INTRODUCTION TO Python

Introduction to Python

https://github.com/squillero/python-lectures/

september 2023 **version 0.5**

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

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
```

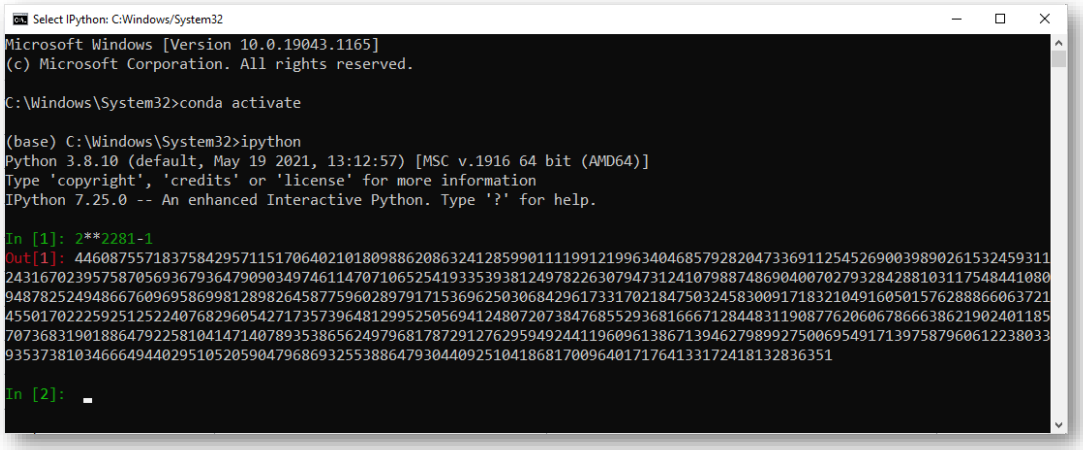
- Use the standard package manager (pip) to get additional modules

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

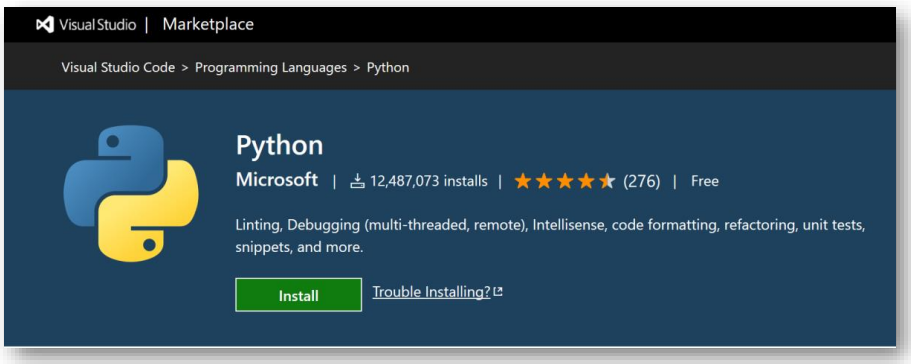
Getting Python

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

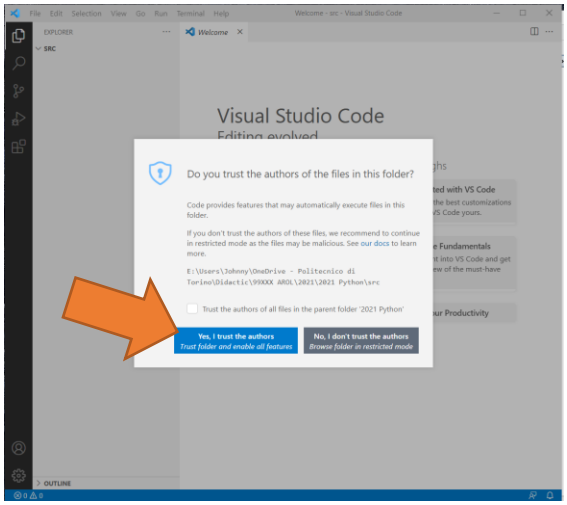


IDE

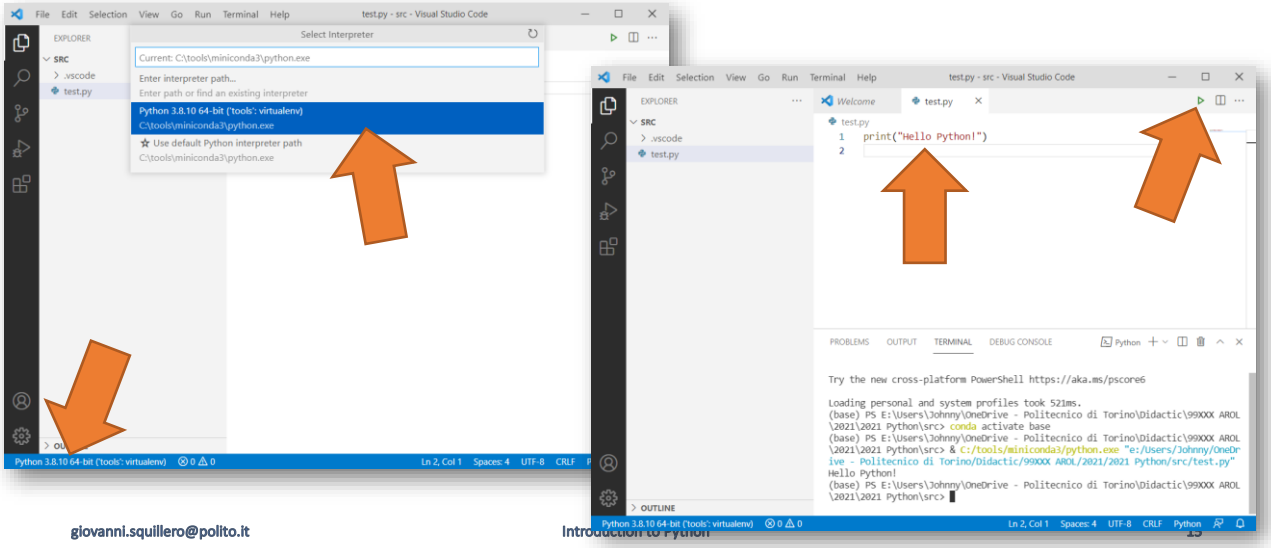
- Install Visual Studio Code and the Python Extension



Open a “directory”

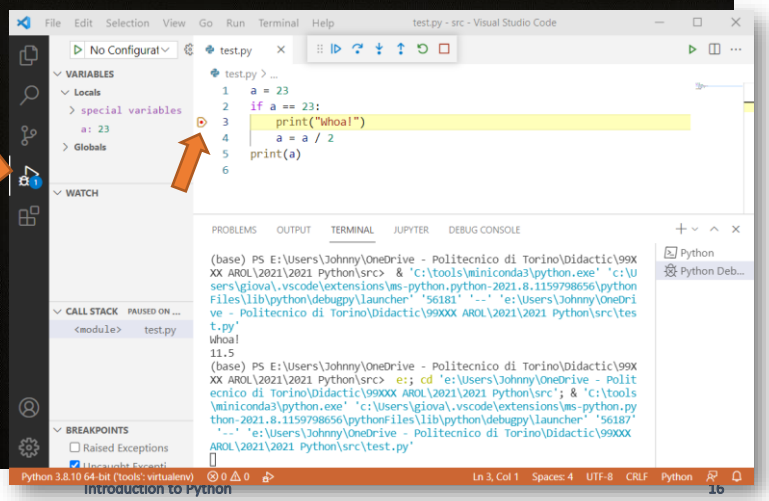


Set interpreter & Press play

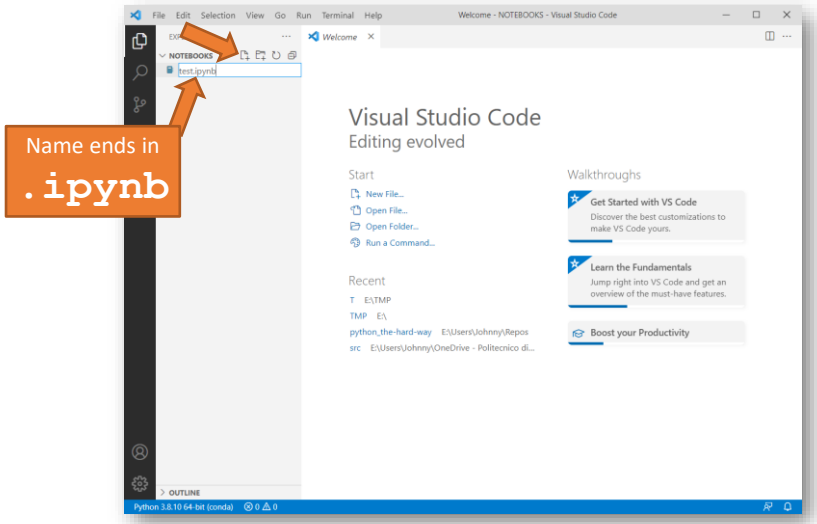


Kick off

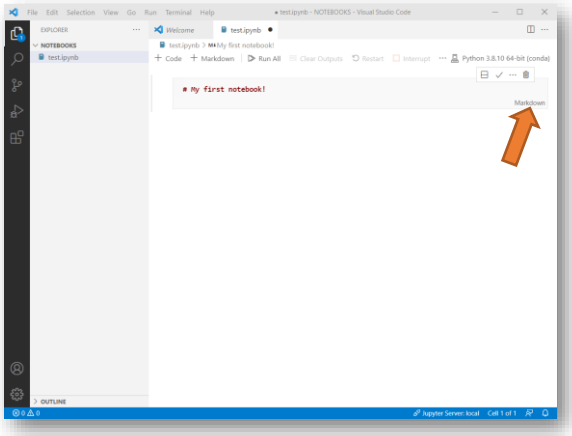
```
a = 23
if a == 23:
    print("Whoa!")
    a = a / 2
print(a)
```

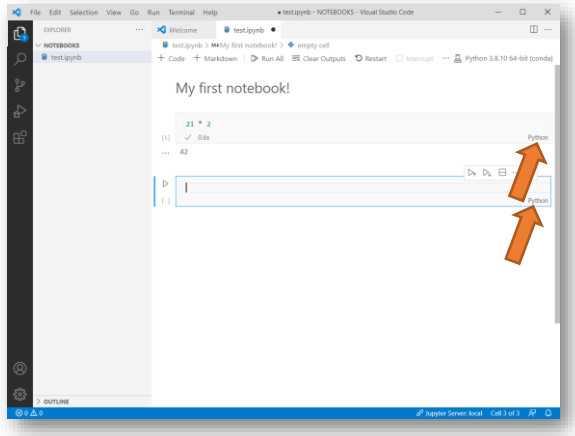


Jupyter Notebook

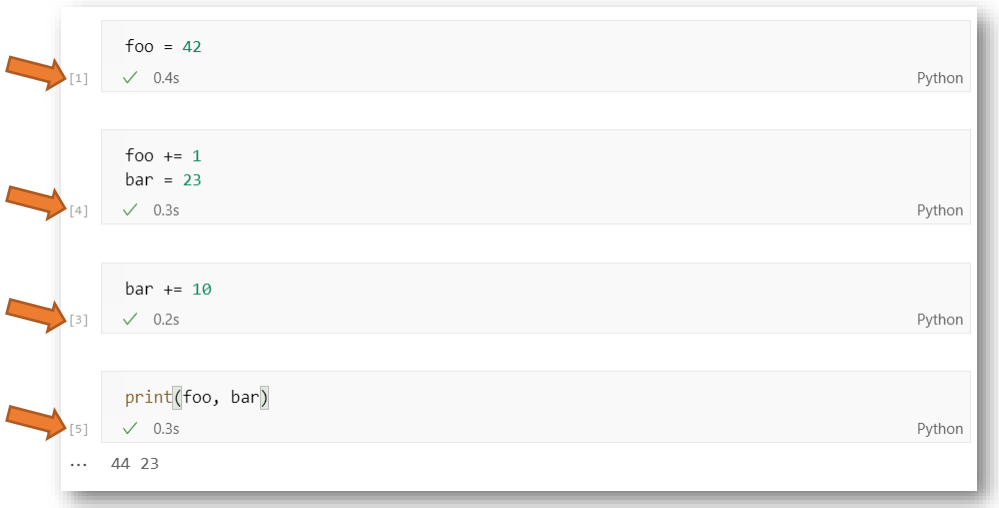


My first notebook





Execution order…

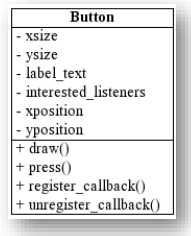


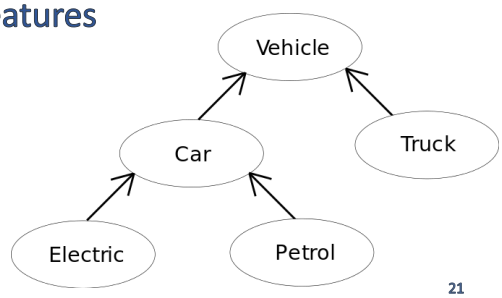
Data Model

- 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



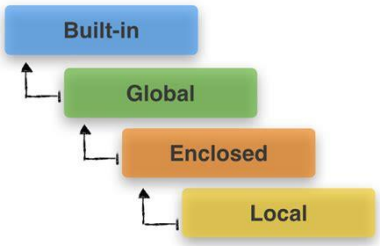


Naming & Binding

- 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 = 

Naming & Binding

A = 

B = 

C = A

Naming & Binding

```
drink(A)
drink(B)
drink(C)
```

Naming & Binding

```
A =
B =
C = A
A =
```

Naming & Binding

```
A = 
```


```
B = A

C = 
```


```
drink(A)
```

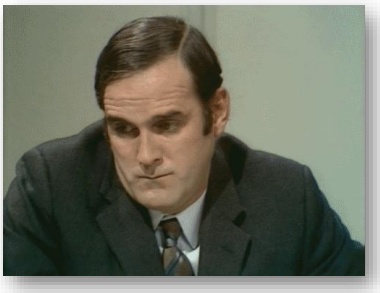
Alias vs. Copy

```
A = 
```


```
B = A

C = copy(A)

drink(A)
```

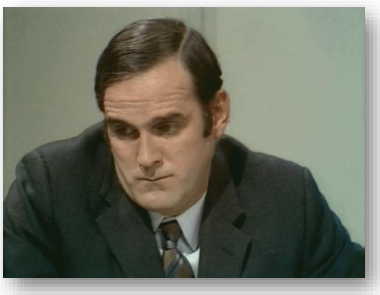
Variables?

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



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

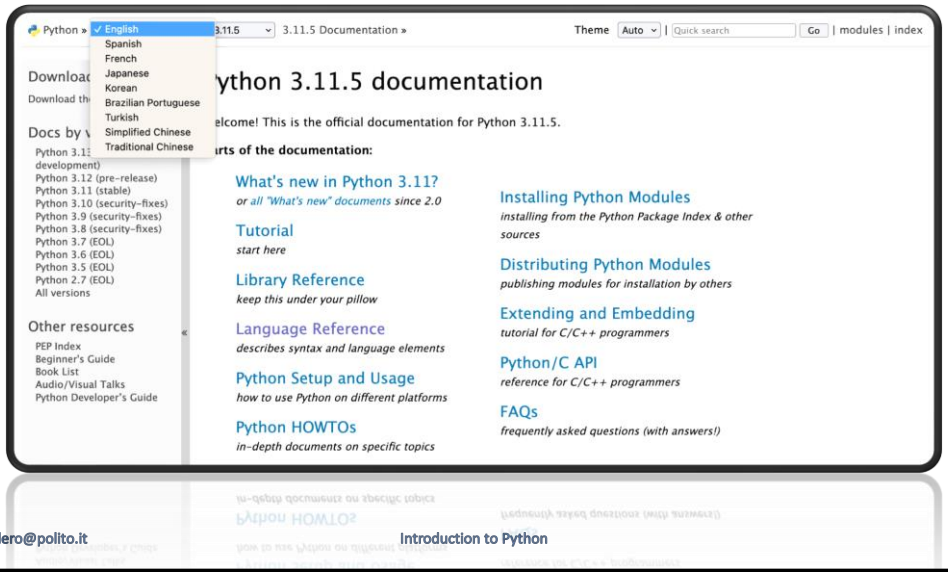
```
foo = 42
```

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/

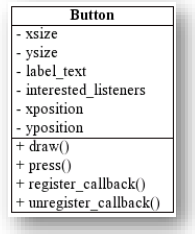


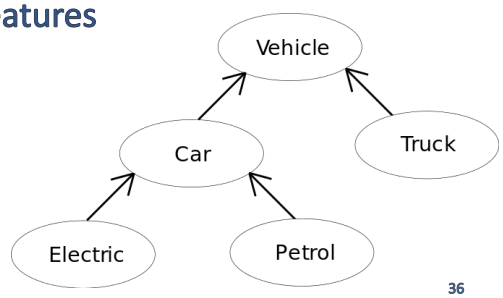
Data Model

- 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





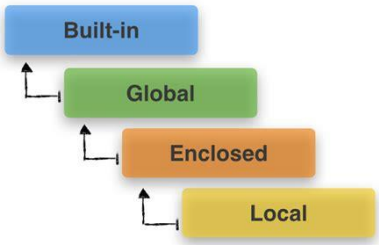
Naming & Binding

- 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 = 

Naming & Binding

A = 

B = 

C = A

Naming & Binding

drink(A)

drink(B)

drink(C)

Naming & Binding

A = 

B = 

C = A

A = 

Naming & Binding

A = 

B = A

C = 


```
drink(A)
```

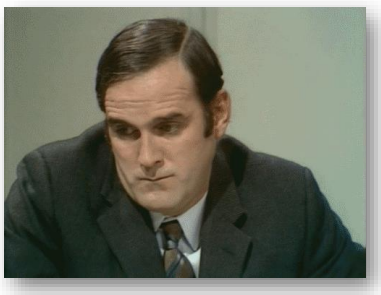
Alias vs. Copy

```
A = 
```


```
B = A

C = copy(A)

drink(A)
```

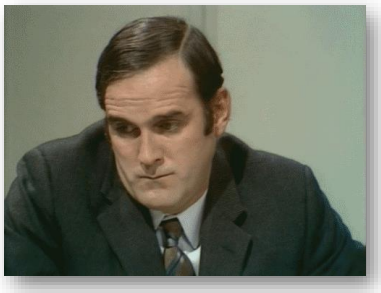
Variables?

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



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

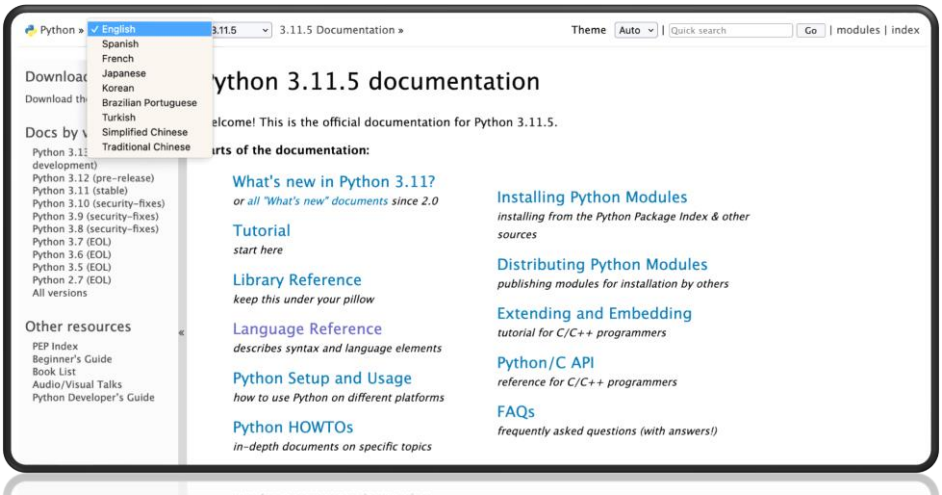
```
foo = 42
```

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`**)

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

- Caveat:
  - Sets are **mutable**, frozen sets are **immutable**

Standard Type Hierarchy

- Mappings
  - Dictionaries (**`dict`**)

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


Standard Type Hierarchy

- None (**`NoneType`**)

```
foo = None
```

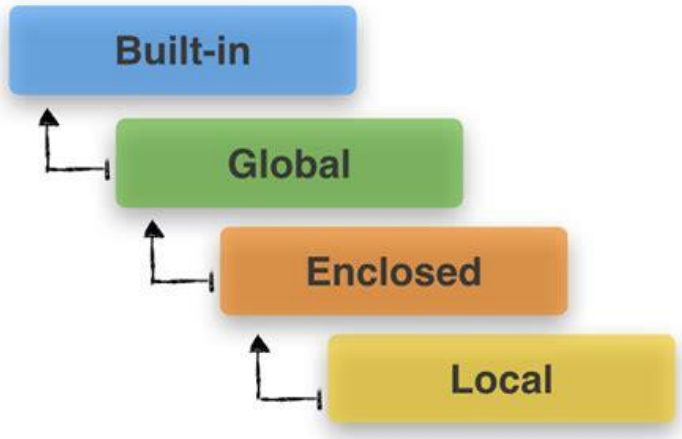
Standard Type Hierarchy

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

Introduction to Python

Basic Syntax

L E G B !



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`

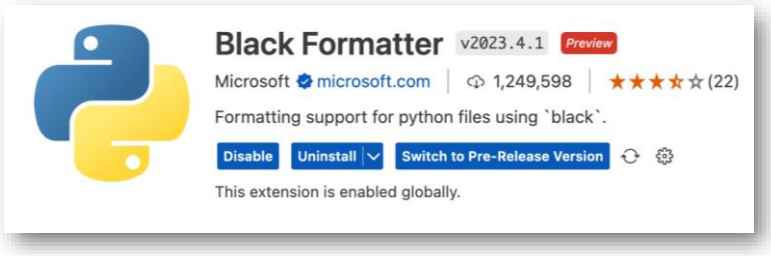
Style

- 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

```
for item in range(10):
    print('I')
    print('am')
    print('a')
    if item % 2 == 0:
        print('funny')
        print('and')
        print('silly')
    else:
        print('dull')
        print('and')
        print('serious')
    print('block')
    print('used')
    print('as')
    print('example.')
```


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

Real-number operations

Operation	Result
`math.trunc(x)`	*x* truncated to `Integral`
`round(x[, n])`	*x* rounded to *n* digits, rounding half to even. If *n* is omitted, it defaults to 0.
`math.floor(x)`	the greatest `Integral` <= *x*
`math.ceil(x)`	the least `Integral` >= *x*

Bitwise operations

Operation	Result	Notes
`x | y`	bitwise *or* of *x* and *y*	(4)
`x ^ y`	bitwise *exclusive or* of *x* and *y*	(4)
`x & y`	bitwise *and* of *x* and *y*	(4)
`x << n`	*x* shifted left by *n* bits	(1)(2)
`x >> n`	*x* shifted right by *n* bits	(1)(3)
`~x`	the bits of *x* 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
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)
`s.count(x)`	total number of occurrences of *x* in *s*

Looping over sequences

```python
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

```python
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()`

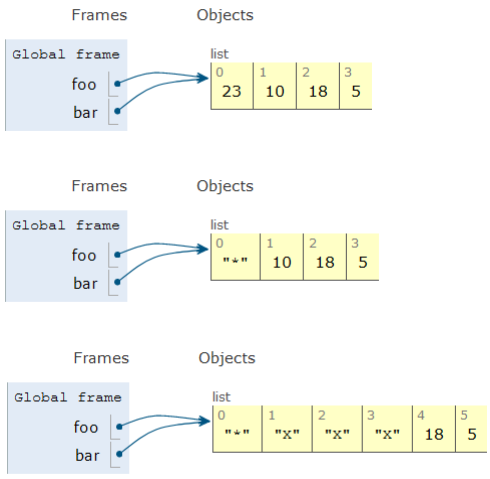
```
birthday = [["Giovanni", 23, 10], ["Paola", 18, 5]]
print(birthday[0])

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)

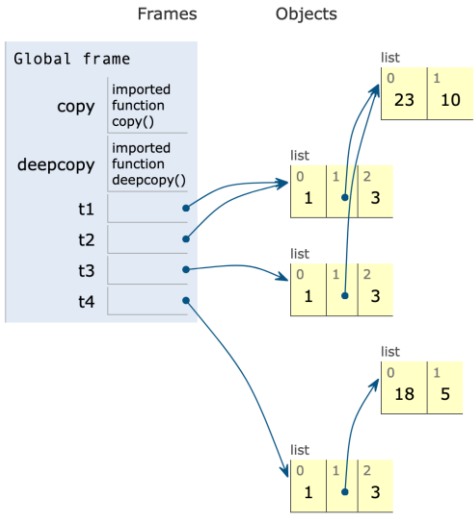
Slices & Name binding

```
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
`s[i] = x`	item *i* of *s* is replaced by *x*
`s[i:j] = t`	slice of *s* from *i* to *j* is replaced by the contents of the iterable *t*
`del s[i:j]`	same as `s[i:j] = []`
`s[i:j:k] = t`	the elements of `s[i:j:k]` are replaced by those of *t*
`del s[i:j:k]`	removes the elements of `s[i:j:k]` from the list
`s.append(x)`	appends *x* to the end of the sequence (same as `s[len(s):len(s)] = [x]`)
`s.clear()`	removes all items from *s* (same as `del s[:]`)

`s.copy()`	creates a shallow copy of *s* (same as `s[:]`)
`s.extend(t)` or `s += t`	extends *s* with the contents of *t* (for the most part the same as `s[len(s):len(s)] = t`)
`s *= n`	updates *s* with its contents repeated *n* times
`s.insert(i, x)`	inserts *x* into *s* at the index given by *i* (same as `s[i:i] = [x]`)
`s.pop()` or `s.pop(i)`	retrieves the item at *i* and also removes it from *s*
`s.remove(x)`	remove the first item from *s* where `s[i]` is equal to *x*
`s.reverse()`	reverses the items of *s* 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!

```python
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

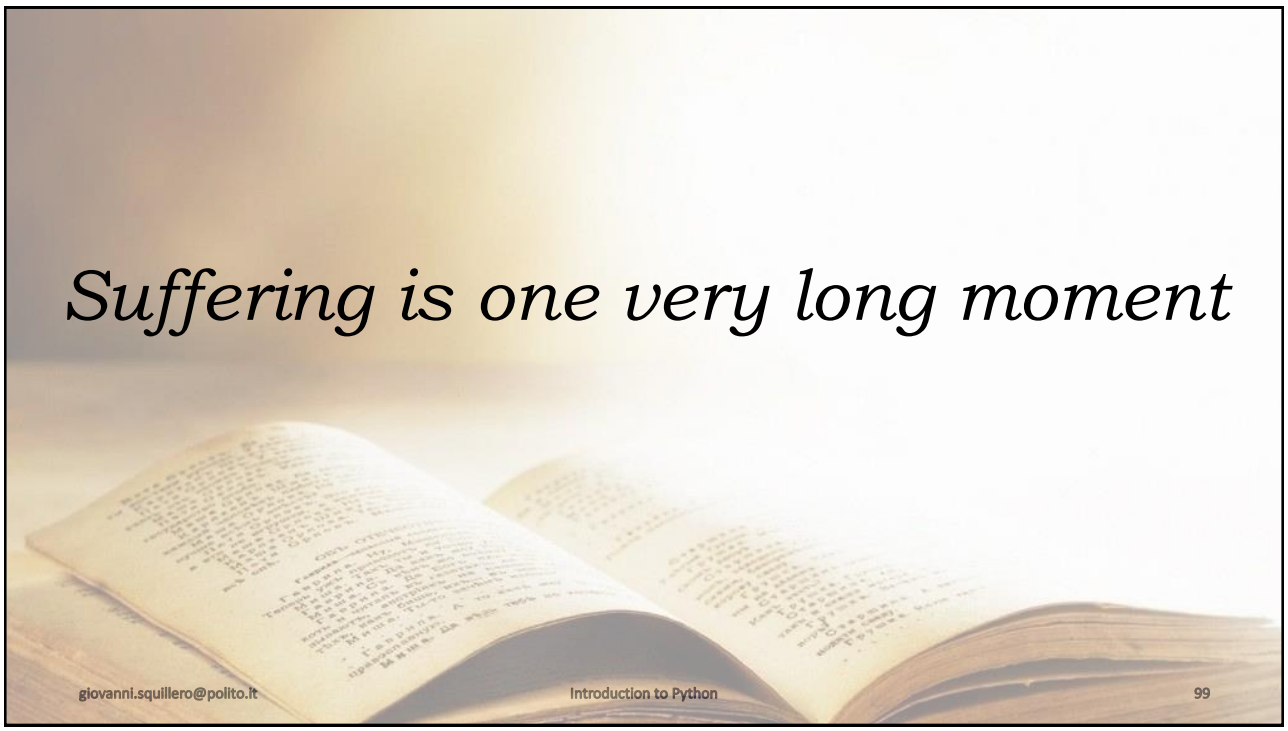
```
{'M', 'A', 'I'}
```

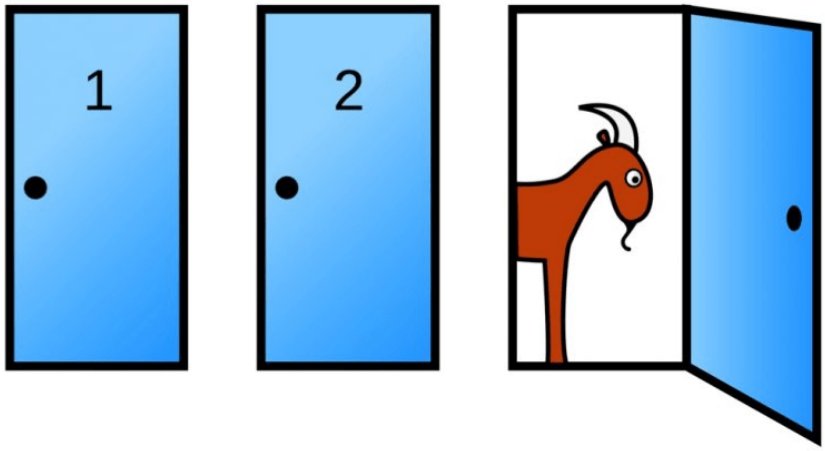

Copy and Delete

- **`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))
```

- Caveat:
  - Names vs. Variables

More caveats

- Functions are first-class citizen
- Function names are just “names”
- Remember scope!
- No **return** statement is equivalent to a **return None**

```
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

foo()
```

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


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

```python
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 **kargs

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

```python
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 **kargs

- 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')
```

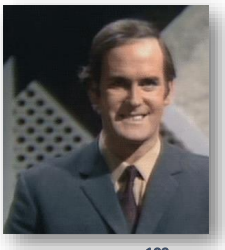
✓ 0.2s Python

```
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])
```

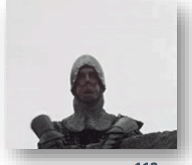

Closures and Scope

- 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
```



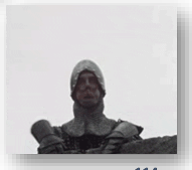
Closures and Scope

- 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
3,125
```

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
```


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

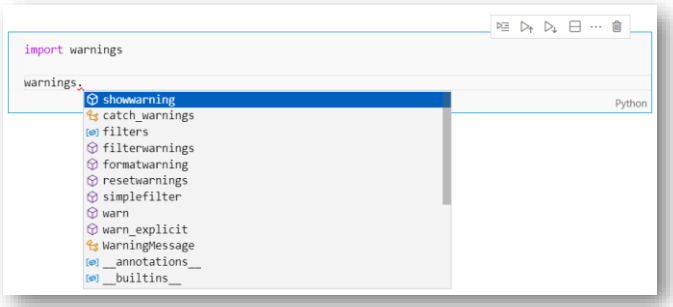
```python
import logging

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...
```

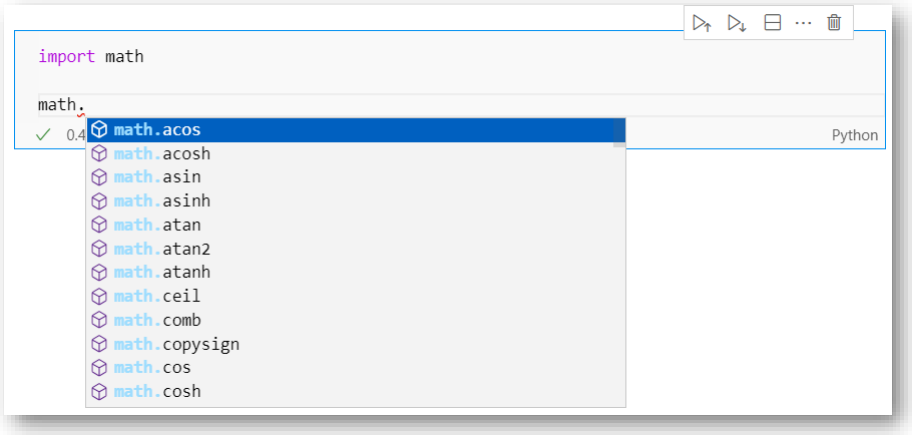

Notable Modules

- 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`**



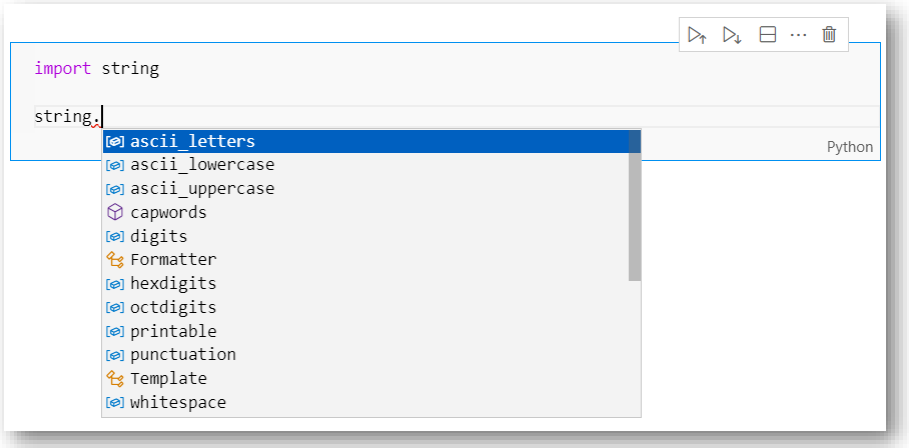
Notable Modules

- Mathematical functions, use `cmath` for complex numbers



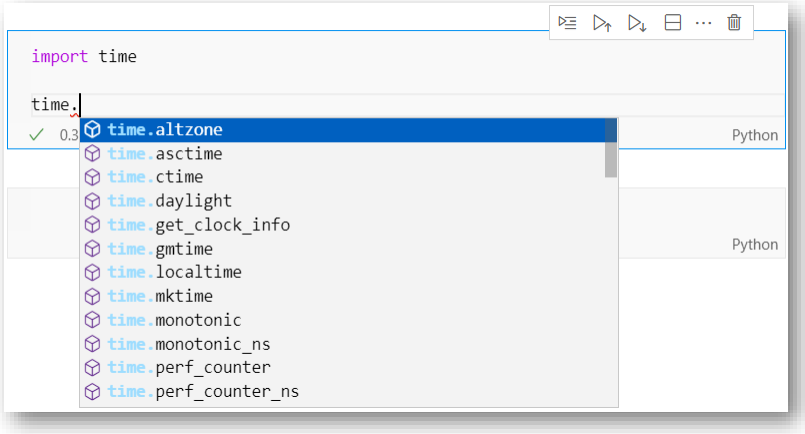
Notable Modules

- Common string operations and constants



Notable Modules

- Various time-related functions



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'
```

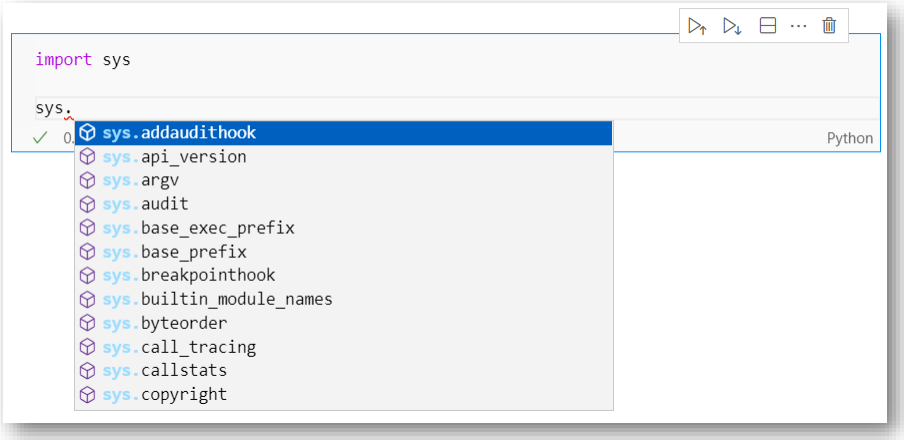
```
os.
```

Python

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

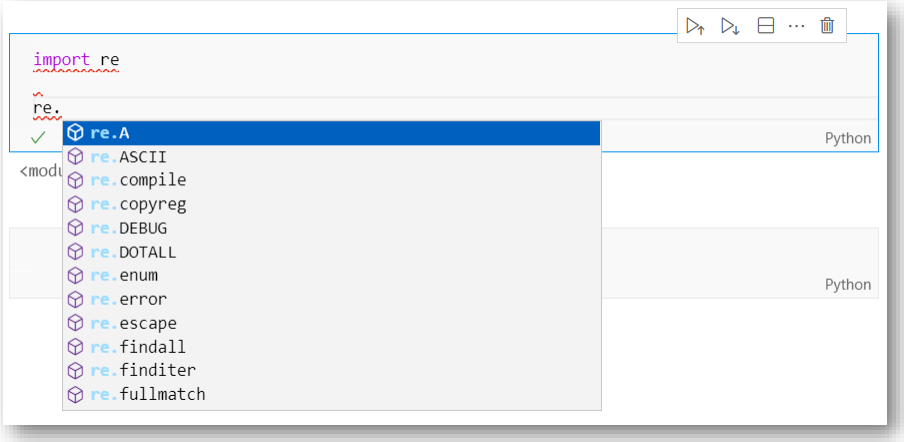
Notable Modules

- System-specific parameters and functions



Notable Modules

- Regular expression operations



Notable Modules

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

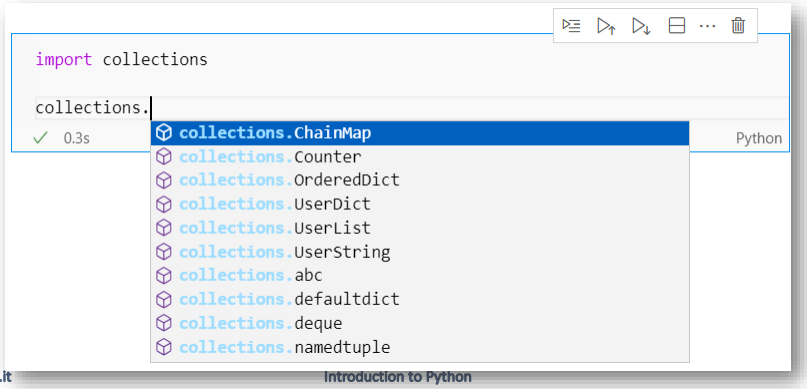
Examples	Results
`product('ABCD', repeat=2)`	`AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD`
`permutations('ABCD', 2)`	`AB AC AD BA BC BD CA CB CD DA DB DC`
`combinations('ABCD', 2)`	`AB AC AD BC BD CD`
`combinations_with_replacement('ABCD', 2)`	`AA AB AC AD BB BC BD CC CD DD`

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 **collection** contains specialized container datatypes providing alternatives to Python's general purpose built-in containers

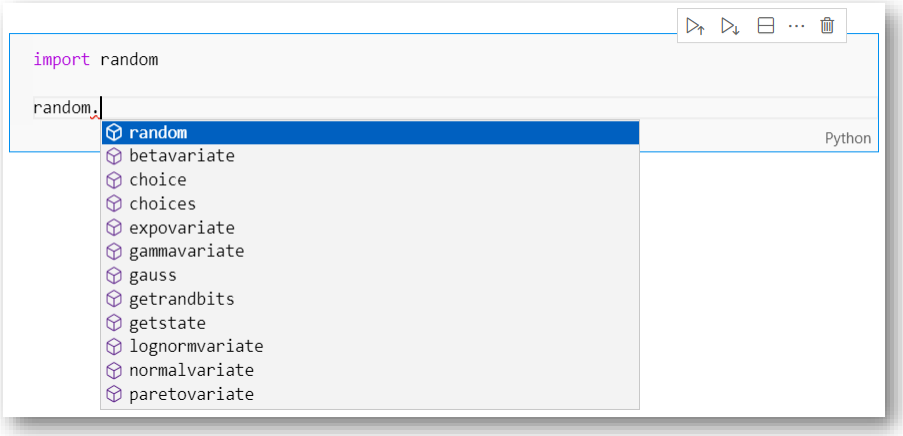


Notable Modules: collections

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

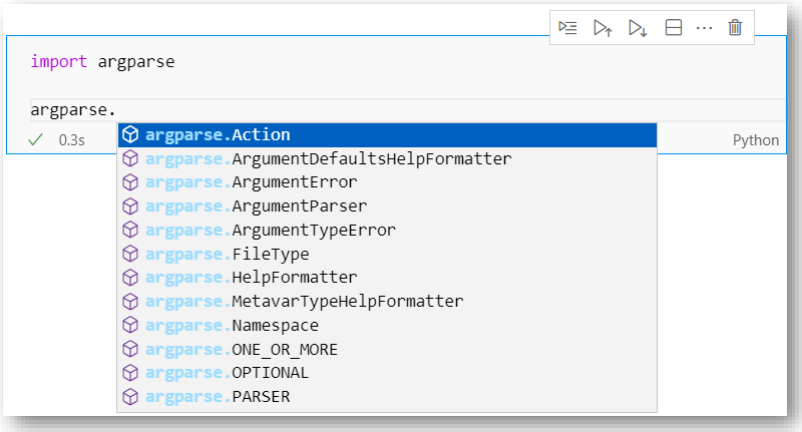
Notable Modules

- Generate pseudo-random numbers



Notable Modules

- Parse command-line arguments



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
```
def foo(x):
    print(f"FileA's foo({x})!")
```

file_b.py
```
import file_a

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})!")
```

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

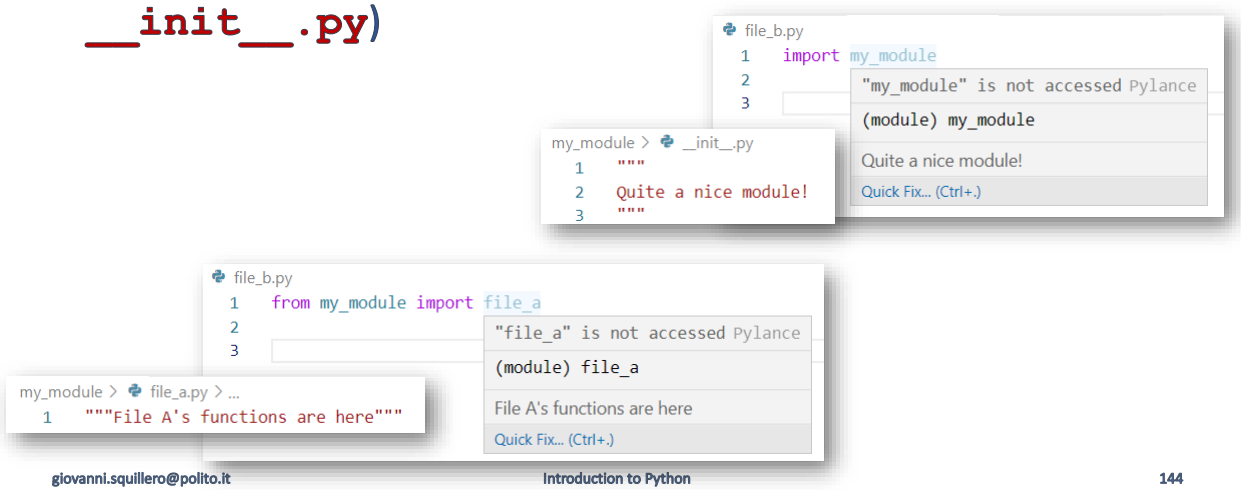
```
import foo                  # foo.py is a file
import mymod                # mymod is a directory containing __init__.py
from mymod import bar       # mymod/bar.py is a file
from mymod.bar import quz   # quz is a function in mymod/bar.py
```

✓ 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`**)



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

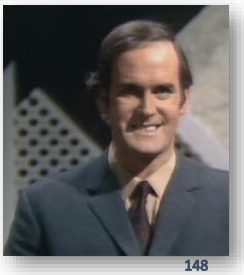
- **`Exception`**
- **`ArithmeticError`**
  - **`OverflowError`**, **`ZeroDivisionError`**, **`FloatingPointError`**
- **`LookupError`**
  - **`IndexError`**, **`KeyError`**
- **`OSError`**
  - System-related error, including I/O failures
- **`UnicodeEncodeError`**, **`UnicodeDecodeError`** and **`UnicodeTranslateError`**
- **`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 automagically
  - Default encoding is **`'utf-8'`**

Open mode

- The mode may by specified setting the named parameter **`mode`** to 1 ore 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

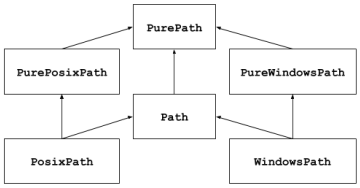
- **`read(size)`**
  - Reads up to n bytes, default slurp the whole file
- **`readline()`**
  - Reads 1 line
- **`readlines()`**
  - Reads the whole file are returns a list of lines
- **`write(text)`**
  - Write text into the file, no automatic newline is added
- **`tell()/seek(offset)`**
  - Gets/set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

```python
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

```python
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_200cla
['graphs/foie_gras/binary/120core_200classes_g0.txt', '238', 'graphs/foie_gras/binary/120core_200cla
['graphs/foie_gras/binary/120core_200classes_g0.txt', '238', 'graphs/foie_gras/binary/120core_200cla
['graphs/foie_gras/binary/120core_200classes_g0.txt', '238', 'graphs/foie_gras/binary/120core_200cla
['graphs/foie_gras/binary/120core_200classes_g0.txt', '238', 'graphs/foie_gras/binary/120core_200cla
['graphs/foie_gras/binary/120core_200classes_g0.txt', '238', 'graphs/foie_gras/binary/120core_200cla
['graphs/foie_gras/binary/120core_200classes_g0.txt', '238', 'graphs/foie_gras/binary/120core_200cla
['graphs/foie_gras/binary/120core_200classes_g0.txt', '238', 'graphs/foie_gras/binary/120core_200cla
```

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 MEASU
['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
```


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
```