

# Introduction to Python

Security (CT0657) 2026-27  
Università Ca' Foscari Venezia

Riccardo Focardi

<https://www.unive.it/people/focardi>  
[secgroup.dais.unive.it](https://secgroup.dais.unive.it)



Università  
Ca' Foscari  
Venezia

# Why Python?

**Standard** in the **IT security** industry, many tools are written in Python or support plugins/bindings in Python

**Flexible**: supports multiple **paradigms** (imperative, object-oriented, functional)

**Highly supported**: huge **library**

**Interpreted**: quick **prototyping**

**Extensible**: add **built-in modules** in C

**Dynamic typing**: no static types but strong **dynamic types**. Forbids non well-typed operations at runtime: no subtle errors such as in **PHP** or **JS**

**Simple syntax**: easy to read, easy to write: no semicolons “;”, no curly braces “{”

**Warning**: indentation matters!

# Python interpreter

The interpreter, with no arguments, starts in **interactive mode**

⇒ Useful for simple experiments and getting used with data types and commands

We will use **python3** (python will run python 2.7 so use python3)

```
$ python3
Python 3.6.9 (default, Nov  7 2019, 10:44:02)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello, World!')
Hello, World!
>>>
```

# Numbers

```
>>> 3+4
7
>>> 4-2
2
>>> 3*4
12
>>> 3/4    # float division
0.75
>>> 3//4    # integer division
0
>>> 4%3     # modulo
1
>>> 3**4    # exponentiation
81
```

## Dynamic typing

```
>>> 3/1
3.0
>>> 3//1
3
>>> type(3)
<class 'int'>
>>> type(1)
<class 'int'>
>>> type(3/1)
<class 'float'>
>>> type(3//1)
<class 'int'>
```

# Variables

Variables do not need to be declared and are just assigned

```
>>> a = 3+2
>>> a
5
>>> a = a*2
>>> a
10

>>> print(a) # works with any type
10
```

Dynamic typing

```
>>> type(a)
<class 'int'>
>>> a = 3.0
>>> type(a)
<class 'float'>

>>> a = 3          # integer
>>> a = a/2
>>> type(a)
<class 'float'>
```

# Strings

**Delimiters:** strings are delimited by either ' or "

```
>>> 'ciao'
'ciao'
>>> "ciao"
'ciao'
>>> ' "ciao" '   # nested
' "ciao" '
>>> " 'ciao' "   # nested
" 'ciao' "
>>> ' \'ciao\' '  # escaped
" 'ciao' "
```

**Multiline:** strings delimited by """ or ''' are multiline

```
>>> """
... hello
... this
... is
... multiline
... """
'\nhello\nthis\nis\nmultiline\n'
```

# Strings: basic functions

**len**: returns the length of a string

```
>>> a = 'ciao'
>>> len(a)
4
>>> print(a)
ciao
```

**+**: concatenates two strings

```
>>> print(a + ' ' + a)
ciao ciao
```

**format**: takes a format string and replaces arguments (like printf in C)

{ [field] [! conv] [: format] }

**field**: name or position

**conv**: s ([str](#)), r ([repr](#)), a ([ascii](#)), useful with objects

**format**: specifies the format (fill, alignment, width, precision, type). See the [documentation](#) for detail

# Format string examples:

```
>>> 'ciao {0} {1}'.format('Riccardo', 'Focardi')      # position of args
'ciao Riccardo Focardi'

>>> 'Coordinates: {lat}, {long}'.format(lat='37.24N', long='-115.81W')
'Coordinates: 37.24N, -115.81W'                      # name of args

>>> '{:>30}'.format('right aligned')   # no position/name takes the first
'right aligned'                        # format specification after :

>>> 'int: {0:d};  hex: {0:x};  oct: {0:o};  bin: {0:b}'.format(42)
'int: 42;  hex: 2a;  oct: 52;  bin: 101010'          # arg 0 in various formats

>>> n = 42
>>> f'int: {n:d};  hex: {n:x};  oct: {n:o};  bin: {n:b}'
'int: 42;  hex: 2a;  oct: 52;  bin: 101010'          # f-string f'...'
```

# Strings slicing

Strings can be accessed:

**s[i]** with the **index** as arrays: returns a single char

**s[i:j]** **slicing** to obtain another string: returns the substring from i to j-1

**s[i:j:s]** **slicing** with step s (one char every s)

**Note:** index starts from 0

```
>>> 'ciao'[2]
'a'                                # third char
>>> 'ciao'[2:4]
'ao'                               # 2 to 3
>>> 'ciao'[-1]
'o'                                # 4-1 = 3
>>> 'ciao'[-2:-4]
''                                  # empty! (no err)
>>> 'ciao'[-2:-4:-1] # OK!
'ai'                               # -2 to -3
>>> 'ciao'[::-1]     # correct?
'oaic'               # reverse!
>>> 'ciao'[:]
'ciao'                # the full string
```

# str class

**Positions** in Python strings:

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
  0   1   2   3   4   5
-6  -5  -4  -3  -2  -1
```

**Immutable**: cannot **modify** elements,  
need to create a new string

**No overflow**: **limits** are checked at  
runtime (dynamic types)

```
>>> s = 'Python'
>>> s[0] = 'p'
TypeError: 'str' object does not
support item assignment

>>> s = 'p' + s[1:]
>>> s
'python'           # this is a new string

>>> s[6]
IndexError: string index out of range
>>> s[-7]
IndexError: string index out of range
```

# Lists

Versatile data type supporting  
**indexing, slicing, modification**  
(mutable), append (+)

Can contain items of **different types**

```
>>> l = [1,2,3]
>>> m = ['a', 'b', 'c']

>>> l + m
[1, 2, 3, 'a', 'b', 'c']
```

## Slicing

```
>>> l[1:]
[2, 3]
```

```
>>> m[1:3]
['b', 'c']
```

## Modification (mutable):

```
>>> l[0] = 0
>>> l
[0, 2, 3]
```

# Lists: assignment to slices

Slicing can also be used for modifying a substring

```
>>> m
['a', 'b', 'c']

>>> m[1:3]
['b', 'c']

>>> m[1:3] = ['B', 'C']
>>> m
['a', 'B', 'C']
```

Single index returns an **element** while a slice returns a **list**

```
>>> m[1]
'B'

>>> m[1:2]
['B']

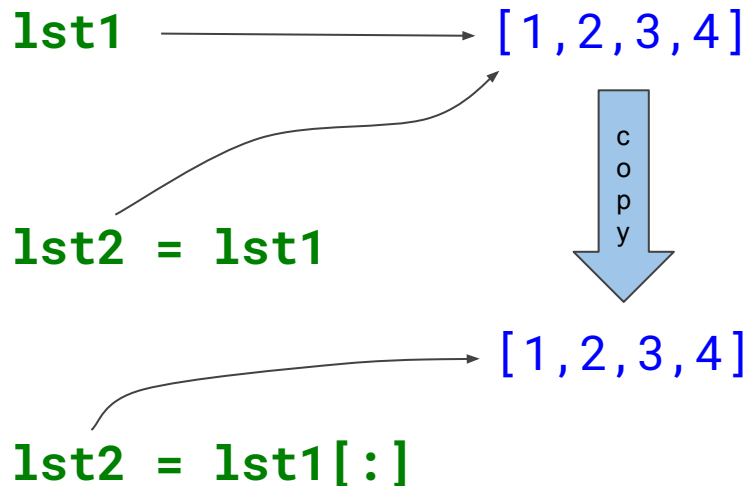
>>> m[1] = [1,2,3,4,5]
>>> m
['a', [1, 2, 3, 4, 5], 'C']
>>> m[1:2] = [1,2,3,4,5]
>>> m
['a', 1, 2, 3, 4, 5, 'C']
```

# Reference vs. copy

Lists are references, assignment does not copy the list!

**lst2 = lst1** copies the **reference**:  
any change to lst2 will also affect  
lst1 and vice-versa

**lst2 = lst1[:]** copies the **list**:  
any change to lst2 will only affect  
lst2's copy and vice-versa



# Exercise 1: Reverse word order

Print the words found in a given string in reverse order

**INPUT:** 'This is a test in which we revert the order of words'

**OUTPUT:** 'words of order the revert we which in test a is This'

Useful links: [split](#), [join](#)

# Control flow (indentation matters!)

## if-then-else

```
if a == b:
    print('a equal b')
elif a == c:
    print('a equal c')
else:
    print('different')
```

## for loop

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

## while loop

```
a = 0
while True:
    if a%2 != 0:
        a += 1
        continue
    try:
        print(a, 10/a)
    except:
        pass
    if a == 10:
        break
    a += 1
```

# Functions

## Definition

```
def sum(x,y):  
    return x+y
```

## Keyword arguments

```
def sum(x=0,y=0):  
    return x+y
```

Keyword arguments have default values

```
>>> sum()  
0  
>>> sum(1)  
1  
>>> sum(1,2)  
3  
>>> sum(y=3)  
3  
>>> sum(y=3,x=1)  
4  
>>> sum(1,y=3,x=1)  
TypeError: sum() got multiple values  
for argument 'x'
```

# Python programs and modules

**Shebang (hashbang):** tells how to execute the interpreter:

```
#!/usr/bin/env python3
```

**Encoding:** what encoding to use

```
# -*- coding: utf-8 -*-
```

Python executes any code which is not in a function, even when you import it. “Isolate” main as:

```
if __name__ == '__main__':  
    main()
```

**Modules:** sys contains functions related to execution:

```
import sys
```

**Example:** sys.argv command line arguments

It is possible to include python programs in other programs and use them as modules

# Exercise 2: Caesar cipher

Decrypt a given ciphertext encrypted with Caesar cipher (letters are shifted 3 position ahead in the alphabet)

ABCDEFGHIJKLMNOPQRSTUVWXYZ  
DEFGHIJKLMNOPQRSTUVWXYZABC

Lq fubswrjudskb, d Fdhvdu flskhu, dovr nqrzq dv Fdhvdu'v flskhu, wkh vkliw flskhu, Fdhvdu'v frgh ru Fdhvdu vkliw, lv rqh ri wkh vlpsohvw dqg prvw zlghob nqrzq hqfubswlrq whfkqltxhv. Lw lv d wbsh ri vxevwlwxwlrq flskhu lq zklfk hdfk ohwwhu lq wkh sodlqwhaw lv uhsodfhg eb d ohwwhu vrph ilahg qxpehu ri srvlwlrqv grzq wkh doskdehw. Iru hadpsoh, zlwk d ohiw vkliw ri 3, G zrxog eh uhsodfhg eb D, H zrxog ehfrph E, dqg vr rq. Wkh phwkrg lv qdphg diwhu Mxolxv Fdhvdu, zkr xvhg lw lq klv sulydwh fruuhvsrcghqfh

Useful links: [chr](#), [ord](#), [join](#), [string methods](#) (ciphertext: /home/rookie/Python)

# Functional programming

Iterators are objects representing **streams of data**

**next(iterator)**: generates the next element of the stream  
⇒ the element is **removed**!

**Note**: iterators generate elements on-the-fly and update the state

**iter(iterable)**: generates an **iter-ator** from any *iter-able*

```
>>> i = iter([1,2,3])
>>> i
<list_iterator object at 0x7f693...>
>>> next(i)
1
>>> next(i)
2
>>> next(i)
3
>>> next(i)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

# Functional programming

**filter**(f, iterable): returns an iterator such that  $f(item)$  is true

**map**(f, iterable): returns iterator of transformed items  $f(item)$

**map**(f, i1, i2, ...): returns iterator of items  $f(e1, e2, \dots)$  where  $e1, e2, \dots$  are from  $i1, i2, \dots$

**range**(n): iterable yielding numbers from 0 to  $n-1$

```
>>> list( filter(lambda x:x%2==0, range(10)) )  
[0, 2, 4, 6, 8]  
>>> list( map(lambda x,y:x*y, range(10), range(10)) )  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]  
>>> list( map(lambda x:x*x, range(10)) )  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

# List comprehensions

Create a list using iterables

Can use many **for** and **if** constructs

Can be nested

```
[(x,y,...) for x in i1
            for y in i2 ... ]
```

**Note:** list is  $\text{len}(i1) \times \text{len}(i2) \times \dots$  long

```
[(x,y,...) for x in i1 if c1
            for y in i2 if c2 ... ]
```

Conditions  $c1, c2, \dots$  should hold

Pairs (even,odd) numbers

```
>>> [ (x,y) for x in range(4) if x%2==0
...      for y in range(4) if y%2==1 ]
[(0, 1), (0, 3), (2, 1), (2, 3)]
```

Nested comprehension:

```
>>> [ x*x for x in
...     [ y for y in range(10) if y%2==0 ]
... ]
[0, 4, 16, 36, 64]
```

# if-then-else in list comprehensions

If you need to use **if-then-else** in list comprehensions the **for** should go **after** the conditional:

```
[ e1 if conditional else e2 for item in list ]
```

```
[ e1 if c1 else  
  e2 if c2 else e3 ...  
  for item in list ]
```

**Exercise 2:** try to solve exercise 2 using list comprehensions

# Sets, tuples, dictionaries

**Sets:** unique elements, no ordering

**in:** membership testing: `1 in {1,2,3}`

**|:** set union `{1,2,3}|{4}` is `{1,2,3,4}`

**&:** set intersection `{1,2,3}&{3,4}` is `{3}`

**Tuples:** immutable sequences

**packing :** `x = (1,2,3)` or `x = 1,2,3`

**unpacking :** `y,z,w = x`

**singleton:** `x = (1,)`   **empty:** `x=()`

**Dictionaries:** associative arrays indexed by (unique) keys

**in:** membership, `'a' in {'a': 2, 'b': 5}`

**Add element:** `d[ ' z ' ] = 6`

**Del element:** `del d[ ' z ' ]`

`d.keys()`: [view](#) of d's keys

`d.values()`: [view](#) of d's values

`d.items()`: [view](#) of d's pairs

# Exercise 3: Frequency analysis

Print the list of pairs (character, number of occurrences) found in a given string, sorted by the number of occurrences

```
exercise.frequency('This is a test in which we count the frequency of  
letters. Guess what? Blank space is the most frequent!')
```

**OUTPUT:** [(' ', 19), ('e', 12), ('t', 10), ('s', 9), ('h', 6), ('i', 5),  
('n', 5), ('a', 4), ('c', 4), ('u', 4), ('w', 3), ('o', 3), ('f', 3), ('r',  
3), ('q', 2), ('l', 2), ('T', 1), ('y', 1), ('.', 1), ('G', 1), ('?', 1),  
('B', 1), ('k', 1), ('p', 1), ('m', 1), ('!', 1)]

Useful links: [dictionaries](#), [sorted](#)