

INTRODUCTION TO Python



Introduction to Python

<https://staff.polito.it/giovanni.squillero/lectures/>

Copyright © 2023 by Giovanni Squillero.

Permission to make digital or hard copies for personal or classroom use of these files, either with or without modification, is granted without fee provided that copies are not distributed for profit, and that copies preserve the copyright notice and the full reference to the source repository. To republish, to post on servers, or to redistribute to lists, contact the Author. These files are offered as-is, without any warranty.

september 2023

version 0.5

giovanni.squillero@polito.it

Introduction to Python

2



Why Python?

- High-level programming language, truly portable
- Actively developed, open-source and community-driven
- Batteries included, huge code base
- Steep learning curve, easy to learn and to use
- Powerful as a scripting language
- Support both programming in the large and in the small
- Can be used interactively
- De-facto standard in several domain (e.g., data science)

Why Yet Another Introduction?

- No-nonsense introduction to Python for programmers (in any C-like language)
- Only three (full) days
- Not quite a *gentle introduction*
- Not the usual *Dummy's guide to...*
- And no reference to Objects (although OO in Python is beautiful)



Material

- Official online documentation
 - <https://docs.python.org/3/>
 - <https://www.python.org/dev/peps/>
- Spare online resources
 - <https://stackoverflow.com/questions/tagged/python>
 - <https://www.google.com/search?q=python>
 - <https://pythontutor.com>
 - <https://godbolt.org>
- Course repo & similar stuff
 - <https://github.com/squillero/python-lectures>

Introduction to Python

Set-up



Working with Python

- Download the latest official Python
 - <https://www.python.org/downloads/>
- Use the standard package manager (pip) to get additional modules

Python “Vibrant” Ecosystem



giovanni.squillero@polito.it

Introduction to Python

9

Working with Python

- Download the latest official Python
 - <https://www.python.org/downloads/>
- Set-up a virtual environment

```
python -m venv directory
source ./directory/bin/activate
```

- Use the standard package manager (pip) to get additional modules

giovanni.squillero@polito.it

Introduction to Python

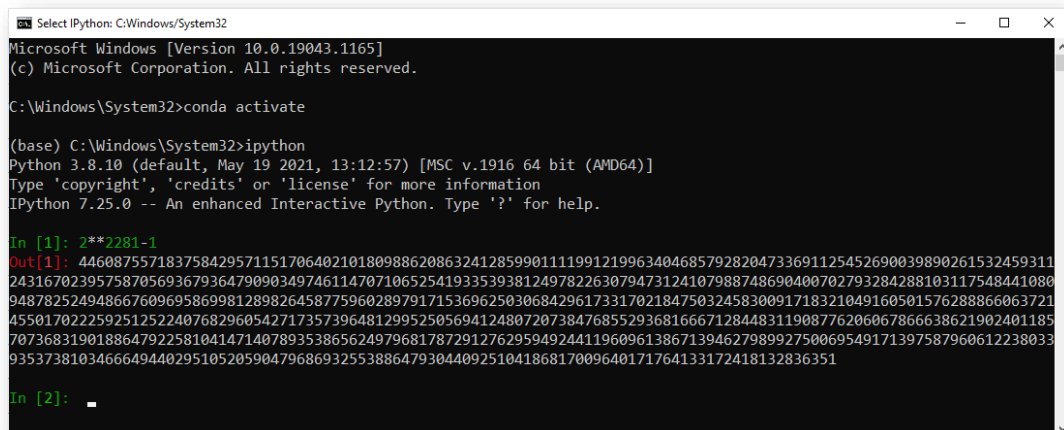
10

Dependency Managers (2023)

- Poetry (<https://python-poetry.org/>)
 - Great, but using PyTorch can be tricky
- Anaconda (<https://www.anaconda.com/>)
 - Slightly bloated and not really “free”
- Virtualenv (<https://virtualenv.pypa.io/>)
 - Based on venv

Getting Python

- Let's test Python calculating the 17th Mersenne prime



```

Select IPython: C:\Windows\System32
Microsoft Windows [Version 10.0.19043.1165]
(c) Microsoft Corporation. All rights reserved.

C:\Windows\System32>conda activate

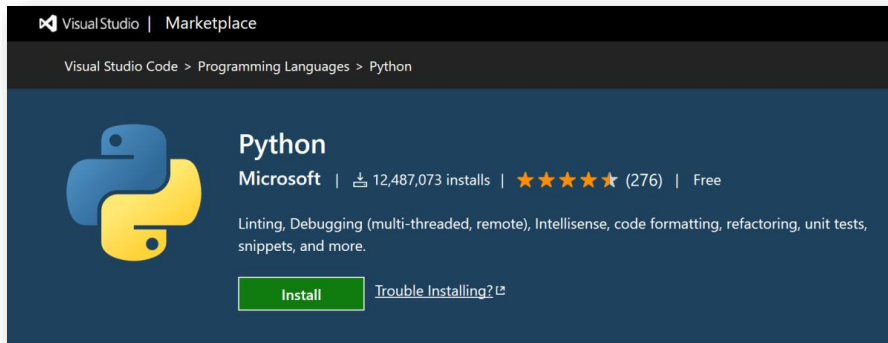
(base) C:\Windows\System32>ipython
Python 3.8.10 (default, May 19 2021, 13:12:57) [MSC v.1916 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.25.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: 2**2281-1
Out[1]: 4460875571837584295711517064021018098862086324128599011119912199634046857928204733691125452690039890261532459311
243167023957587056936793647909034974611470710652541933539381249782263079473124107988748690400702793284288103117548441080
948782524948667609695869981289826458775960289791715369625030684296173317021847503245830091718321049160501576288866063721
455017022259251252240768296054271735739648129952505694124807207384768552936816667128448311908776206067866638621902401185
707368319018864792258104147140789353865624979681787291276295949244119609613867139462798992750069549171397587960612238033
93537381034666494402951052059047968693255388647930440925104186817009640171764133172418132836351

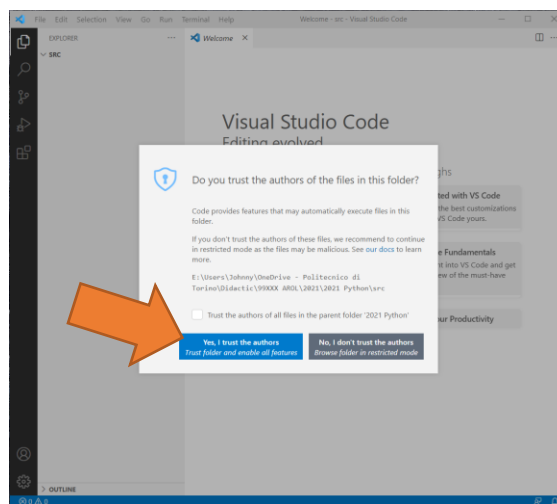
In [2]: -
  
```

IDE

- Install Visual Studio Code and the Python Extension



Open a "directory"



Set interpreter & Press play

The first screenshot shows the 'Select interpreter' dialog box in Visual Studio Code. The 'Python 3.8.10 64-bit (tools: virtualenv)' option is selected. An orange arrow points to the 'Python 3.8.10 64-bit (tools: virtualenv)' option, and another orange arrow points to the 'Run and Debug' icon in the bottom left corner of the editor.

The second screenshot shows the same editor with the file 'test.py' open. The code contains a single line: `print("Hello Python!")`. An orange arrow points to the 'Run and Debug' icon in the bottom right corner of the editor. The terminal window at the bottom shows the output: `Hello Python!`.

giovanni.squillero@polito.it

Kick off

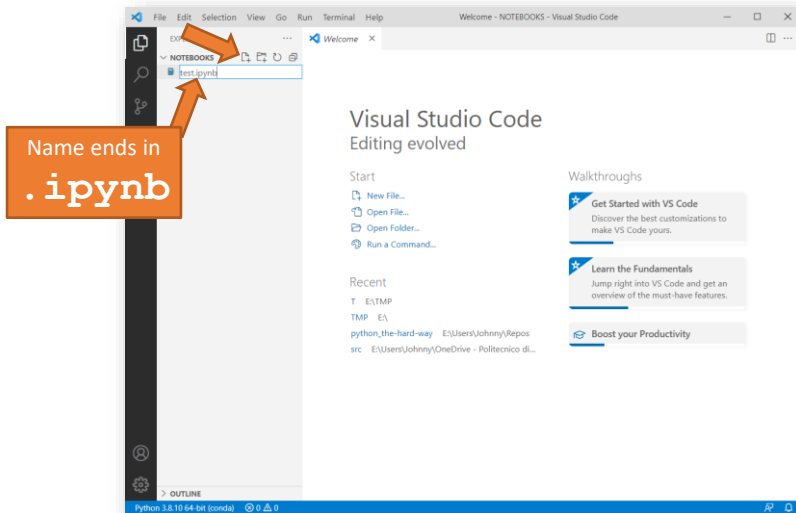


```
a = 23
if a == 23:
    print("Whoa!")
    a = a / 2
print(a)
```

The screenshot shows Visual Studio Code in debug mode. The code from the previous slide is loaded. The 'Run and Debug' icon in the bottom right corner is highlighted with an orange arrow. The 'Variables' pane on the left shows the current state of the program: `a: 23`. The terminal window at the bottom shows the output: `Whoa!`.

giovanni.squillero@polito.it

Jupyter Notebook

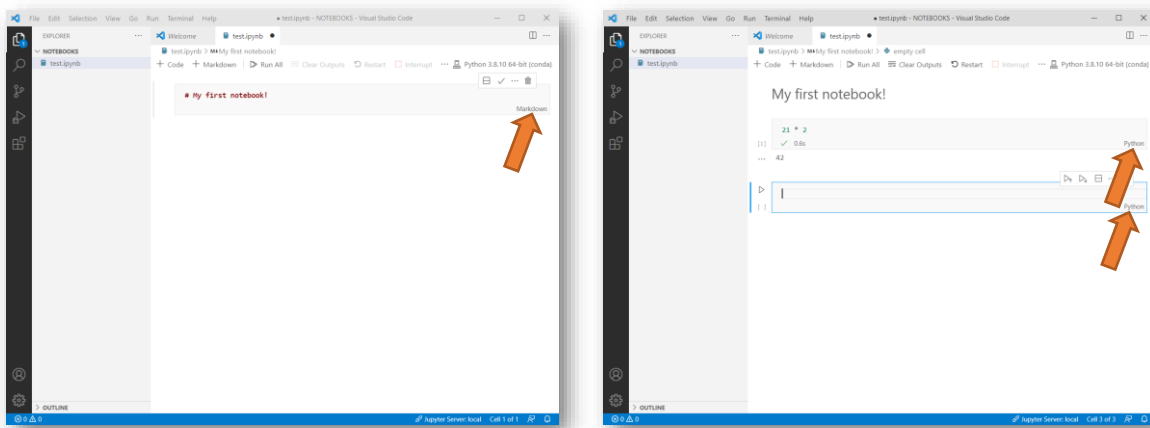


giovanni.squillero@polito.it

Introduction to Python

17

My first notebook



giovanni.squillero@polito.it

Introduction to Python

18

Execution order...

```

[1] foo = 42
    ✓ 0.4s Python

[4] foo += 1
    bar = 23
    ✓ 0.3s Python

[3] bar += 10
    ✓ 0.2s Python

[5] print(foo, bar)
    ✓ 0.3s Python
... 44 23

```

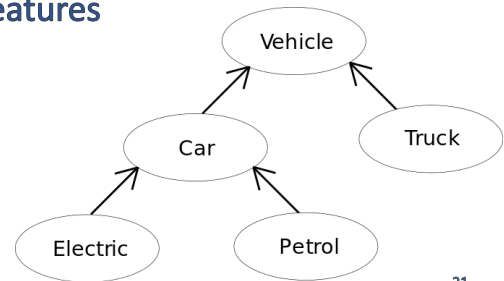
Data Model

- Python is a **strongly-typed, dynamic, object-oriented** language
- Every object has a **type** and a **value**
- Objects are Python's **abstraction for data**
 - Side note: code is also represented by objects
- Every object has an **identity**
 - Python Objects are forever!
 - The identity never changes once it has been created — **is, id()**
 - ... but the value of a mutable object may vary

Object Oriented Paradigm (in 1 slide)

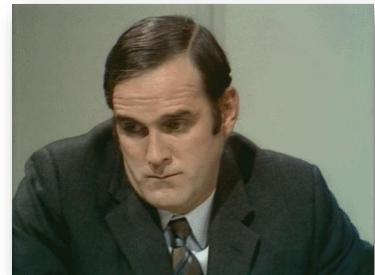
- An **object** contains both **data** and **code**
- An objects is the instance of a **class** (class ↔ type)
- Subclass hierarchy
 - A class **inherits** the structure from its parent(s)
 - The child class may **add** or **specialize** features
 - **isinstance(x, Car)** is **True** if **x** is a **Petrol**
- **Polymorphism**
 - The caller may safely ignore the exact class in the hierarchy it is operating on

Button
- xsize
- ysize
- label_text
- interested_listeners
- xposition
- yposition
+ draw()
+ press()
+ register_callback()
+ unregister_callback()




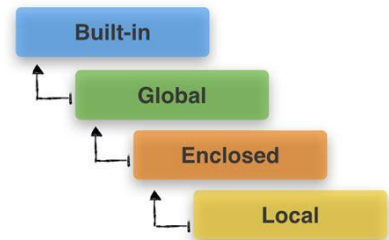
Variables?

- Objects
- Names
- Variables (i.e., objects with a name)



Naming & Binding

- A variable is an object with a name
- Names refer to objects
-  `foo = 42`
`foo` is a name not a variable
- The scope defines the visibility of a name
- Whenever a name is used, it is resolved using the nearest enclosing scope



Naming & Binding

A = 



Naming & Binding

A = 

B = 



giovanni.squillero@polito.it

Introduction to Python

25

Naming & Binding

A = 

B = 

C = **A**



giovanni.squillero@polito.it

Introduction to Python

26

Naming & Binding

drink (A)

drink (B)

drink (C)



giovanni.squillero@polito.it

Introduction to Python

27

Naming & Binding

A = 

B = 

C = A

A = 



giovanni.squillero@polito.it

Introduction to Python

28

Naming & Binding

A = 

B = **A**

C = 

drink (**A**)

giovanni.squillero@polito.it

Introduction to Python



Alias vs. Copy

A = 

B = **A**

C = **copy** (**A**)

drink (**A**)

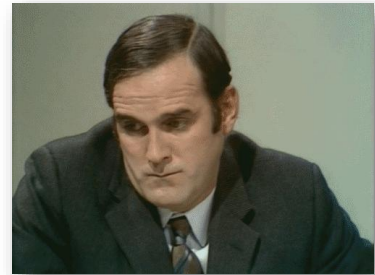
giovanni.squillero@polito.it

Introduction to Python



Immutable vs. Mutable

- **Immutable**
 - E.g., numbers
- **Mutable**
 - Some containers
 - Some compound objects
 - Some complex objects



Immutable vs. Mutable

- The bag itself (object **identity**) did not change
- The bag content (object **value**) did change



Operations

`foo = 42`

Binds the name *foo* to the object **42** (an integer)

`foo = foo - 35`

Creates a the new object **7**,
rebinds the name *foo* to it

<https://docs.python.org/>

Introduction to Python

Data Types



Standard Type Hierarchy

- Number
 - Integral
 - Integers (**int**)
 - Booleans (**bool**)
 - Real (**float**)
 - Complex (**complex**)
 - *Fancier*
 - Fractions (**fractions.Fraction**)
 - Real numbers with user-defined precision (**decimal.Decimal**)
- Caveat:
 - Numbers are **immutable**

```
foo = 42
bar = True
baz = 4.2
qux = 4+2j
```

Standard Type Hierarchy

- Sequences
 - Immutable
 - String (**str**)
 - Tuples (**tuple**)
 - Bytes (**bytes**)
 - Mutable
 - Lists (**list**)
 - Byte Arrays (**bytearray**)

```
foo = "42"
bar = (4, 2)
baz = bytearray([0x04, 0x02])
qux = [4, 2]
tud = b'\x04\x02'
```

Standard Type Hierarchy

- Set Types
 - Sets (**set**)
 - Frozen Sets (**frozenset**)
- Caveat:
 - Sets are **mutable**, frozen sets are **immutable**

```
foo = {4, 2}
bar = frozenset({4, 2})
```

Standard Type Hierarchy

- Mappings
 - Dictionaries (**dict**)

```
foo = {'Giovanni':23, 'Paola':18}
```

Standard Type Hierarchy

- None (**NoneType**)

```
foo = None
```

Standard Type Hierarchy

- NotImplemented
- Ellipsis (...)
- Callable types
- Modules
- Custom classes
- Class instances
- I/O objects
- Internal types

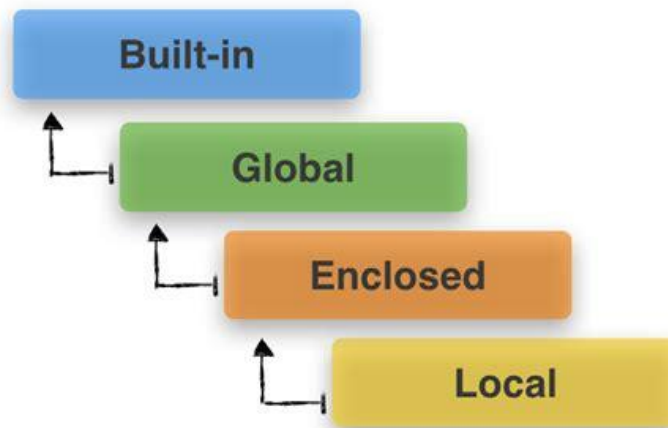


Introduction to Python

Basic Syntax



LEGB!



Style (TL;DR)

- `module_name`
- `package_name`
- `ClassName`
- `method_name`
- `ExceptionName`
- `function_name`
- `GLOBAL_CONSTANT_NAME`
- `global_var_name`
- `instance_var_name`
- `function_parameter_name`
- `local_var_name`



<https://github.com/squillero/style>

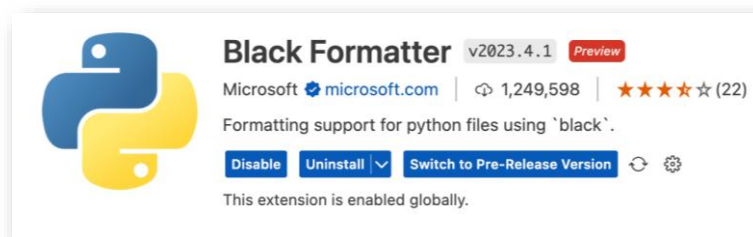
Style

- Source file is UTF-8, all Unicode runes can be used
- Single underscore is a valid name (`_`)
- Safe rule:
 - Use only printable standard ASCII characters for names
 - Don't start names with single/double underscore unless you really know what you are doing
 - Append an underscore not to clash with keywords or common names, e.g., `int_` and `list_`

```
無 = 0
_ = 42
```

Style

- Use an automatic formatter
 - Suggested: `black`



Comments

- Comments starts with hash (#) and ends with the line
- (Multi-line) (doc)strings might be used to comment/document the code in specific contexts

```
# This is a comment

"""
This is a multi-line docstring,
that may also help documenting the code
"""
```

Basic I/O: print

- Type `print(` in Visual Studio Code and wait for help

```
print()
```

(***values: object**, sep: str | None = ..., end: str | None = ..., file: SupportsWrite[str] | None = ..., flush: bool = ...) -> None

print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default. Optional keyword arguments:

file: a file-like object (stream); defaults to the current sys.stdout.

sep: string inserted between values, default a space.

end: string appended after the last value, default a newline.

flush: whether to forcibly flush the stream.

Pythonic Approach

- Functions are simple and straightforward
- Specific “named arguments” can be optionally used to tweak the behavior

```
print("Foo", "Bar")  
print("Foo", "Bar", file=sys.stderr, sep='|')
```

Basic I/O: `input`

- Type `input` (in Visual Studio Code and wait for help

```
input()  
(__prompt: Any = ...) -> str
```

Read a string from standard input. The trailing newline is stripped.

The prompt string, if given, is printed to standard output without a trailing newline before reading input.

If the user hits EOF (*nix: Ctrl-D, Windows: Ctrl-Z+Return), raise EOFError. On *nix systems, readline is used if available.

Conditional Execution

- Generic form

– **if** [**elif** [... **elif**]] [**else**]

```
if answer == "yes" or answer == "YES":
    print("User was positive")
elif answer == "no" or answer == "NO":
    print("User was negative")
else:
    print("Dunno!?")
```

- Caveats

- C-Like relational operators: **==** **!=** **<** **<=** **>** **>=**
- Human-readable logic operators: **not** **and** **or**
- Numeric intervals: **if 10 <= foo < 42:**
- Special operators: **is in**
- Truth Value Testing: **if name:**

While loop

- Vanilla, C-like **while**

```
foo = 117
while foo > 0:
    print(foo)
    foo //= 2
```

Indentation

- Python uses indentation for defining **{ blocks }**
- Both tabs and spaces are allowed
 - consistency is required, let alone desirable

```

1
2 for item in range(10):
3     print('I')
4     print('am')
5     print('a')
6     if item % 2 == 0:
7         print('funny')
8         print('and')
9         print('silly')
10    else:
11        print('dull')
12        print('and')
13        print('serious')
14    print('block')
15    print('used')
16    print('as')
17    print('example.')
18
19
20
21

```

Modules

- Python code libraries are organized in modules
- Names in modules can be imported in several way

```

import math
print(math.sqrt(2))
✓ 0.4s Python
1.4142135623730951

```

```

from math import sqrt
from cmath import sqrt as c_sqrt
print(sqrt(2), c_sqrt(-2))
✓ 0.4s Python
1.4142135623730951 1.4142135623730951j

```

Constructors

- Standard object constructors can be used to create empty/default objects, or to convert between types

```
foo = int()           # the default value for numbers is 0
bar = float("4.2")
baz = str(42)
```

Numbers

- Standard mathematical operators:

+ **-** ***** **/** **%** **//** ******

- Caveats:

/ always returns a floating point

// always returns an integer — although not always of class **int**

Mod (**%**) always returns a result — even with a negative or float

i Operators

- In-place operators (**iadd**, **isub**, ...):

+= -= *= /= %= //=

- If the object is mutable:

changes the object instead of creating a new one

- May be more efficient
- Mind the side effects!

- If the object is immutable:

foo ★= bar is equivalent to **foo = foo ★ bar**

Numeric operations

Operation	Result	Notes
<code>x + y</code>	sum of <code>x</code> and <code>y</code>	
<code>x - y</code>	difference of <code>x</code> and <code>y</code>	
<code>x * y</code>	product of <code>x</code> and <code>y</code>	
<code>x / y</code>	quotient of <code>x</code> and <code>y</code>	
<code>x // y</code>	floored quotient of <code>x</code> and <code>y</code>	(1)
<code>x % y</code>	remainder of <code>x / y</code>	(2)
<code>-x</code>	<code>x</code> negated	
<code>+x</code>	<code>x</code> unchanged	
<code>abs(x)</code>	absolute value or magnitude of <code>x</code>	
<code>int(x)</code>	<code>x</code> converted to integer	(3)(6)
<code>float(x)</code>	<code>x</code> converted to floating point	(4)(6)
<code>complex(re, im)</code>	a complex number with real part <code>re</code> , imaginary part <code>im</code> . <code>im</code> defaults to zero.	(6)
<code>c.conjugate()</code>	conjugate of the complex number <code>c</code>	
<code>divmod(x, y)</code>	the pair <code>(x // y, x % y)</code>	(2)
<code>pow(x, y)</code>	<code>x</code> to the power <code>y</code>	(5)
<code>x ** y</code>	<code>x</code> to the power <code>y</code>	(5)

Real-number operations

Operation	Result
<code>math.trunc(x)</code>	<code>x</code> truncated to <code>Integral</code>
<code>round(x[, n])</code>	<code>x</code> rounded to <code>n</code> digits, rounding half to even. If <code>n</code> is omitted, it defaults to 0.
<code>math.floor(x)</code>	the greatest <code>Integral</code> $\leq x$
<code>math.ceil(x)</code>	the least <code>Integral</code> $\geq x$

Bitwise operations

Operation	Result	Notes
<code>x y</code>	bitwise <i>or</i> of <code>x</code> and <code>y</code>	(4)
<code>x ^ y</code>	bitwise <i>exclusive or</i> of <code>x</code> and <code>y</code>	(4)
<code>x & y</code>	bitwise <i>and</i> of <code>x</code> and <code>y</code>	(4)
<code>x << n</code>	<code>x</code> shifted left by <code>n</code> bits	(1)(2)
<code>x >> n</code>	<code>x</code> shifted right by <code>n</code> bits	(1)(3)
<code>~x</code>	the bits of <code>x</code> inverted	

Sequences

- Ordered data structure
 - Support positional access
 - Are iterable
- Among the different sequences, the most popular are
 - Lists: mutable, heterogenous
 - Tuples: immutable, heterogenous
 - Strings: immutable, homogenous

Sequence operations

Operation	Result
<code>x in s</code>	True if an item of <code>s</code> is equal to <code>x</code> , else False
<code>x not in s</code>	False if an item of <code>s</code> is equal to <code>x</code> , else True
<code>s + t</code>	the concatenation of <code>s</code> and <code>t</code>
<code>s * n</code> or <code>n * s</code>	equivalent to adding <code>s</code> to itself <code>n</code> times
<code>s[i]</code>	<code>i</code> th item of <code>s</code> , origin 0
<code>s[i:j]</code>	slice of <code>s</code> from <code>i</code> to <code>j</code>
<code>s[i:j:k]</code>	slice of <code>s</code> from <code>i</code> to <code>j</code> with step <code>k</code>
<code>len(s)</code>	length of <code>s</code>
<code>min(s)</code>	smallest item of <code>s</code>
<code>max(s)</code>	largest item of <code>s</code>
<code>s.index(x[, i[, j]])</code>	index of the first occurrence of <code>x</code> in <code>s</code> (at or after index <code>i</code> and before index <code>j</code>)
<code>s.count(x)</code>	total number of occurrences of <code>x</code> in <code>s</code>

Looping over sequences

```
giovanni = "Giovanni Adolfo Pietro Pio Squillero"

for letter in giovanni:
    print(letter)

for index in range(len(giovanni)):
    print(index, giovanni[index])

for index, letter in enumerate(giovanni):
    print(index, letter)
```

- **Caveat: `range()` is a class designed to efficiently generate indexes; the full constructor is:**
`class range(start, stop[, step])`

Looping over multiple sequences

```
for a, b in zip('GIOVANNI', 'XYZ'):
    print(f"{a}:{b}")
```

✓ 0.3s

Python

G:X

I:Y

O:Z

Non-structured Statements

- **continue** and **break**
- **else** with **while** and **for**

```
foo = 0
bar = int(input("Start @ "))
baz = int(input("Break @ "))
while foo < 20:
    foo += 1
    if foo < bar:
        continue
    if foo == baz:
        break
    print(foo)
else:
    print("Natural end of the loop!")
```

Lists and Tuples

- A list is a heterogenous, mutable sequence
- A tuple is a heterogenous, immutable sequence
- A list may contain tuples as elements, and vice versa
- Only lists may be sorted, but all iterable may be accessed through **sorted()**

```
birthday = [("Giovanni", 23, 10), ("Paola", 18, 5)]
print(birthday[0])

birthday_alt = [("Giovanni", 23, 10), ("Paola", 18, 5)]
print(birthday_alt[1][2])

birthday_alt.sort()
print(birthday_alt)

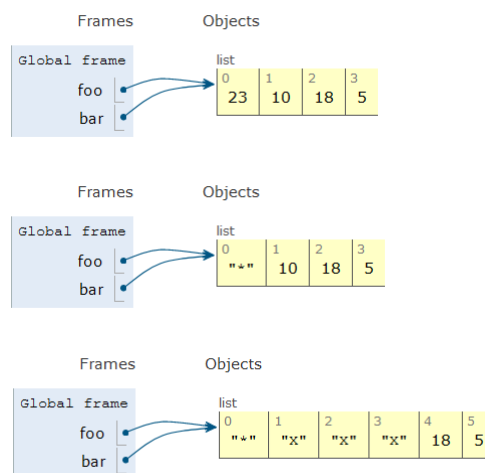
foo = (23, 10, 18, 5)
print(sorted(foo))
```

Sort vs. Sorted

- **sorted(foo)** returns a list containing all elements of foo sorted in ascending order
- **bar.sort()** modify bar, sorting its elements in ascending order
- Named options
 - **key**
 - **reverse** (Boolean)

Slices & Name binding

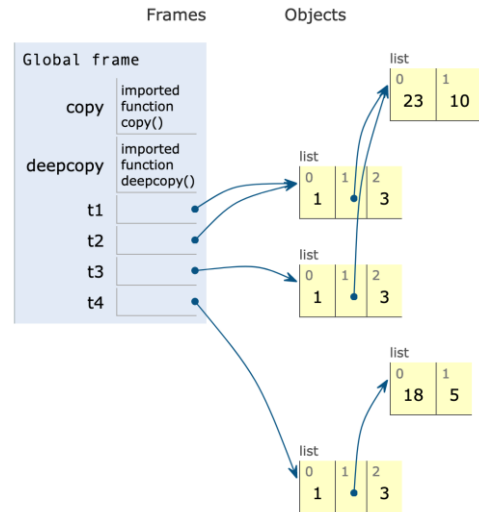
```
foo = [23, 10, 18, 5]
bar = foo
bar[0] = '*'
foo[1:2] = list("xxx")
```



Alias, copy, and deepcopy

```
t1 = [1, [23, 10], 3]
t2 = t1
t3 = copy(t1)
t4 = deepcopy(t1)
```

```
t4[1][0] = 18
t4[1][1] = 5
```



Mutable-sequence operations

Operation	Result
<code>s[i] = x</code>	item <i>i</i> of <i>s</i> is replaced by <i>x</i>
<code>s[i:j] = t</code>	slice of <i>s</i> from <i>i</i> to <i>j</i> is replaced by the contents of the iterable <i>t</i>
<code>del s[i:j]</code>	same as <code>s[i:j] = []</code>
<code>s[i:j:k] = t</code>	the elements of <code>s[i:j:k]</code> are replaced by those of <i>t</i>
<code>del s[i:j:k]</code>	removes the elements of <code>s[i:j:k]</code> from the list
<code>s.append(x)</code>	appends <i>x</i> to the end of the sequence (same as <code>s[len(s):len(s)] = [x]</code>)
<code>s.clear()</code>	removes all items from <i>s</i> (same as <code>del s[:]</code>)
<code>s.copy()</code>	creates a shallow copy of <i>s</i> (same as <code>s[:]</code>)
<code>s.extend(t)</code> or <code>s += t</code>	extends <i>s</i> with the contents of <i>t</i> (for the most part the same as <code>s[len(s):len(s)] = t</code>)
<code>s *= n</code>	updates <i>s</i> with its contents repeated <i>n</i> times
<code>s.insert(i, x)</code>	inserts <i>x</i> into <i>s</i> at the index given by <i>i</i> (same as <code>s[i:i] = [x]</code>)
<code>s.pop()</code> or <code>s.pop(i)</code>	retrieves the item at <i>i</i> and also removes it from <i>s</i>
<code>s.remove(x)</code>	remove the first item from <i>s</i> where <code>s[i]</code> is equal to <i>x</i>
<code>s.reverse()</code>	reverses the items of <i>s</i> in place

Conditions over Sequences

- Use **any()** or **all()** to check that some/all elements in a sequence evaluates to **True**

```
print(foo)
print(any(foo))
print(all(foo))
```

✓ 0.4s

Python

```
[False, False, False, False, False, True]
```

```
True
```

```
False
```

Strings

- Strings are immutable sequences of Unicode runes

```
string1 = "Hi!, I'm a \"string\""
string2 = 'Hi!, I\'m also a "string"'
```

```
beatles = """John Lennon
Paul McCartney
George Harrison
Ringo Starr"""
```

```
bts = '''RM
진
슈가
제이홉
지민
뷔
정국'''
```

Strings

- The complete list of functions is huge
- Official documentation
 - <https://docs.python.org/3/library/stdtypes.html#textseq>
 - <https://docs.python.org/3/library/stdtypes.html#string-methods>

String formatting

- Use f-strings!

```
discoverer = 'Leonhard Euler'
number = 2**31-1
year = 1772

print(f"Mersenne primes discovered by {discoverer} in {year}: {number:,"} +
      f" {len(str(number))} digits.")
```

✓ 0.3s Python

Mersenne primes discovered by Leonhard Euler in 1772: 2,147,483,647 (10 digits).

- More info:
 - https://docs.python.org/3/reference/lexical_analysis.html#f-strings
 - <https://docs.python.org/3/library/string.html#formatspec>

Notable `str` methods

```

"Giovanni Adolfo Pietro Pio".split()
[11] ✓ 0.2s Python
... ['Giovanni', 'Adolfo', 'Pietro', 'Pio']

str.split("Giovanni Adolfo Pietro Pio")
[12] ✓ 0.3s Python
... ['Giovanni', 'Adolfo', 'Pietro', 'Pio']

```

- **Caveat:**

"bar".foo() vs. **str.foo("bar")**

More notable `str` methods

```

"Giovanni" + " Whoa!" * 3
[6] ✓ 0.4s Python
... 'Giovanni Whoa! Whoa! Whoa!'

"|".join(list('Giovanni'))
[8] ✓ 0.3s Python
... 'G|i|o|v|a|n|n|i'

'Pio' in 'Giovanni Adolfo Pietro Pio Squillero'
[9] ✓ 0.3s Python
... True

```

All `str` methods

```

capitalize() casefold() center(width[, fillchar])
count(sub[, start[, end]])
encode(encoding="utf-8", errors="strict")
endswith(suffix[, start[, end]]) expandtabs(tabsize=8)
find(sub[, start[, end]]) format(*args, **kwargs)
format_map(mapping) index(sub[, start[, end]]) isalnum()
isalpha() isascii() isdecimal() isdigit() isidentifier()
islower() isnumeric() isprintable() isspace() istitle()
isupper() join(iterable) ljust(width[, fillchar]) lower()
lstrip([chars]) partition(sep) removeprefix(prefix, /)
removesuffix(suffix, /) replace(old, new[, count])
rfind(sub[, start[, end]]) rindex(sub[, start[, end]])
rjust(width[, fillchar]) rpartition(sep)
rsplit(sep=None, maxsplit=-1) rstrip([chars])
split(sep=None, maxsplit=-1) splitlines([keepends])
startswith(prefix[, start[, end]]) strip([chars]) swapcase()
title() translate(table) upper() zfill(width)

```

Dictionary

- Heterogeneous associative array
 - Keys are required to be *hashable*, thus immutable
- Syntax similar to sequences, but no positional access

```

stone = dict()
stone['23d1364a'] = 'Mick'
stone['5465ba78'] = 'Brian'
stone['06cc49dd'] = 'Ian'
stone['c2f65729'] = 'Keith'
stone['713c0a4e'] = 'Ronnie'
stone['3ed50ef3'] = 'Charlie'

print(stone['3ed50ef3'])

```

Dictionary Keys and Values

```

stone.keys()
✓ 0.3s Python
dict_keys(['23d1364a', '5465ba78', '06cc49dd', 'c2f65729', '713c0a4e', '3ed50ef3'])

stone.values()
✓ 0.6s Python
dict_values(['Mick', 'Brian', 'Ian', 'Keith', 'Ronnie', 'Charlie'])

stone.items()
✓ 0.3s Python
dict_items([('23d1364a', 'Mick'), ('5465ba78', 'Brian'), ('06cc49dd', 'Ian'),
('c2f65729', 'Keith'), ('713c0a4e', 'Ronnie'), ('3ed50ef3', 'Charlie')])

```

For loops and Dictionaries

- When casted to a list or to an iterator, a dictionary is the sequence of its keys (preserving the insertion order)

```

for s in stone.keys():
|   print(f"{s} -> {stone[s]}")

for s in stone:
|   print(f"{s} -> {stone[s]}")

for k, v in stone.items():
|   print(f"{k} -> {v}")

```

Sets

- Sets can be seen as dictionaries without values
- When casted to a list or to an iterator, a set is the sequence of its elements (not preserving the insertion order)
- The standard set operations can be used: **add**, **remove**, **in**, **not in**, **<= (issubset)**, **<**, **- (difference)**, **&**, **|**, **^ (symmetric_difference)**, ...

```
foo = set("MAMMA")
bar = set("MIA")
print(foo | bar)

✓ 0.5s Python
{'M', 'A', 'I'}
```

Copy and Delete

- **del**: delete things
 - A whole object: **del foo**
 - An element in a list: **del foo[1]**
 - An element in a dictionary: **del foo['key']**
 - An element in a set: **del foo['item']**
- **copy**: Shallow copy an object (list, dictionary, set, ...)
 - Example: **foo = bar.copy()**
 - More Pythonic alternatives may exist, e.g.: **foo = bar[:]**

pass

- The **pass** statement does nothing
- It can be used when a statement is required syntactically but the program requires no action

Introduction to Python

Functions



Functions

- Keyword **def**

```
def countdown(x):
    if not isinstance(x, int) or x <= 0:
        return False
    while x > 0:
        x //= 2
        print(x)
    return True

print(countdown("10"))
print(countdown(10))
```

- Caveats:
 - Names vs. Variables

More Caveats

- Functions return exactly one argument
 - No **return** statement is equivalent to a **return None**
 - Magic packing/unpacking is used for returning multiple arguments, i.e., **return 23, 10** is equivalent to **return (23, 10)**



Even More caveats

- Functions are first-class citizen
- Function names are just “names”
- Remember scope!



```
def foo():
    print("foo")

if input() == "crazy":
    def foo():
        print("crazy")

foo()
```

```
foo = input()
def bar():
    print(f"bar:{foo}")

bar()
foo = "Giovanni!"
bar()
```

Docstrings

- A docstring is a string literal that occurs as the first statement in a function (or module, or class, or method) definition
 - It is shown by most IDEs for helping coders
 - It becomes the `__doc__` special attribute of that object

```
def foo(x):
    """My FOO function"""
    pass

(x) -> None
My FOO function
foo()
```

Keyword and Default Arguments

- Remember scope and name binding
- Arguments may be optional if a default is provided
- Keyword arguments can be in any order

```
foo = 1
bar = 2
baz = 3

def silly_function(bar, baz=99):
    print(foo, bar, baz)

silly_function(23)
silly_function(23, baz=10)
silly_function(baz=10, bar=23)
```

Positional vs. Keyword Arguments

- Arguments preceding the `/` are positional-only and cannot be specified to using keywords
- Arguments following the `*` must be specified using keywords
- Arguments between `/` and `*` might be either positional or keyword

```
def foo(x, y, /, foo=1, bar=2, *, baz=3):
    print(f"POSITIONAL: x={x}; y={y}")
    print(f"foo={foo}; bar={bar}")
    print(f"KEYWORD ONLY: baz={baz}")
```

✓ 0.3s

Python

*args and **kwargs

- When a formal parameter is in the form ***name** it receives a tuple with all the remaining positional arguments

```
def foo(x, y, *args):
    print(f"POSITIONAL: x={x}; y={y}")
    print(f"type(args) = {args}")

foo(23, 10, 18, 5)
✓ 0.2s Python
```

POSITIONAL: x=23; y=10
<class 'tuple'> = (18, 5)

*args and **kwargs

- When a formal parameter is in the form ****name** it receives a dictionary with all the remaining keyword arguments
- The argument ****name** must follow ***name**

```
def foo(x, y, *args, foo, **kwargs):
    print(f"args: {type(args)} = {args}")
    print(f"kwargs: {type(kwargs)} = {kwargs}")

foo(23, 10, 18, 5, foo='foo', bar='bar', baz='baz')
✓ 0.2s Python
```

args: <class 'tuple'> = (18, 5)
kwargs: <class 'dict'> = {'bar': 'bar', 'baz': 'baz'}

True Pythonic Scripts

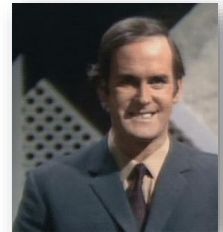
- Define all global 'constants' first, then functions
- Test `__name__`

```
GLOBAL_CONSTANT = 42

def foo():
    pass

def main():
    print(foo)
    pass

if __name__ == '__main__':
    # parse command line
    # setup logging
    main()
```



Callable Objects

- Names can refer to functions (*callable* objects)
- Functions are first class citizen

```
def foo(bar):
    print(f"foo{bar}")

qux = foo
qux('123')
```

✓ 0.3s Python

foo123

Lambda Keyword

- Lambda expressions are short (1 line), usually simple, and anonymous functions. They can be used to calculate values

```
foo = lambda x: 2**x
foo(10)
✓ 0.3s Python
1024
```

- or perform simple tasks

```
foo = lambda x: print(f"foo{str(x)}")
foo(10)
✓ 0.3s Python
foo10
```

Lambda Keyword

- Lambda expressions are quite useful to define simple, scratch functions to be used as argument in behavioral parametrization, e.g., as **key** for the **sort()** function

```
sorted_keys = sorted(my_dict, key=lambda k: my_dict[k])
```

Closures and Scope

- Consider how names are resolved

```
foo = 42
func = lambda x: foo + x
print(func(10))
foo = 1_000
print(func(10))
```

✓ 0.3s Python

52
1010



Closures and Scope

- Consider how names are resolved

```
def make_inc(number):
    return lambda x: number + x

i10 = make_inc(10)
i500 = make_inc(500)

print(i10(1), i500(1))
```

✓ 0.4s Python

11 501



Introduction to Python

List Comprehensions & Generators



List Comprehensions

- A concise way to create lists

```
[x for x in range(10)]
```

✓ 0.4s

Python

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
[x for x in range(50) if x % 3 == 0]
```

✓ 0.3s

Python

```
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48]
```

```
[(x, x**y) for x in range(2, 4) for y in range(4, 7)]
```

✓ 0.4s

Python

```
[(2, 16), (2, 32), (2, 64), (3, 81), (3, 243), (3, 729)]
```

Generators

- Like list comprehension, but elements are not actually calculated unless explicitly required by **next()**

```
foo = (x**x for x in range(100_000))
foo
✓ 0.3s Python
<generator object <genexpr> at 0x0000026D42A1C0B0>

for x in range(3):
| print(f"{next(foo):,}")
✓ 0.3s Python

1
1
4

for x in range(3):
| print(f"{next(foo):,}")
✓ 0.4s Python

27
256
3, Introduction to Python 103
```

giovanni.squillero@polito.it

Generators

- Can be quite effective inside **any()** or **all()**

```
any([n % 23 == 0 for n in range(1, 1_000_000_000)])
✓ 63.8s Python
True

any(n % 23 == 0 for n in range(1, 100_000_000))
✓ 0.3s Python
True
```

giovanni.squillero@polito.it

Introduction to Python

104

Generators

- Can be used to create lists and sets

```
numbers = set(a*b for a in range(11) for b in range(11))
print("All integral numbers that can be expressed as a*b with a and b less than 10: " +
      '|'.join(str(_) for _ in sorted(numbers)))
```

✓ 0.3s

Python

```
All integral numbers that can be expressed as a*b with a and b less than 10: 0 1 2 3 4 5
6 7 8 9 10 12 14 15 16 18 20 21 24 25 27 28 30 32 35 36 40 42 45 48 49 50 54 56 60 63 64
70 72 80 81 90 100
```

Introduction to Python

Modules



Modules

- Python code libraries are organized in modules
- Names in modules can be imported in several way

```
import math
print(math.sqrt(2))
✓ 0.4s Python
1.4142135623730951
```

```
from math import sqrt
from cmath import sqrt as c_sqrt
print(sqrt(2), c_sqrt(-2))
✓ 0.4s Python
1.4142135623730951 1.4142135623730951j
```

Notable Modules

```
import logging

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s %(levelname)s: %(message)s',
    datefmt='%H:%M:%S',
)

logging.debug("For debugging purpose")
logging.info("Verbose information")
logging.warning("Watch out, it looks important")
logging.error("We have a problem")
logging.critical("Armageddon was yesterday, now we have a real problem...")
✓ 0.8s Python
```

```
[12:07:40] INFO: Verbose information
[12:07:40] WARNING: Watch out, it looks important
[12:07:40] ERROR: We have a problem
[12:07:40] CRITICAL: Armageddon was yesterday, now we have a real problem...
```

Notable Modules

- Warnings that alert the user of some important condition that doesn't warrant raising an exception and terminating the program (e.g., uses of a obsolete feature)
- Caveat: **logging** vs. **warnings**

A screenshot of a Python IDE window showing the `warnings` module. The code in the editor is `import warnings` followed by `warnings.`. A dropdown menu is open, listing various attributes of the `warnings` module, with `showwarning` selected. The list includes: `catch_warnings`, `filters`, `filterwarnings`, `formatwarning`, `resetwarnings`, `simplefilter`, `warn`, `warn_explicit`, `WarningMessage`, `__annotations__`, and `__builtins__`.

giovanni.squillero@polito.it

Introduction to Python

109

Notable Modules

- Mathematical functions, use **cmath** for complex numbers

A screenshot of a Python IDE window showing the `math` module. The code in the editor is `import math` followed by `math.`. A dropdown menu is open, listing various mathematical functions from the `math` module, with `math.acos` selected. The list includes: `math.acosh`, `math.asin`, `math.asinh`, `math.atan`, `math.atan2`, `math.atanh`, `math.ceil`, `math.comb`, `math.copysign`, `math.cos`, and `math.cosh`.

giovanni.squillero@polito.it

Introduction to Python

110

Notable Modules

- Common string operations and constants

A screenshot of a Python IDE showing the completion menu for the `string` module. The code in the editor is `import string` followed by `string.` on the next line. The completion menu lists various attributes and constants of the `string` module, with `ascii_letters` selected. The list includes `ascii_letters`, `ascii_lowercase`, `ascii_uppercase`, `capwords`, `digits`, `Formatter`, `hexdigits`, `octdigits`, `printable`, `punctuation`, `Template`, and `whitespace`. The word "Python" is visible on the right side of the menu.

Notable Modules

- Various time-related functions

A screenshot of a Python IDE showing the completion menu for the `time` module. The code in the editor is `import time` followed by `time.` on the next line. The completion menu lists various functions and constants of the `time` module, with `time.altzone` selected. The list includes `time.altzone`, `time.asctime`, `time.ctime`, `time.daylight`, `time.get_clock_info`, `time.gmtime`, `time.localtime`, `time.mktime`, `time.monotonic`, `time.monotonic_ns`, `time.perf_counter`, and `time.perf_counter_ns`. The word "Python" is visible on the right side of the menu.

Notable Modules: **time**

- **perf_counter / perf_counter_ns**
 - Clock with the highest available resolution to measure a short duration, it does include time elapsed during sleep and is system-wide
 - Only the difference between the results of two calls is valid
- **process_time / process_time_ns**
 - Sum of the system and user CPU time of the current process. It does not include time elapsed during sleep
 - Only the difference between the results of two calls is valid

Notable Modules: **time**

- Measure performances:
 - Use **process_time** in a script
 - Use **%timeit** in a notebook

```
%timeit fibonacci_numbers = [n for n, _ in zip(fibonacci(), range(100))]
```

✓ 7.9s

Python

9.76 μs ± 8.35 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

Notable Modules: **time**

- **sleep**
 - Suspend execution of the calling thread for the given number of seconds.

Notable Modules

- Data pretty printer, sometimes a good replacement for **print**
 - Notez bien: tons of customizations importing the whole module

```
from pprint import pprint

numbers = [set(y+x**y for y in range(x)) for x in range(1, 10)]
pprint(numbers)
```

✓ 0.4s

Python

```
[[{1},
 {1, 3},
 {1, 11, 4},
 {1, 18, 67, 5},
 {128, 1, 6, 629, 27},
 {1, 7781, 38, 7, 1300, 219},
 {1, 2405, 8, 16812, 51, 117655, 346},
 {1, 66, 515, 4100, 32773, 262150, 2097159, 9},
 {4782976, 1, 6565, 43046729, 10, 59054, 83, 531447, 732}]
```

Notable Modules

- Miscellaneous operating system interfaces

```
import os

os.getcwd()
✓ 0.5s Python

'e:\\Users\\Johnny\\Repos\\python_the-hard-way'
```

os.

- abort
- access
- add_dll_directory
- altsep
- chdir
- chmod
- close
- closerange
- cpu_count
- curdir
- defpath
- device_encoding

giovanni.squillero@polito.it

Introduction to Python

117

Notable Modules

- System-specific parameters and functions

```
import sys

sys.
✓ 0. Python

sys.addaudithook
sys.api_version
sys.argv
sys.audit
sys.base_exec_prefix
sys.base_prefix
sys.breakpointhook
sys.builtin_module_names
sys.byteorder
sys.call_tracing
sys.callstats
sys.copyright
```

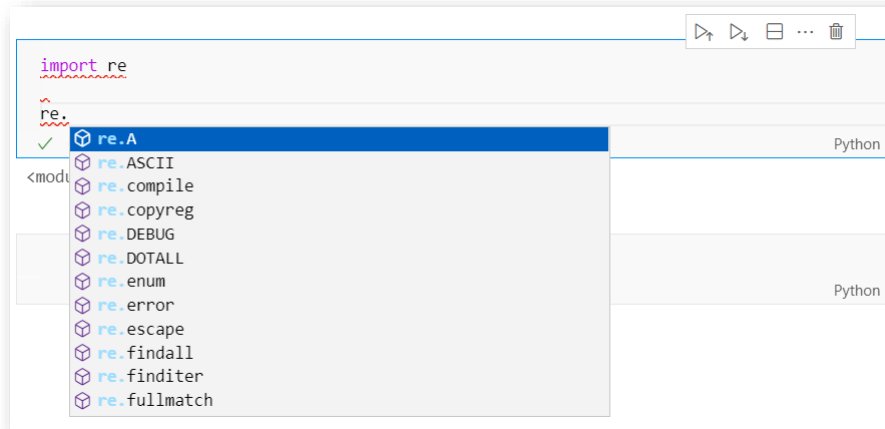
giovanni.squillero@polito.it

Introduction to Python

118

Notable Modules

- Regular expression operations



Notable Modules

- Real Python programmers do not love double loops
- Use **itertools** for efficient looping

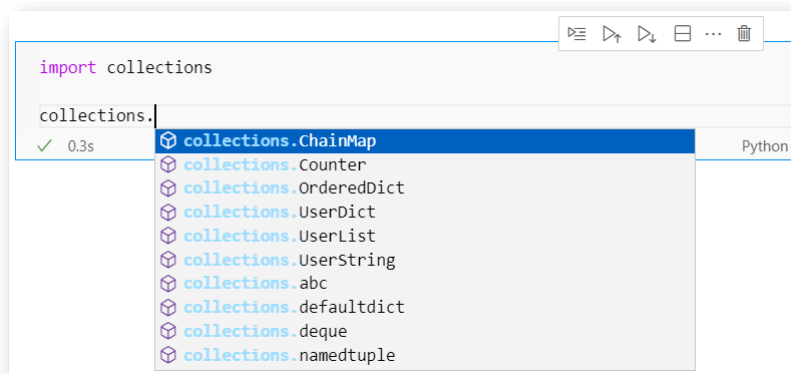
Examples	Results
<code>product('ABCD', repeat=2)</code>	AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD
<code>permutations('ABCD', 2)</code>	AB AC AD BA BC BD CA CB CD DA DB DC
<code>combinations('ABCD', 2)</code>	AB AC AD BC BD CD
<code>combinations_with_replacement('ABCD', 2)</code>	AA AB AC AD BB BC BD CC CD DD

More `itertools`

- Infinite loops
 - `count`, `cycle`, `repeat`
- Terminating on the shortest sequence
 - `accumulate`, `chain`, `chain.from_iterable`, `compress`, `dropwhile`, `filterfalse`, `groupby`, `islice`, `starmap`, `takewhile`, `tee`, `zip_longest`

Notable Modules

- The module `collections` contains specialized container datatypes providing alternatives to Python's general purpose built-in containers



```
import collections

collections.|
```

✓ 0.3s

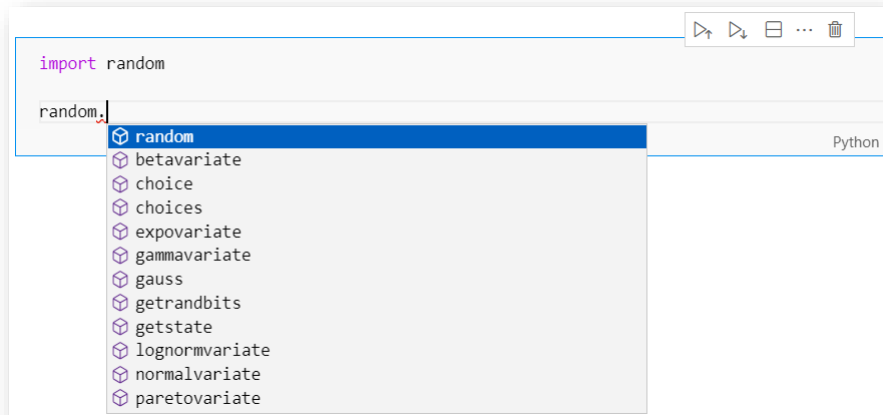
- `collections.ChainMap` Python
- `collections.Counter`
- `collections.OrderedDict`
- `collections.UserDict`
- `collections.UserList`
- `collections.UserString`
- `collections.abc`
- `collections.defaultdict`
- `collections.deque`
- `collections.namedtuple`

Notable Modules: **collections**

<code>namedtuple()</code>	factory function for creating tuple subclasses with named fields
<code>deque</code>	list-like container with fast appends and pops on either end
<code>ChainMap</code>	dict-like class for creating a single view of multiple mappings
<code>Counter</code>	dict subclass for counting hashable objects
<code>OrderedDict</code>	dict subclass that remembers the order entries were added
<code>defaultdict</code>	dict subclass that calls a factory function to supply missing values
<code>UserDict</code>	wrapper around dictionary objects for easier dict subclassing
<code>UserList</code>	wrapper around list objects for easier list subclassing
<code>UserString</code>	wrapper around string objects for easier string subclassing

Notable Modules

- Generate pseudo-random numbers



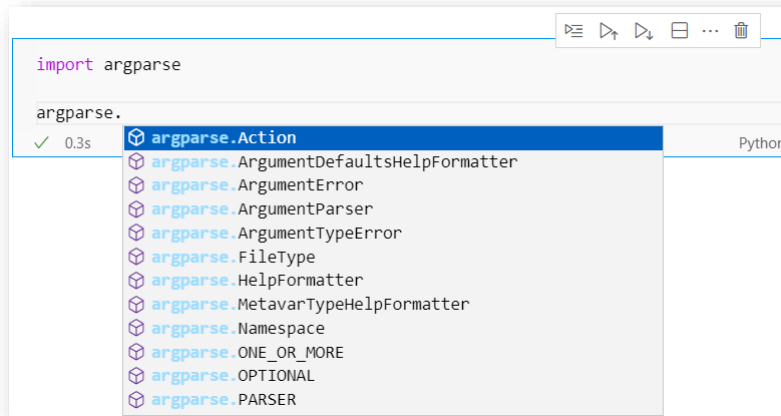
```
import random

random.
```

The screenshot shows a Python IDE window with a code editor. The code editor contains the text `import random` on the first line and `random.` on the second line. A dropdown menu is open below the second line, listing the following submodules: `random`, `betavariate`, `choice`, `choices`, `expovariate`, `gammavariate`, `gauss`, `getrandbits`, `getstate`, `lognormvariate`, `normalvariate`, and `paretovariate`. The `random` submodule is highlighted in blue. The window title bar shows standard navigation icons and the word "Python".

Notable Modules

- Parse command-line arguments



```
import argparse

argparse.
```

0.3s

- ✓ `argparse.Action`
- `argparse.ArgumentDefaultsHelpFormatter`
- `argparse.ArgumentError`
- `argparse.ArgumentParser`
- `argparse.ArgumentTypeError`
- `argparse.FileType`
- `argparse.HelpFormatter`
- `argparse.MetavarTypeHelpFormatter`
- `argparse.Namespace`
- `argparse.ONE_OR_MORE`
- `argparse.OPTIONAL`
- `argparse.PARSER`

Notable Modules: **argparse**

- Quite complex and powerful
- Help and recipes available from python.org

```
parser = argparse.ArgumentParser()
parser.add_argument('-v', '--verbose', action='count', default=0, help='increase log verbosity')
parser.add_argument('-d',
                    '--debug',
                    action='store_const',
                    dest='verbose',
                    const=2,
                    help='log debug messages (same as -vv)')
args = parser.parse_args()

if args.verbose == 0:
    logging.getLogger().setLevel(level=logging.WARNING)
elif args.verbose == 1:
    logging.getLogger().setLevel(level=logging.INFO)
elif args.verbose == 2:
    logging.getLogger().setLevel(level=logging.DEBUG)
```

User modules

- A Python file is a “module” and can be imported

```
file_a.py > ...
1  def foo(x):
2  |  print(f"FileA's foo({x})!")
```

```
file_b.py
1  import file_a
2
3  file_a.foo(23)
```

- When a file is imported, it is evaluated by the interpreter
 - All statements are executed
 - The `__name__` is set to the actual name of the file and not `“__main__”`

User modules

- A directory is a “module” and can be imported
- If the directory contains the file `__init__.py`, it is automatically read and evaluated by the interpreter
 - Other files may be imported using `from pkg import foo`

```
my_module > file_a.py > ...
1  def foo(x):
2  |  print(f"my_module's foo({x})!")
3
```

```
file_b.py
1  from my_module import file_a
2
3  file_a.foo(23)
```

- The files may also be imported writing appropriate `import` instructions in `__init__.py`

User modules

- Several alternatives, with obscure, yet important differences

```

import foo                # foo.py is a file
import mymod              # mymod is a directory containing __init__.py
from mymod import bar     # mymod/bar.py is a file
from mymod.bar import quz # quz is a function in mymod/bar.py

```

✓ 0.4s Python

- ... and even more alternatives

Docstrings in user modules

- Docstrings can be specified as the first statement in files (e.g., **`__init__.py`**)

```

file_b.py
1 import my_module
2
3

```

my_module > `__init__.py`

```

1 """
2 Quite a nice module!
3 """

```

"my_module" is not accessed Pylance
(module) my_module
Quite a nice module!
Quick Fix... (Ctrl+.)

```

file_b.py
1 from my_module import file_a
2
3

```

my_module > `file_a.py` > ...

```

1 """File A's functions are here"""

```

"file_a" is not accessed Pylance
(module) file_a
File A's functions are here
Quick Fix... (Ctrl+.)

Introduction to Python

Exceptions



Exceptions

- Like (almost) all modern languages, Python implements a mechanism for handling unexpected events in a smooth way through “exceptions”

```
try:  
    val = risky_code()  
except ValueError:  
    val = None  
except Exception as problem:  
    logging.critical(f"Yeuch: {problem}")
```

```
if val == None:  
    raise ValueError("Yeuch, invalid value")
```

Notable Exceptions

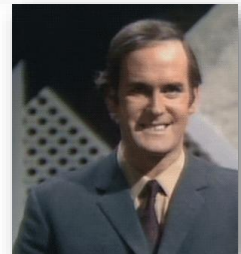
- **Exception**
- **ArithmeticError**
 - **OverflowError, ZeroDivisionError, FloatingPointError**
- **LookupError**
 - **IndexError, KeyError**
- **OSError**
 - System-related error, including I/O failures
- **UnicodeEncodeError, UnicodeDecodeError** and **UnicodeTranslateError**
- **ValueError**

Assert

- Check specific conditions at run-time
- Ignored if compiler is optimizing (`-O` or `-OO`)
- Generate an **AssertionError**

```
assert val != None, "Yeuch, invalid val"
```

- Advice: Use a lot of asserts in your code



Introduction to Python

i/o



Working with files

- As simple as

```
try:
    with open('file_name', 'r') as data_input:
        # read from data_input
        pass
except OSError as problem:
    logging.error(f"Can't read: {problem}")
```

- Caveat:
 - Use **try/except** to handle errors,
 - **with** to dispose resource automatically
 - Default encoding is **'utf-8'**

Open mode

- The mode may be specified setting the named parameter **mode** to 1 or more characters

Character	Meaning
'r'	open for reading (default)
'w'	open for writing, truncating the file first
'x'	open for exclusive creation, failing if the file already exists
'a'	open for writing, appending to the end of the file if it exists
'b'	binary mode
't'	text mode (default)
'+'	open for updating (reading and writing)

Under the hood

- If mode is not *binary*, a **TextIOWrapper** is used
 - buffered text stream providing higher-level access to a **BufferedIOBase** buffered binary stream
- If mode is *binary* and *read*, a **BufferedReader** is used
 - buffered binary stream providing higher-level access to a readable, non seekable **RawIOBase** raw binary stream
- If mode is *binary* and *write*, a **BufferedWriter** is used
 - buffered binary stream providing higher-level access to a writeable, non seekable **RawIOBase** raw binary stream

Reading/Writing text files

- **read(size)**
 - Reads up to n bytes, default slurp the whole file
- **readline()**
 - Reads 1 line
- **readlines()**
 - Reads the whole file and returns a list of lines
- **write(text)**
 - Write text into the file, no automatic newline is added
- **tell()/seek(offset)**
 - Gets/set the position inside a file

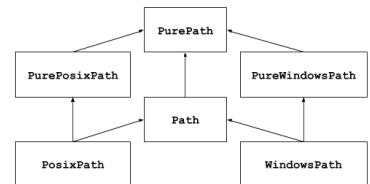


Example

```
try:
    with open('input.txt') as input, open('output.txt', 'w') as output:
        for line in input:
            output.write(line)
except OSError as problem:
    logging.error(problem)
```

Paths

- To manipulate paths in a somewhat portable way across different operating systems, use either
 - **os.path**
 - **pathlib** (more object oriented)
- Standard function names, such as **basename()** or **isfile()**
- Paths are always *local* path, to force:
 - **posixpath** (un*x)
 - **ntpath** (windoze)



Pickle

- Binary serialization and de-serialization of Python objects

```
import pickle

foo = get_really_complex_data_structure()
pickle.dump(foo, open('dump.p', 'wb'))
bar = pickle.load(open('dump.p', 'rb'))
```

- Caveats:
 - File operations should be inside **try/catch**
 - Use **protocol=0** to get a human-readable pickle file
 - The module is not *secure*! An attacker may tamper the pickle file and make **unpickle** execute arbitrary code



Read CSV

- As standard file

```

file = os.path.join(os.getcwd(), 'data_files', 'big_graphs_benchmark.csv')
try:
    with open(file) as csv_file:
        for line in csv_file:
            print(line.split(';'))
except OSError as problem:
    logging.error(f"Can't read {file.name}: {problem}")

```

✓ 0.8s Python

```

['GRAPH 0', 'GRAPH 0 SIZE', 'GRAPH 1', 'GRAPH 1 SIZE', 'ALGORITHM', 'SOLUTION SIZE', 'QUALITY MEASUR
['graphs/foie_gras/binary/120core_200classes_g0.txt', '238', 'graphs/foie_gras/binary/120core_200cl
['graphs/foie_gras/binary/120core_200classes_g0.txt', '238', 'graphs/foie_gras/binary/120core_200cl
['graphs/foie_gras/binary/120core_200classes_g0.txt', '238', 'graphs/foie_gras/binary/120core_200cl
['graphs/foie_gras/binary/120core_200classes_g0.txt', '238', 'graphs/foie_gras/binary/120core_200cl
['graphs/foie_gras/binary/120core_200classes_g0.txt', '238', 'graphs/foie_gras/binary/120core_200cl
['graphs/foie_gras/binary/120core_200classes_g0.txt', '238', 'graphs/foie_gras/binary/120core_200cl
['graphs/foie_gras/binary/120core_200classes_g0.txt', '238', 'graphs/foie_gras/binary/120core_200cl
['graphs/foie_gras/binary/120core_200classes_g0.txt', '238', 'graphs/foie_gras/binary/120core_200cl

```

giovanni.squillero@polito.it

Introduction to Python

143

Read CSV

- With the CSV module (*sniffing* the correct format)

```

import csv
file = os.path.join(os.getcwd(), 'data_files', 'big_graphs_benchmark.csv')
try:
    with open(file) as csv_file:
        dialect = csv.Sniffer().sniff(csv_file.read())
        csv_file.seek(0)
        for x in csv.reader(csv_file, dialect=dialect):
            print(x)
except OSError as problem:
    logging.error(f"Can't read {file.name}: {problem}")

```

✓ 0.8s Python

```

['GRAPH 0', 'GRAPH 0 SIZE', 'GRAPH 1', 'GRAPH 1 SIZE', 'ALGORITHM', 'SOLUTION SIZE', 'QUALITY MEASUR
['graphs/foie_gras/binary/120core_200classes_g0.txt', '238', 'graphs/foie_gras/binary/120core_200cl
['graphs/foie_gras/binary/120core_200classes_g0.txt', '238', 'graphs/foie_gras/binary/120core_200cl
['graphs/foie_gras/binary/120core_200classes_g0.txt', '238', 'graphs/foie_gras/binary/120core_200cl
['graphs/foie_gras/binary/120core_200classes_g0.txt', '238', 'graphs/foie_gras/binary/120core_200cl

```

giovanni.squillero@polito.it

Introduction to Python

144

Read Config

- Handle (read and write) standard config files

```
import configparser

config = configparser.ConfigParser()
config.read(os.path.join(os.getcwd(), 'data_files', 'sample-config.ini'))
print(config['Simple Values']['key'])
print('no key' in config['Simple Values'])
print('spaces in values' in config['Simple Values'])
```

Python

```
value
False
True
```

```
1 [Simple Values]
2 key=value
3 spaces in keys=allowed
4 spaces in values=allowed as well
5 spaces around the delimiter = obviously
6 you can also use : to delimit keys from values
```

Introduction to Python

Linters



Linting?

lint [from Unix's `lint(1)`, named for the bits of fluff it supposedly picks from programs] 1. /vt./ To examine a program closely for style, language usage, and portability problems, esp. if in C, esp. if via use of automated analysis tools, most esp. if the Unix utility `lint(1)` is used. This term used to be restricted to use of `lint(1)` itself, but (judging by references on Usenet) it has become a shorthand for desk check at some non-Unix shops, even in languages other than C. Also as /v./ `delint`. 2. /n./ Excess verbiage in a document, as in "This draft has too much **lint**".

pylint

- A static code-analysis tool which looks for programming errors, helps enforcing a coding standard, sniffs for code smells and offers simple refactoring suggestions

```
conda install pylint
```

- Or

```
pip install -U pylint
```

```
(base) λ pylint ex07_random.py
***** Module ex07_random
ex07_random.py:1:0: C0114: Missing module docstring (missing-module-docstring)
ex07_random.py:14:0: C0116: Missing function or method docstring (missing-function-docstring)
-----
Your code has been rated at 8.75/10 (previous run: 9.38/10, -0.62)
```

flake8

- A tool for enforcing style consistency across

```
conda install flake8
```

- Or

```
pip install -U flake8
```

```
(base) λ flake8 ex07_random.py
ex07_random.py:24:80: E501 line too long (99 > 79 characters)
```

```
(base) λ flake8 --max-line-length=100 ex07_random.py
```

bandit

- A static code-analysis tool which finds common security issues in Python code

```
conda install -c conda-forge bandit
```

- Or

```
pip install -U bandit
```

```
(base) A bandit ex07_random.py
[main] INFO profile include tests: None
[main] INFO profile exclude tests: None
[main] INFO cli include tests: None
[main] INFO cli exclude tests: None
[main] INFO running on Python 3.8.11
[mode_visitor] INFO Unable to find qualified name for module: ex07_random.py
Run started:2021-09-09 11:46:00,080207

Test results:
+ Score: 10/11:bandit Standard pseudo-random generators are not suitable for security/cryp
tographic purposes
  Severity: Low  Confidence: High
  Location: ex07_random.py:21
  More info: https://bandit.readthedocs.io/en/latest/blacklists/blacklist_calls.html#b311-rand
om
20
    }) < 2:
21         sequence.append(random.randint(MIN_RANDOM, MAX_RANDOM))
22
23
-----
Code scanned:
  Total lines of code: 19
  Total lines skipped (#nosec): 0

Run metrics:
Total issues (by severity):
  Undefined: 0.0
  Low: 1.0
  Medium: 0.0
  High: 0.0
Total issues (by confidence):
  Undefined: 0.0
  Low: 0.0
  Medium: 0.0
  High: 1.0
Files skipped (0)
```