Python

The Hard Way

1

*** DRAFT v0.1 ***

https://github.com/squillero/python_the-hard-way/

2

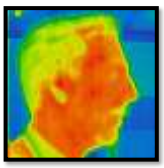
Why Python?

- 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 some domain (e.g., data science)

Why "the Hard Way"?

- No-nonsense Python for programmers
- Not a *gentle introduction*
- Not the usual *Dummy's guide to…*

Who is this Giovanni Squillero, anyway?



Associate Professor
of Computer Science
@ PoliTO (DAUIN)

- Courses:
  - Computer Sciences
  - Computational Intelligence
  - Futuro del Lavoro
  - Mimetic Learning
  - Computer Architectures [in Uzbekistan]

- Current ML projects
  - Test & Validation
  - Analysis of Mass Specra (COVID)
- Current CI projects
  - Antimicrobial susceptibility from DNA
  - *AR*tificial Intellgence (AI4MUSE)
- Research (not applied)
  - Diversity promotion in EC
  - Feature-selection is ML
  - EDA
  - Games/Economy
  - Multi-agent systems

5

6

Python — The Hard Way

Set-up

7

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`
- Course repo
  - `https://github.com/squillero/python_the-hard-way`

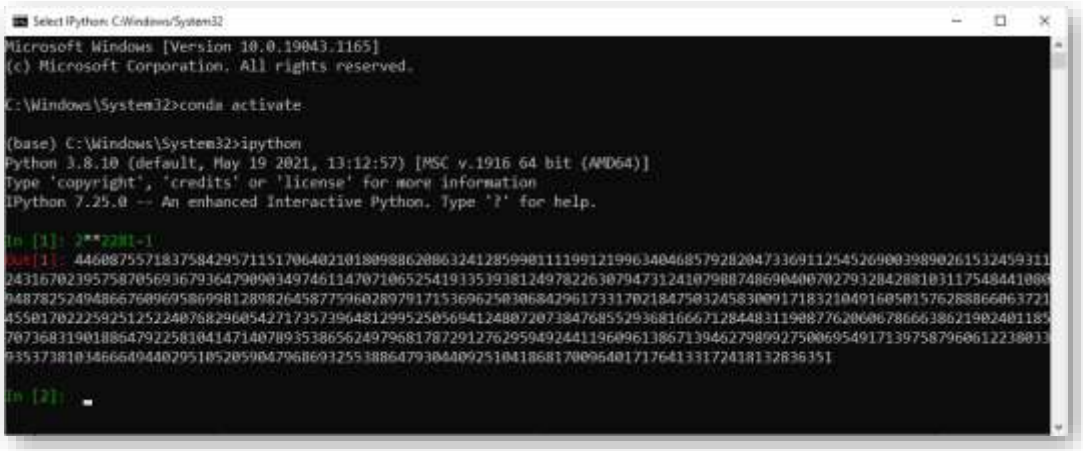
8

Getting Python

- Official Python downloads
  - **https://www.python.org/downloads/**
- On Linux
  - Get the default package from the distro
  - Then use **Virtualenv** (https://virtualenv.pypa.io/)
- On Windows and MacOS
  - Install **Anaconda** (https://www.anaconda.com/products/individual)
  - If size is a **real** issue, also consider **micromamba**
- If everything else fail:
  - Try **ActivePython** (https://www.activestate.com/products/python/)

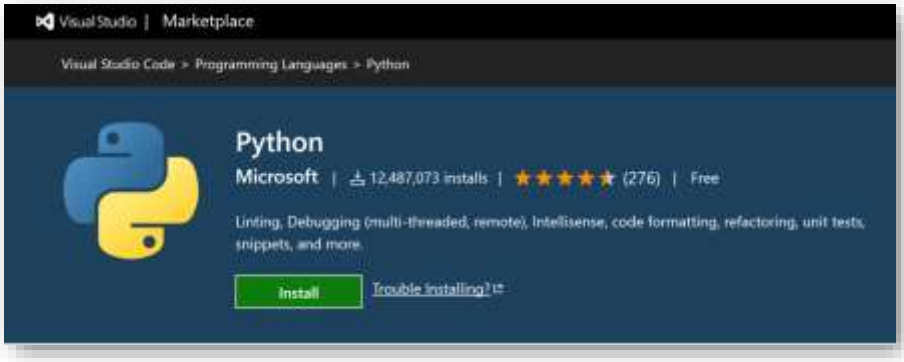
Getting Python

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

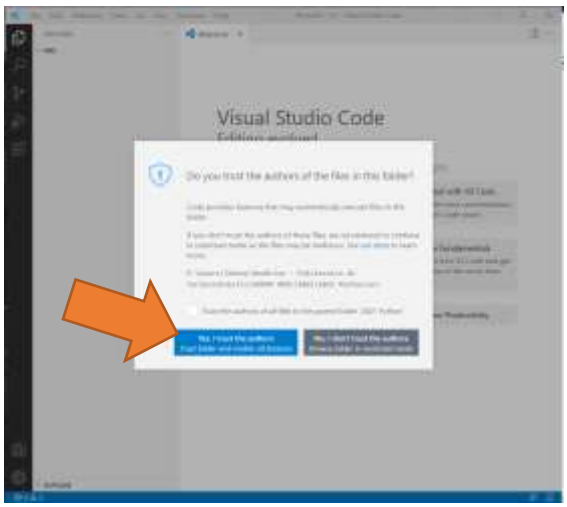


IDE

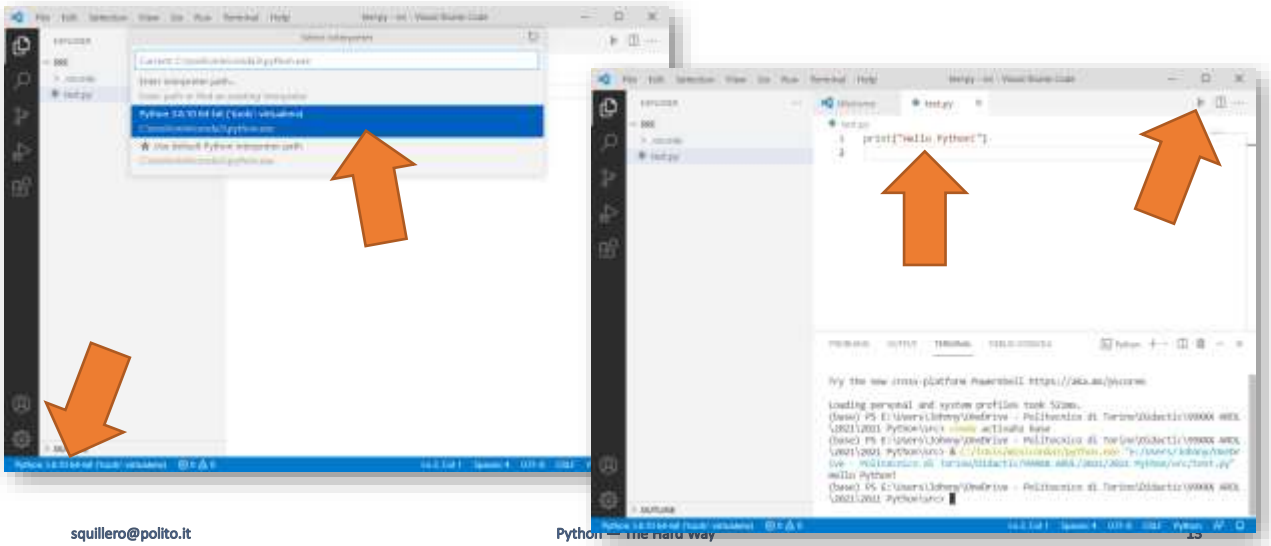
- Install Visual Studio Code and the Python Extension



Open a “directory”

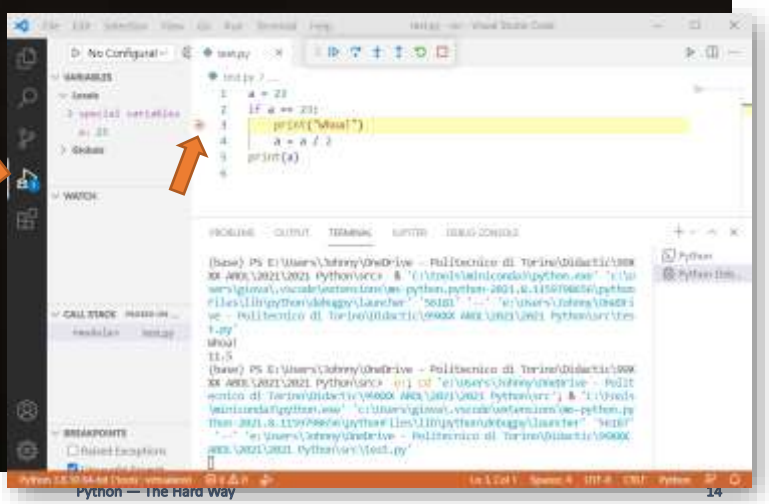


Set interpreter & Press play

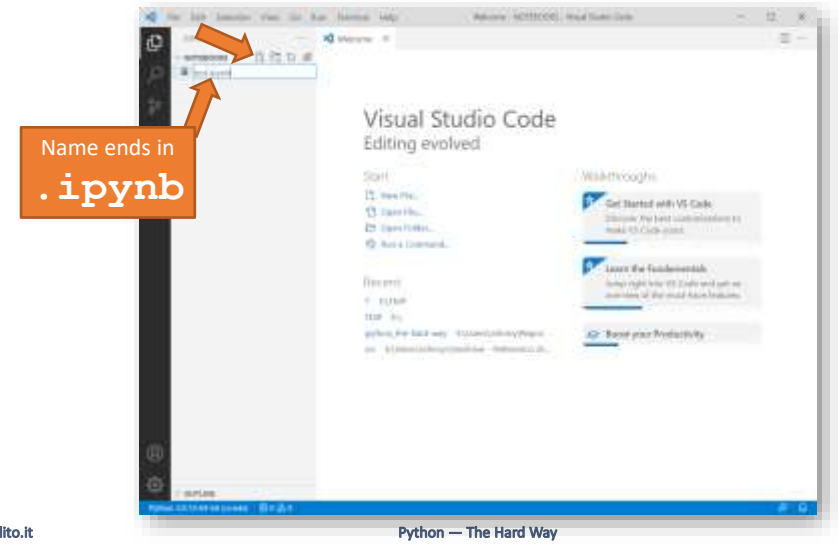


Kick off

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

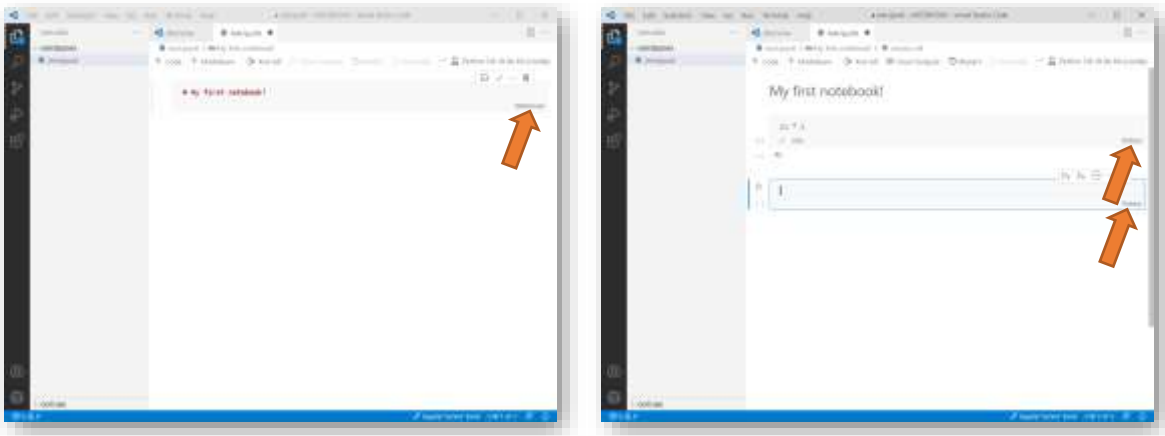


Jupyter Notebook



15

My first notebook



16

Execution order…

Python — The Hard Way

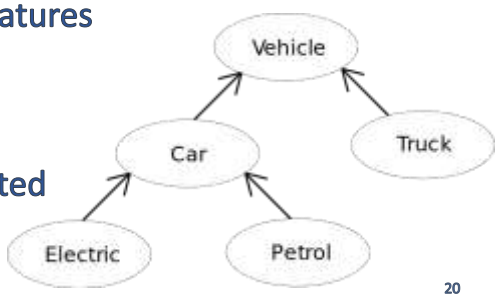
Data Types

Data Model

- Python is a **strongly-typed**, **dynamic**, **object-oriented** language
  - Objects are Python's abstraction for data
  - Code is also represented by objects
  - Every object has an **identity**
    - The identity never changes once it has been created — `is`, `id()`
  - Every object has a **type** and a **value**

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**
  - caller ignores which class in the supported hierarchy it is operating on



Naming & Binding

- Names refer to objects

  `foo = 42`
  `foo` is a name (not a "variable")
- The scope defines the visibility of a name
- When a name is used, it is resolved using the nearest enclosing scope

Standard Type Hierarchy

- Number
  - Integral
    - Integers (**`int`**)
    - Booleans (**`bool`**)
  - Real (**`float`**)
  - Complex (**`complex`**)
- 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]
fud = b'\x04\x02'
```


Standard Type Hierarchy

- **Mutable** Sequences vs. **Immutable** Sequences

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

Python — The Hard Way

Basic Syntax

29

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`

30

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 know what you are doing
  - int_

```
無 = 0
_ = 42
```

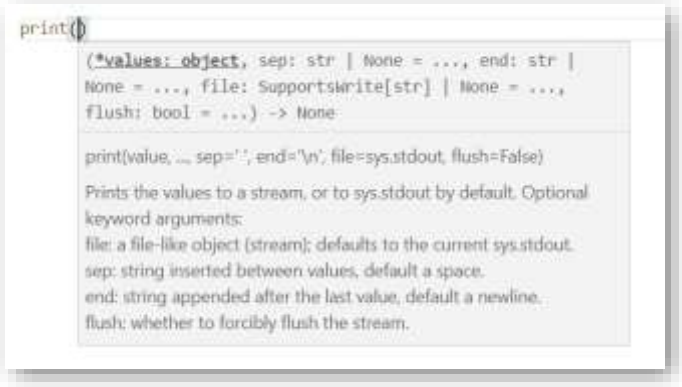

Comments

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

```
# This is a comment

"""
This is a multi-line string,
and in certain contexts may be used as a comment
"""
```

Basic I/O: `print`

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



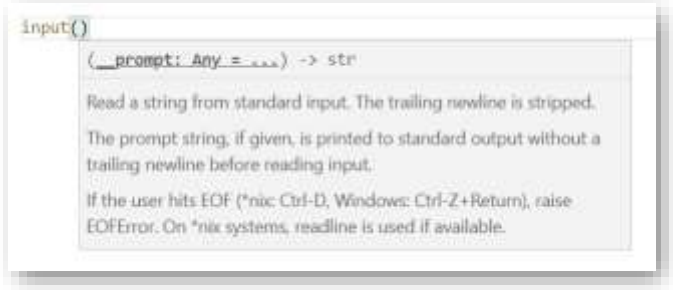
Pythonic Approach

- 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



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.')
```

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

- Operators almost C-like:

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

- 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
- No self pre/post inc/dec-rement: **++ --**
  (numbers are immutable, names are not variables)

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

```
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
```


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()`**

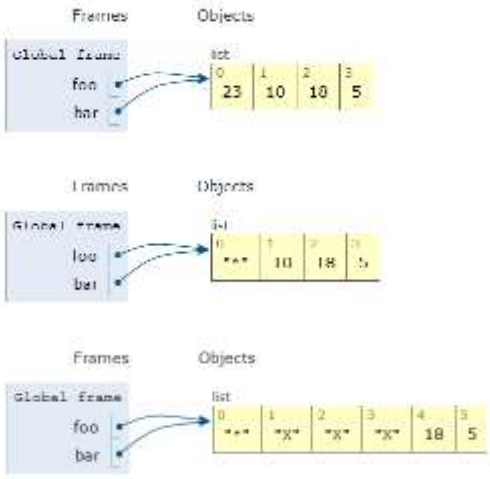
```
birthday = [["Giovanni", 23, 10], ["Paola", 18, 5]]
print(birthday[0])

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, 10, 5]
bar = foo
bar[0] = "x"
foo[1:2] = list("xxx")
```



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

- Several alternatives available
- 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

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[:]`**

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
```


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!")
```

Python — The Hard Way

Functions

65

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

66

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()
```


Named and Default Arguments

- Remember scope and name binding
- Arguments may be optional
- Named 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)
```

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()
```

Python — The Hard Way

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 0x000002604741C080>
```

```
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
```

Python — The Hard Way

Modules

75

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
```


76

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

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

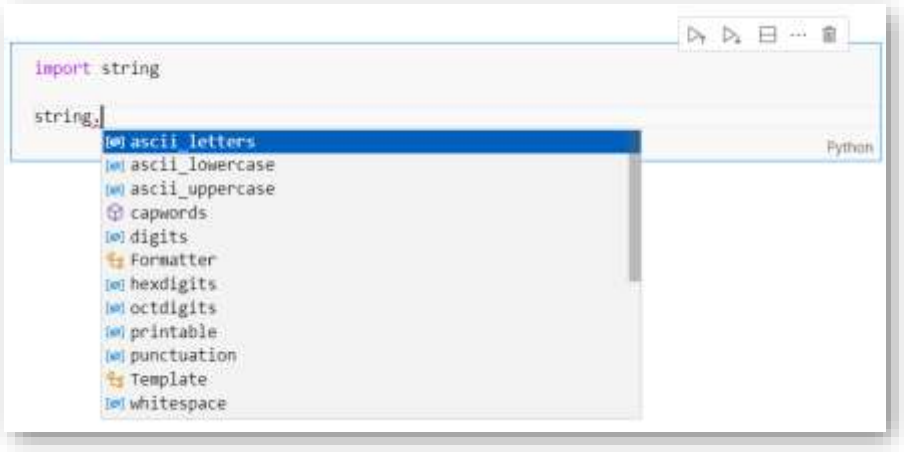
```python
import math

math.
```

✓ 0.4 Python

```
math.acos
math.acosh
math.asin
math.asinh
math.atan
math.atan2
math.atanh
math.ceil
math.comb
math.copysign
math.cos
math.cosh
```

Notable Modules

- Common string operations and constants



Notable Modules

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

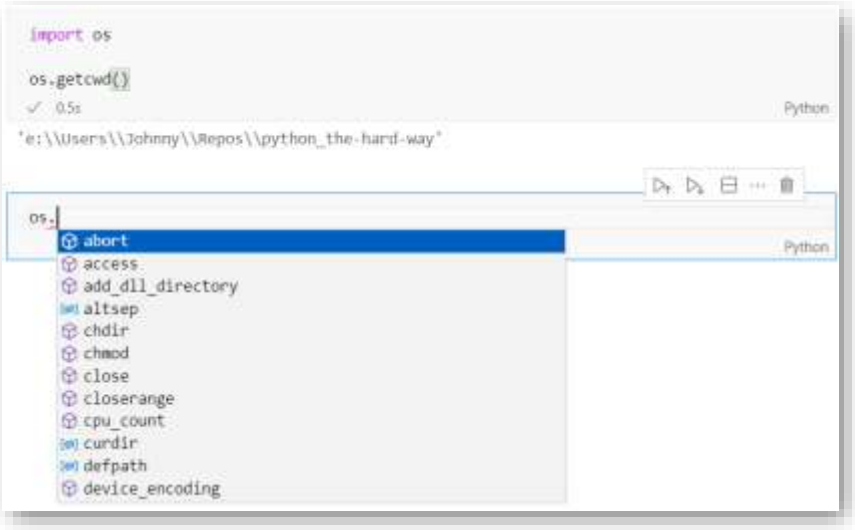
```
from pprint import pprint

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}]
```

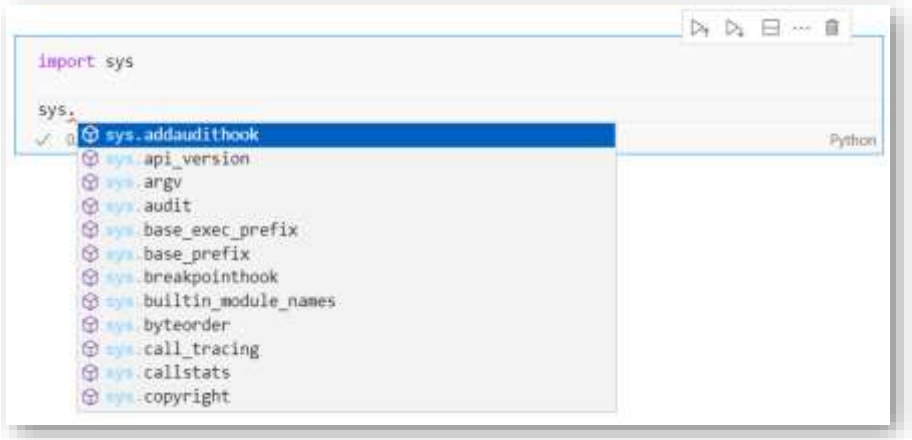
Notable Modules

- Miscellaneous operating system interfaces



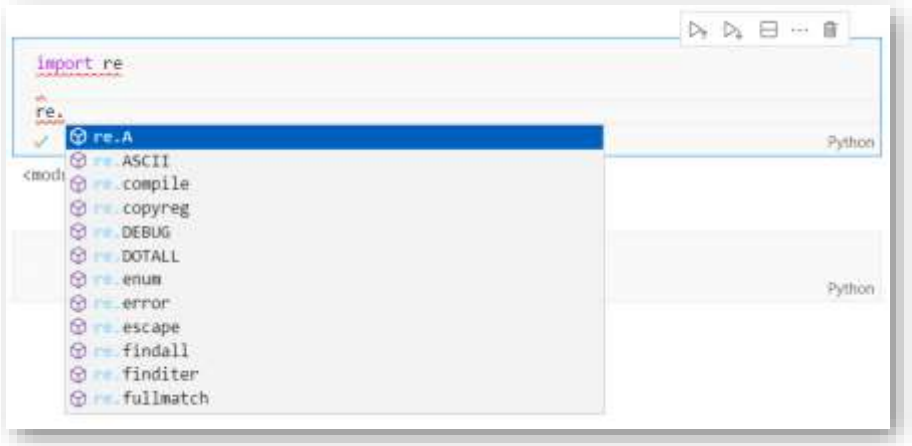
Notable Modules

- 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

- Generate pseudo-random numbers

Python — The Hard Way

Exceptions

87

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")
```

88

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

Python — The Hard Way

i/o

91

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'`**

92

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

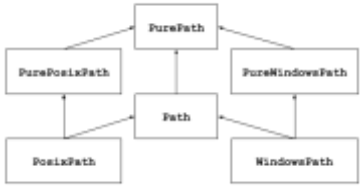
- `read(size)`
  - Reads up to n bytes, default slurp the whole file
- `readline()`
  - Reads 1 line
- `readlines()`
  - Reads the whole file are returns a list of lines
- `write(text)`
  - Write text into the file, no automatic newline is added
- `tell()/seek(offset)`
  - Gets/set the position inside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

```
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_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

```python
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
```

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