --> x = sqrt(Y)
Y = np.random.uniform(0,1,10000)
X = np.sqrt(Y)
fig, ax=plt.subplots(figsize=(8,8))
plt.label("hist(x)")
plt.hist(X,bins=30,range=(0,1),density=True)
plt.plot(x,2*x)
plt.show()
```



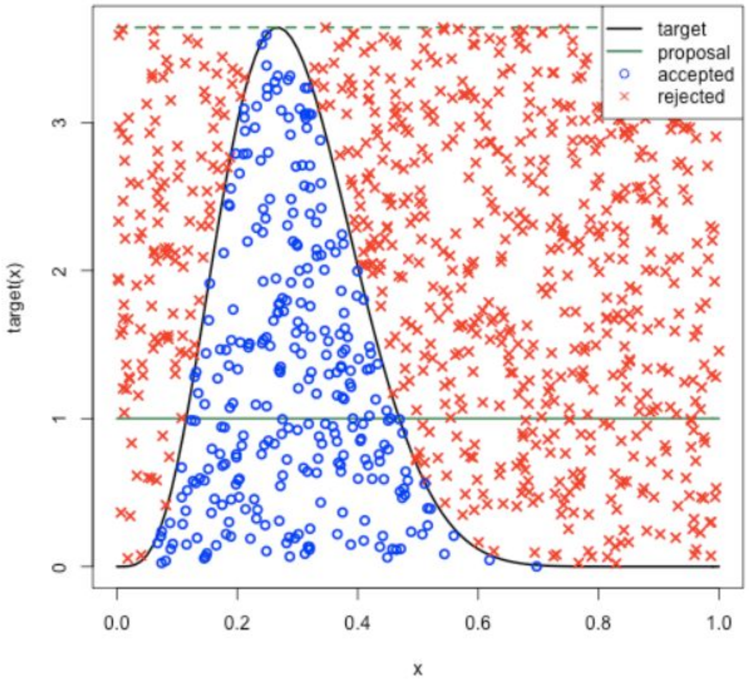
Monte carlo rejection method

The inverse transform sampling is not useful when F^{-1} does not have an explicit form. In this situations a different techniques, always using the inverse sampling, can be used. Suppose we have an additional pdf function $g(x)$ with a cdf that has an explicit form and such that $f(x) \leq M g(x)$ over the whole domain on which $f(x)$ is defined, with M a positive constant. In this case to get x samples we use the inverse sampling on the cdf of $g(x)$. Then in order to decide if we can accept the x value we proceed as follows:

- we sample a random number u from a uniform distribution with limits 0 to $M g(x)$;
- if $u \leq f(x)$ we accept x, otherwise we reject it.

Note that in principle for using this method the pdf does not have to be normalized at 1, since any shift by a constant do not change the relative difference between $f(x)$ and $g(x)$.

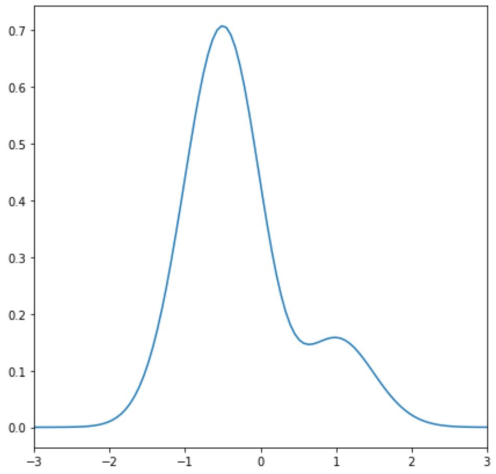
The simplest $g(x)$ that we can choose is a constant function i.e. $g(x) = C$, with $C \geq \max(f(x))$:



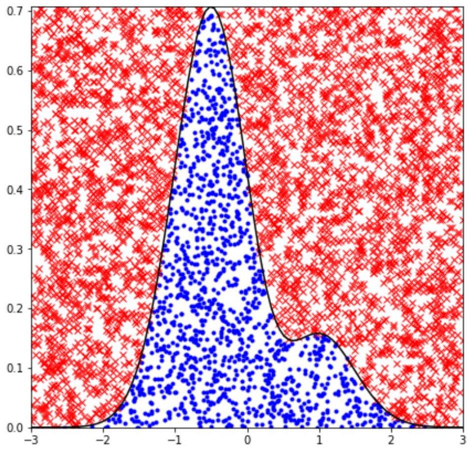
Monte carlo rejection method

xrand is now distributed as f(x)

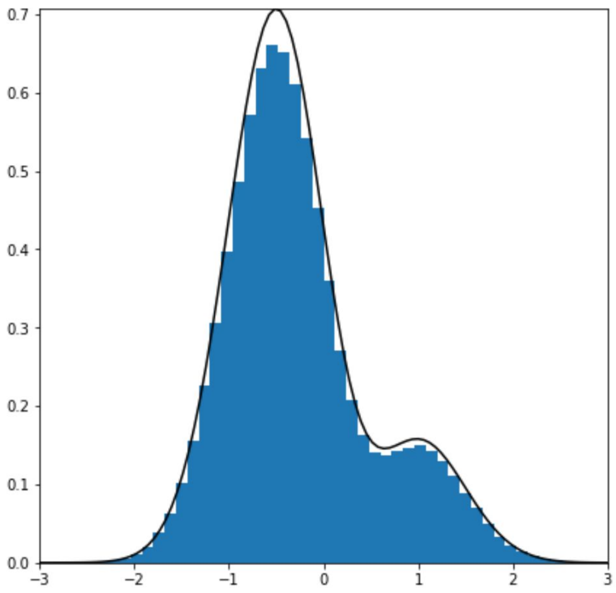
```python
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return np.sin(x-0.5)**2 * np.exp(-x**2) + .2*np.exp(-x**2

fig,ax = plt.subplots(figsize=(7,7))
x=np.linspace(-3,3,100)
plt.plot(x, f(x))
plt.xlim(-3,3)
plt.show()
```




```python
N=5000
XRAND = np.random.uniform(-3,3,N)
YRAND = np.random.uniform(0,np.amax(f(x)), N)
filt = YRAND < f(XRAND)
fig,ax = plt.subplots(figsize=(7,7))
plt.plot(XRAND[filt], YRAND[filt], "b.")
plt.plot(XRAND[~filt], YRAND[~filt], "rx")
plt.plot(x, f(x), "k-")
plt.xlim(-3,3)
plt.ylim(0, np.amax(f(x)))
plt.show()
```




```python
plt.hist(XRAND[filt], bins=50,range=(-3,3), density=True)
plt.plot(x, f(x), "k-")
plt.xlim(-3,3)
plt.ylim(0, np.amax(f(x)))
plt.show()
```



STEP UP YOUR CODING GAME

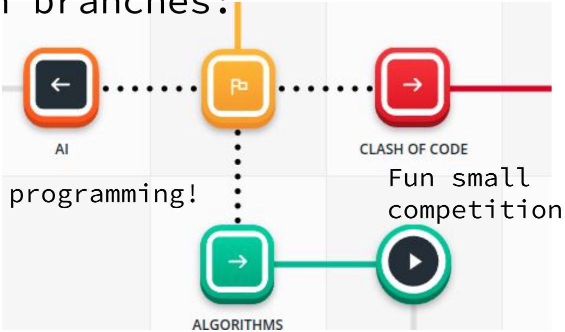
The new way to improve your programming skills while having fun and getting noticed

GET STARTED

Practice & learn the fun way

CODINGAME.COM

Registration needed
First: solve a simple puzzle
Then: decide on what you want to do!
Main branches:



Bot programming!

Fun small competitions