DM587/DM579
Scientific Programming
Linear Algebra with Applications

Python - Part 1

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

[*Based on the booklet "Python Essentials"*]

Outline

Outline

Contents

/// Introductory Classes

Week	Date	Topics and Slides	Suggested reading
43	Oct 23	Python - Part 1: basics, data types, control flow	App A, B and ch 1-3 of [HJ1]; [DB]
43	Oct 25	Python - Part 2: std library, OO progr.	
44	Oct 31	Python - Part 3: exceptions, file i/o, numpy	
44	Nov 2	Python - Part 4: graphics, data viz, pandas	
45	Nov 9	Least Squares Data Fitting	
47	Nov 16	Graph Isomorphism and Molecules	
46	Nov 23	Page Rank	
49	Nov 30	Eigenfaces	
48	Dec 4	From Random Polygon to Ellipse	
50	Dec 11	Linear Programming	

Student Assessment

DM579 (7.5 ECTS)

- Theoretical assignment
  (with David, 27/10)
- Three Weekly Python assignments in DM587

7-point grading scale
external censor

DM587 (5 ECTS)

- Six Weekly Python assignments in DM587

7-point grading scale
external censor

Weekly Assignments

- Three/Six weekly Python assignments (aka labs)
- All scored from 0 to 100 (some with extra points). Hence 300/600 points overall.

For DM579:

- You must achieve an average score $> 50\%$ (ie, > 150 points) in the labs to be guaranteed to pass the Applications part
- Final grade: is based on an overall impression. Indicatively, the grade of the labs can only change by at most one grade the grade of the Theoretical part.
- The grade of the Theoretical part will not be unveiled before the end of the third assignment.

For DM587:

- You must achieve an average score $> 50\%$ (ie, 300 points) in the labs to be guaranteed to pass
- Final grade: is based on how many points above 300.

Labs: Practicalities

- Submissions via **git**
  - Read the Appendix A and B
  - Check that your repository exists in `https://git.imada.sdu.dk`
- Specifications file with examples
- Automatic grading after submission up to the deadline
  - You can submit as many times as you wish
  - Only the last grading before the deadline counts
  - But do not submit without passing the local tests or with syntax errors (this will be noted)
  - Assessment at `https://dalila.imada.sdu.dk/`; or in `grades.txt` after `gitpull`
- We will check for plagiarism

Honor Code

By registering to this course, you agree to:

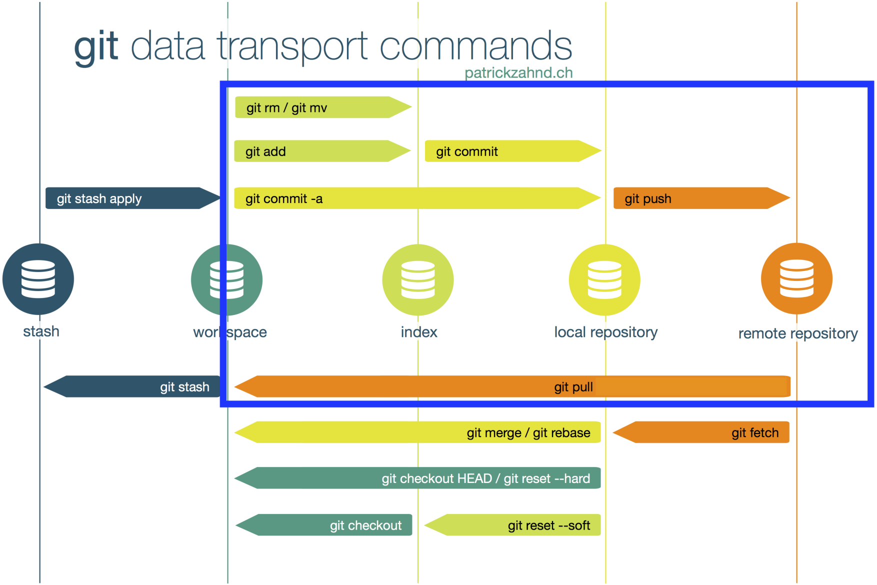
- complete all assignments with your own work;
- acknowledge any and all external sources used in your work;
- refrain from any activity that would dishonestly or fraudulently improve your results or disadvantage others in the course;
- refrain from disclosing answers of assignments to others;
- maintain only one user account and not let anyone else use your username and/or password; and not access or attempt to access any other user's account, or misrepresent or attempt to misrepresent your identity while using the git system.

This Honor Code is not intended to prohibit discussion of course material. While students must submit work that is their own, students should feel free to discuss lectures and exercise sheets or other course material with others either in-person or online.

Structure of the Course

(and similar in weeks 49 and 50)

Week	43	44	45	46	47	48
Man 8-10	f (U20)					
Tir 10-12		f (U23)				
Ons 16-18	f (U42)	asg 0	asg 1		asg 3	
Tor 12-14	h2e (U156)	f (U23) asg 1	f (U23) asg 2	asg 2	f (U23) asg 4	asg 4
Tor 14-16				f (U55) asg 3		f (U42) asg 5
Fre 8-10					h2e (U21)	h2e (U24)
Fre 10-12			h2e (U31)			
Fre 12-14		h2e (U28A)				
Fre 14-16				h2e (U31)		


git data transport commands
patrickzahnd.ch
git rm / git mv
git add
git commit
git stash apply
git commit -a
git push
stash
workspace
index
local repository
remote repository
git stash
git pull
git merge / git rebase
git fetch
git checkout HEAD / git reset --hard
git checkout
git reset --soft

Outline

Running Python — Scripts

A Python script

```
# python_intro.py
"""This is the file header.
The header contains basic information about the file.
"""

if __name__ == "__main__":
    print("Hello, world!\n")    # indent with four spaces (not TAB)
```

- insert in a file with a text editor, for example, emacs, vim, VS code.
- execute from command prompt on Terminal on Linux or Mac and Command Prompt on Windows or Bash Shell on Windows Subsystem for Linux (WSL)

Running Python — Interactively

Python:

```
$ python                             # Start the Python interpreter.
>>> print("This is plain Python.")   # Execute some code.
This is plain Python.
```

IPython:

```
>>> exit()                           # Exit the Python interpreter.
$ ipython                            # Start IPython.

In [1]: print("This is IPython!")    # Execute some code.
This is IPython!

In [2]: %run python_intro.py          # Run a particular Python script.
Hello, world!
```

IPython

- Object introspection: quickly reveals all methods and attributes associated with an object.
- `help()` provides interactive help.

```
# A list is a basic Python data structure. To see the methods associated with
# a list, type the object name (list), followed by a period, and press tab.
In [1]: list.   # Press 'tab'.
              append()  count()   insert()  remove()
              clear()   extend()  mro()     reverse()
              copy()    index()   pop()     sort()

# To learn more about a specific method, use a '?' and hit 'Enter'.
In [1]: list.append?
Docstring: L.append(object) -> None -- append object to end
Type:      method_descriptor

In [2]: help()                          # Start IPython's interactive help utility.
help> list                              # Get documentation on the list class.
Help on class list in module __builtin__:
...
<<help> quit                              # End the interactive help session.
```

Resources

- Use IPython side-by-side with a text editor to test syntax and small code snippets quickly.
- VS Code / Spyder3 / JupyterLab
- Consult the internet with appropriate keywords; eg, stackoverflow.com
- The official Python tutorial:
  `http://docs.python.org/3/tutorial/introduction.html`
- PEP8 - Python Enhancement Proposals style guide:
  `http://www.python.org/dev/peps/pep-0008/`
  pylint: linter, a static code analysis tool `https://www.pylint.org/`

Outline

Arithmetics

- +, -, *, /, **, and // operators.
- ** exponentiation; % modular division.
- underscore character _ is a variable with the value of the previous command's output

```
>>> 12 * 3
36
>>> _ / 4
9.0
```

- Data comparisons like < and > act as expected.
- == operator checks for numerical equality and the <= and >= operators correspond to $\leq$ and $\geq$
- Operators `and`, `or`, and `not` (no need for parenthesis)

```
>>> 3 > 2.99
True
>>> 1.0 <= 1 or 2 > 3
True
>>> 7 == 7 and not 4 < 4
True
```

Variables

Basic types: numbers (integer, float), Boolean, string

Dynamically typed language: does not require to specify data type

```
>>> x = 12                          # Initialize x with the integer 12.
>>> y = 2 * 6                       # Initialize y with the integer 2*6 = 12.
>>> x == y                          # Compare the two variable values.
True

>>> x, y = 2, 4                     # Give both x and y new values in one line.
>>> x == y
False
```

Functions: Syntax

```
>>> def add(x, y):
...     return x + y                 # Indent with four spaces.
```

- mixing tabs and spaces confuses the interpreter and causes problems.
- most text editors set the indentation type to spaces (soft tabs)

Functions are defined with parameters and called with arguments,

```
>>> def area(width, height):         # Define the function.
...     return width * height
...
>>> area(2, 5)                       # Call the function.
10
```

```
>>> def arithmetic(a, b):
...     return a - b, a * b          # Separate return values with commas.
...
>>> x, y = arithmetic(5, 2)          # Unpack the returns into two variables.
>>> print(x, y)
3 10
```

Functions: lambda

The keyword `lambda` is a shortcut for creating one-line functions.

```
# Define the polynomials the usual way using 'def'.
>>> def g(x, y, z):
...     return x + y**2 - z**3

# Equivalently, define the polynomials quickly using 'lambda'.
>>> g = lambda x, y, z: x + y**2 - z**3
```

Functions: Docstrings

```
>>> def add(x, y):
...     """Return the sum of the two inputs.""" # one-liner
...     return x + y

>>>def complex(real=0.0, imag=0.0):
...     """Form a complex number.
...
...     Keyword arguments:
...     real -- the real part (default 0.0)
...     imag -- the imaginary part (default 0.0)
...     """ # multi-liner
...     if imag == 0.0 and real == 0.0:
...         return complex_zero
...     ...

>>> def arithmetic(a, b):
...     """Return the difference and the product of the two inputs."""
...     return a - b, a * b
```

Functions: Returned Values

```
>>> def oops(i):
...     """Increment i (but forget to return anything)."""
...     print(i + 1)
...
>>> def increment(i):
...     """Increment i."""
...     return i + 1
...
>>> x = oops(1999)                      # x contains 'None' since oops()
2000                                    # doesn't have a return statement.
>>> y = increment(1999)                 # However, y contains a value.
>>> print(x, y)
None 2000
```

Functions: Arguments

Arguments are passed to functions based on position or name
Positional arguments must be defined before named arguments.

```
# Correctly define pad() with the named argument after positional arguments.
>>> def pad(a, b, c=0):
...     """Print the arguments, plus a zero if c is not specified."""
...     print(a, b, c)
# Call pad() with 3 positional arguments.
>>> pad(2, 4, 6)
2 4 6
# Call pad() with 3 named arguments. Note the change in order.
>>> pad(b=3, c=5, a=7)
7 3 5
# Call pad() with 2 named arguments, excluding c.
>>> pad(b=1, a=2)
2 1 0
# Call pad() with 1 positional argument and 2 named arguments.
>>> pad(1, c=2, b=3)
1 3 2
```

Functions: Generalized Input

- `*args` is a list of the positional arguments
- `**kwargs` is a dictionary mapping the keywords to their argument.

```
>>> def report(*args, **kwargs):
...     for i, arg in enumerate(args):
...         print("Argument " + str(i) + ":", arg)
...     for key in kwargs:
...         print("Keyword", key, "-->", kwargs[key])
...
>>> report("TK", 421, exceptional=False, missing=True)
Argument 0: TK
Argument 1: 421
Keyword missing --> True
Keyword exceptional --> False
```

Outline

Numerical types

Python has four numerical data types: `int`, `long`, `float`, and `complex`.

```
>>> type(3)          # Numbers without periods are integers.
int

>>> type(3.0)        # Floats have periods (3. is also a float).
float
```

Division:

```
>>> 15 / 4           # Float division performs as expected. (but not in Py 2.7!)
3.75
>>> 15 // 4          # Integer division rounds the result down.
3
>>> 15. // 4
3.0
```

Strings

Strings are created with " or '
To concatenate two or more strings, use the + operator between string variables or literals.

```
>>> str1 = "Hello"  # either single or double quotes.
>>> str2 = 'world'
>>> my_string = str1 + " " + str2 + '!'    # concatenation
>>> my_string
'Hello world!'
```

Slicing

- Strings are arrays of characters. Indexing begins at 0!
- Slicing syntax is `[start:stop:step]`. Defaults: `[0:len():1]`.

```
>>> my_string = "Hello world!"
>>> my_string[4]                # Indexing begins at 0.
'o'
>>> my_string[-1]               # Negative indices count backward from the end.
'!'
# Slice from the 0th to the 5th character (not including the 5th character).
>>> my_string[:5]
'Hello'
# Slice from the 6th character to the end.
>>> my_string[6:]
'world!'
# Slice from the 3rd to the 8th character (not including the 8th character).
>>> my_string[3:8]
'lo wo'
# Get every other character in the string.
>>> my_string[::2]
'Hlowrd'
```

Built-in Types

The built-in data structures:

- tuple, list, set, dict
- `collections` module
- Various built in operations

These are always available:

- all versions of Python
- all operating systems
- all distributions of Python
- you do not need to install any package

Fast development:

- exploring ideas
- building prototypes
- solving one-off problems

If you need performance need to optimize or change language

Tuple

- aka, record, structure, a row in a database: ordered collection of elements
- packing and unpacking (unfolding) values.

```
# Basic usage
record = (val1, val2, val3)
a, b, c = record
val = record[n]
```

```
>>> row = ("Mike", "John", "Mads")
>>> row[1]
"John"
>>> both = arithmetic(5,2) # or get them both as a tuple.
>>> print(both)
(3, 10)
```

Mutable vs Immutable Objects

Immutable Objects: built-in types like `int`, `float`, `bool`, `string`, `tuple`. Objects of these types can't be changed after they are created.

```
message = "Welcome to DM587/DM579"
message[0] = 'p'
print(message)

# Error :
#
# Traceback (most recent call last):
#   File "/home/ff856d3c5411909530c4d328eeca165b.py", line 3, in
#     message[0] = 'p'
# TypeError: 'str' object does not support item assignment
```

```
tuple1 = (0, 1, 2, 3)
tuple1[0] = 4
print(tuple1)

# Error :
#
# Traceback (most recent call last):
#   File "e0eaddff843a8695575daec34506f126.py", line 3, in
#     tuple1[0]=4
# TypeError: 'tuple' object does not support item assignment
```

Mutable Objects: are the following built-in types `list`, `dict`, `set` and custom classes

List

- Mutable sequence, array
- Enforcing order

```
# Basic usage
items = [val1, val2, ..., val3]
x = items[n]
items[n] = x
del items[n]
items.append(value)
items.sort()
items.insert(n, value)
items.remove(value)
items.pop()
```

```
>>> my_list = ["Hello", 93.8, "world", 10]
>>> my_list[0]
'Hello'
>>> my_list[-2]
'world'
>>> my_list[:2]
['Hello', 93.8]
```

List

```
>>> my_list = [1, 2]                # Create a simple list of two integers.
>>> my_list.append(4)               # Append the integer 4 to the end.
>>> my_list.insert(2, 3)            # Insert 3 at location 2.
>>> my_list
[1, 2, 3, 4]
>>> my_list.remove(3)               # Remove 3 from the list.
>>> my_list.pop()                   # Remove (and return) the last entry.
4
>>> my_list
[1, 2]
```

Slicing is also very useful for replacing values in a list.

```
>>> my_list = [10, 20, 30, 40, 50]
>>> my_list[0] = -1
>>> my_list[3:] = [8, 9]
>>> print(my_list)
[-1, 20, 30, 8, 9]
```

List

The `in` operator quickly checks if a given value is in a list (or another iterable, including strings).

```
>>> my_list = [1, 2, 3, 4, 5]
>>> 2 in my_list
True
>>> 6 in my_list
False
>>> 'a' in "xylophone"              # 'in' also works on strings.
False
```

Set

- unordered sequence
- uniqueness, membership test

```
# Basic usage
s = {val1, val2, ..., valn↪
    ↪}
s.add(val)
s.discard(val)
s.remove(val) # throws ↪
    ↪exception if the ↪
    ↪element is not ↪
    ↪present in the set
val in s
s.union({val})
s.intersection({val})
s.difference({val})
s.symmetric_difference({↪
    ↪val})
```

```
# Initialize some sets. Repeats are not added.
>>> gym_members = {"John", "John", "Jane", "Bob"}
>>> print(gym_members)
{'John', 'Bob', 'Jane'}

>>> gym_members.add("Josh")
>>> gym_members.discard("John")
>>> print(gym_members)
{'Josh', 'Bob', 'Jane'}

>>> gym_members.intersection({"Josh", "Ian", "↪
    ↪Jared"})
{'Josh'}
>>> gym_members.difference({"Bob", "Sarah"})
{'Josh', 'Jane'}
```

Dict

- mapping, associative (key,value) array (implemented as a hash table)
- unordered
- lookup table, indices, key values need to be immutable

```python
# Basic usage
d = { key1: val1, key2: ↪
    ↪val2, key3: val3 }
val = d[key]
d[key] = val
del d[key]
key in d
d.keys()
d.values()
d.pop(key)
d.items()
```

```python
>>> my_dictionary = {"business": 4121, "math": ↪
    ↪2061, "visual arts": 7321}
>>> print(my_dictionary["math"])
2061

>>> my_dictionary["science"] = 6284
>>> my_dictionary.pop("business")
4121
>>> print(my_dictionary)
{'math': 2061, 'visual arts': 7321, 'science': ↪
    ↪6284}

>>> my_dictionary.keys()
dict_keys(['math', 'visual arts', 'science'])
>>> my_dictionary.values()
dict_values([2061, 7321, 6284])
```

Further Collections


```
>>> from collections import namedtuple
>>> Person = namedtuple('Person', ['first','last','address'])
>>> row = Person('Marco','Chiarandini','Campusvej')
>>> row.first
'Marco'
```

```
>>> from collections import Counter # histograms
>>> c = Counter('xyzzzy')
>>> c
Counter({'x': 1, 'y': 2, 'z': 3})
```

```
>>> from collections import defaultdict  # multidict, one-many relationships
>>> d = defaultdict(list)
>>> d['spam'].append(42)
>>> d['blah'].append(13)
>>> d['spam'].append(10)
>>> d
{'blah': [42], 'spam': [13, 10]}
```

Further Collections

```
>>> from collections import OrderedDict  # remembers the order entries were ↪
    ↪added
>>> # regular unsorted dictionary
>>> d = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}

>>> # dictionary sorted by key
>>> OrderedDict(sorted(d.items(), key=lambda t: t[0]))
OrderedDict([('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)])

>>> # dictionary sorted by value
>>> OrderedDict(sorted(d.items(), key=lambda t: t[1]))
OrderedDict([('pear', 1), ('orange', 2), ('banana', 3), ('apple', 4)])

>>> # dictionary sorted by length of the key string
>>> OrderedDict(sorted(d.items(), key=lambda t: len(t[0])))
OrderedDict([('pear', 1), ('apple', 4), ('orange', 2), ('banana', 3)])
```

Type Casting

```
# Cast numerical values as different kinds of numerical values.
>>> x = int(3.0)
>>> y = float(3)

# Cast a list as a set and vice versa.
>>> set([1, 2, 3, 4, 4])
{1, 2, 3, 4}
>>> list({'a', 'a', 'b', 'b', 'c'})
['a', 'c', 'b']

# Cast other objects as strings.
>>> str(['a', str(1), 'b', float(2)])
"['a', '1', 'b', 2.0]"
```

Outline

The If Statement

```
>>> food = "bagel"
>>> if food == "apple":                 # As with functions, the colon denotes
...     print("72 calories")            # the start of each code block.
... elif food == "banana" or food == "carrot":
...     print("105 calories")
... else:
...     print("calorie count unavailable")
...
calorie count unavailable
```

The While Loop

```
>>> i = 0
>>> while True: # i < 10
...    print(i, end=' ')
...    i += 1
...    if i >= 10:
...        break                    # Exit the loop.
...
0 1 2 3 4 5 6 7 8 9

>>> i = 0
>>> while i < 10:
...     i += 1
...     if i % 3 == 0:
...         continue                # Skip multiples of 3.
...     print(i, end=' ')
1 2 4 5 7 8 10
```

The For Loop

- A `for` loop iterates over the items in any iterable.
- Iterables include (but are not limited to) strings, lists, sets, and dictionaries.

```
>>> colors = ["red", "green", "blue", "yellow"]
>>> for entry in colors:
...     print(entry + "!")
...
red!
green!
blue!
yellow!
```

- The `break` and `continue` statements also work in for loops
- but a `continue` in a for loop will automatically increment the index or item

Built-in Functions

1. `range(start, stop, step)`: Produces a sequence of integers, following slicing syntax.
2. `zip()`: Joins multiple sequences so they can be iterated over simultaneously.
3. `enumerate()`: Yields both a count and a value from the sequence. Typically used to get both the index of an item and the actual item simultaneously.
4. `reversed()`: Reverses the order of the iteration.
5. `sorted()`: Returns a new list of sorted items that can then be used for iteration.

```
# Iterate through the list in sorted (alphabetical) order.
>>> for item in sorted(colors):
...     print(item, end=' ')
...
blue purple red white yellow
```

They (except for `sorted()`) are generators and return an iterator.
To put the items of the sequence in a collection, use `list()`, `set()`, or `tuple()`.

List Comprehension


```
>>> loop_output = []
>>> for i in range(5):
...     loop_output.append(i**2)
...
>>> list_output = [i**2 for i in range(5)]
```

```
[ expression for x in iterable if conditions ] # list
{ expression for x in iterable if conditions } # set
{ key: val for key, val in iterable if conditions } # dict
```

```
>>> colors = ["red", "blue", "yellow"]
>>> {"bright " + c for c in colors}
{'bright blue', 'bright red', 'bright yellow'}

>>> {c[0]:c for c in colors}
{'y': 'yellow', 'r': 'red', 'b': 'blue'}
```

Generators

```
( expression for x in iterable if condition )
```

```
>>> nums = [1, 2, 3, 4, 5, 6]
>>> squares = (i*i for i in nums)
>>> squares
<generator object <genexpr> at 0x110468200>
>>> for n in squares:
    print(n)
1
4
9
16
25
36
```

Decorators — Function Wrappers

```
>>> def typewriter(func):
...     """Decorator for printing the type of output a function returns"""
...     def wrapper(*args, **kwargs):
...         output = func(*args, **kwargs)       # Call the decorated function.
...         print("output type:", type(output)) # Process before finishing.
...         return output                        # Return the function output.
...     return wrapper
```

```
>>> combine = typewriter(combine)
```

```
>>> @typewriter
... def combine(a, b, c):
...     return a*b // c
```

Now calling `combine()` actually calls `wrapper()`, which then calls the original `combine()`.

```
>>> combine(3, 4, 6)
output type: <class 'int'>
2
>>> combine(3.0, 4, 6)
output type: <class 'float'>
2.0
```

Decorators — Function Wrappers

```
>>> def repeat(times):
...     """Decorator for calling a function several times."""
...     def decorator(func):
...         def wrapper(*args, **kwargs):
...             for _ in range(times):
...                 output = func(*args, **kwargs)
...             return output
...         return wrapper
...     return decorator
...
>>> @repeat(3)
... def hello_world():
...     print("Hello, world!")
...
>>> hello_world()
Hello, world!
Hello, world!
Hello, world!
```

Outline

Built-in Functions

Common built-in functions for numerical calculations:

Function	Returns
`input()`	Gets input from console
`abs()`	The absolute value of a real number, or the magnitude of a complex number.
`min()`	The smallest element of a single iterable, or the smallest of several arguments. Strings are compared based on lexicographical order: numerical characters first, then upper-case letters, then lower-case letters.
`max()`	The largest element of a single iterable, or the largest of several arguments.
`len()`	The number of items of a sequence or collection.
`round()`	A float rounded to a given precision in decimal digits.
`sum()`	The sum of a sequence of numbers.

See `https://docs.python.org/3/library/functions.html` for more detailed documentation on all of Python's built-in functions.

Built-in Functions

Function	Description
`all()`	Return `True` if `bool(entry)` evaluates to `True` for every entry in the input iterable.
`any()`	Return `True` if `bool(entry)` evaluates to `True` for any entry in the input iterable.
`bool()`	Evaluate a single input object as `True` or `False`.
`eval()`	Execute a string as Python code and return the output.
`map()`	Apply a function to every item of the input iterable and return an iterable of the results.
`filter()`	Apply a filter to the elements of the input iterable and return an set.

Modules

- A module is a Python file containing code that is meant to be used in some other setting
- All `import` statements should occur at the top of the file, below the header but before any other code.

1. `import <module>` makes the specified module available under the alias of its own name.

```
>>> import math                    # The name 'math' now gives
>>> math.sqrt(2)                   # access to the math module.
1.4142135623730951
```

2. `import <module> as <name>` creates an alias for an imported module. The alias is added to the current namespace, but the module name itself is not.

```
>>> import numpy as np             # The name 'np' gives access to the numpy
>>> np.sqrt(2)                     # module, but the name 'numpy' does not.
1.4142135623730951
>>> numpy.sqrt(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'numpy' is not defined
```

Modules

3. `from <module> import <<<object>>>` loads the specified object into the namespace without loading anything else in the module or the module name itself. This is used most often to access specific functions from a module. The `as` statement can also be tacked on to create an alias.

```
>>> from random import randint  # The name 'randint' gives access to the
>>> r = randint(0, 10000)       # randint() function, but the rest of
>>> random.seed(r)              # the random module is unavailable.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'random' is not defined
```

Running and Importing

```
# example1.py

data = list(range(4))
def display():
    print("Data:", data)

if __name__ == "__main__":
    display()
    print("This file was executed from the command line or an interpreter.")
else:
    print("This file was imported.")
```

```
$ python example1.py
Data: [0, 1, 2, 3]
This file was executed from the command line or an interpreter.
```

The Python Standard Library

Module	Description
`cmath`	Mathematical functions for complex numbers.
`csv`	Comma Separated Value (CSV) file parsing and writing.
`itertools`	Tools for iterating through sequences in useful ways.
`math`	Standard mathematical functions and constants.
`os`	Tools for interacting with the operating system.
`sys`	Tools for interacting with the interpreter.
`string`	Common string literals.
`random`	Random variable generators.
`functools`	Tools for higher-order functions: functions that act on or return other functions.
`time`	Time value generation and manipulation.
`timeit`	Measuring execution time of small code snippets.

Explore the documentation in IPython

Python Packages

- A package is simply a folder that contains a file called `__init__.py`.
- This file is always executed first whenever the package is used.
- A package must also have a file called `__main__.py` in order to be executable.
- Executing the package will run `__init__.py` and then `__main__.py`
- Importing the package will only run `__init__.py`

- Use `from <subpackage.module> import <<<object>>>` to load a module within a subpackage.
- Once a name has been loaded into a package's `__init__.py`, other files in the same package can load the same name with `from . import <<<object>>>`.

To execute a package, run Python from the shell with the flag `-m` (for "module-name") and exclude the extension `.py`.

```
$ python -m package_name
```

See https://docs.python.org/3/tutorial/modules.html#packages for examples and more details.

Mutable vs Immutable Objects

- a mutable object can be changed after it is created, and an immutable object can't.
- Objects of built-in types like (int, float, bool, str, tuple, unicode) are immutable
- Objects of built-in types like (list, set, dict) are mutable

```
>>> x = "Holberton"
>>> y = "Holberton"
>>> id(x)
140135852055856
>>> id(y)
140135852055856
>>> print(x is y) '''comparing the types'''
True

>>> a = 50
>>> type(a)
<class: 'int'>
>>> b = "Holberton"
>>> type(b)
<class: 'string'>
```

Mutable vs Immutable Objects

```
>>> holy = {"moly": 1.99, "hand_grenade": 3, "grail": 1975.41}
>>> tax_prices = holy                    # Try to make a copy for processing.
>>> for item, price in tax_prices.items():
...     # Add a 7 percent tax, rounded to the nearest cent.
...     tax_prices[item] = round(1.07 * price, 2)
...
# Now the base prices have been updated to the total price.
>>> print(tax_prices)
{'moly': 2.13, 'hand_grenade': 3.21, 'grail': 2113.69}

# However, dictionaries are mutable, so 'holy' and 'tax_prices' actually
# refer to the same object. The original base prices have been lost.
>>> print(holy)
{'moly': 2.13, 'hand_grenade': 3.21, 'grail': 2113.69}
```

To avoid this problem, explicitly create a copy of the object by casting it as a new structure.

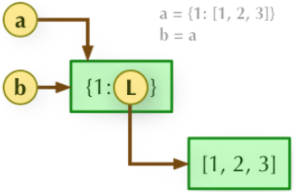
```
>>> tax_prices = dict(holy)
```

Pass by-value or by-reference

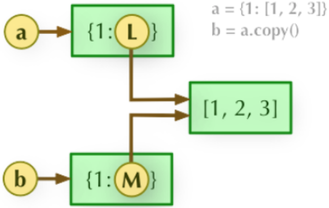
- A pointer refers to a variable by storing the address in memory where the corresponding object is stored.
- Python names are essentially pointers, and traditional pointer operations and cleanup are done automatically.
- Python automatically deletes objects in memory that have no names assigned to them (no pointers referring to them). This feature is called garbage collection.
- All objects that are arguments of functions are passed by reference

Shallow vs Deep Copy

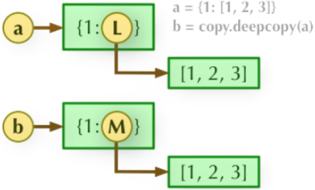
1. `b = a`: Reference assignment, Make `a` and `b` points to the same object.



2. `b = a.copy()`: Shallow copying, `a` and `b` will become two isolated objects, but their contents still share the same reference



3. `b = copy.deepcopy(a)`: Deep copying, `a` and `b`'s structure and content become completely isolated.



Outline

Classes

A class is a template for an object that binds together specified variables and routines.

```python
class Backpack:
    """A Backpack object class. Has a name and a list of contents.

    Attributes:
        name (str): the name of the backpack's owner.
        contents (list): the contents of the backpack.
    """
    def __init__(self, name):           # This function is the constructor.
        """Set the name and initialize an empty list of contents.

        Parameters:
            name (str): the name of the backpack's owner.
        """
        self.name = name                # Initialize some attributes.
        self.contents = []
```

Instantiation


```
# Import the Backpack class and instantiate an object called 'my_backpack'.
>>> from object_oriented import Backpack
>>> my_backpack = Backpack("Fred")
>>> type(my_backpack)
<class 'object_oriented.Backpack'>

# Access the object's attributes with a period and the attribute name.
>>> print(my_backpack.name, my_backpack.contents)
Fred []

# The object's attributes can be modified after instantiation.
>>> my_backpack.name = "George"
>>> print(my_backpack.name, my_backpack.contents)
George []
```

Methods

```python
class Backpack:
    # ...
    def put(self, item):
        """Add an item to the backpack's list of contents."""
        self.contents.append(item)  # Use 'self.contents', not just 'contents'.

    def take(self, item):
        """Remove an item from the backpack's list of contents."""
        self.contents.remove(item)
```

```python
>>> my_backpack.put("notebook")        # my_backpack is passed implicitly to
>>> my_backpack.put("pencils")         # Backpack.put() as the first argument.
>>> my_backpack.contents
['notebook', 'pencils']

# Remove an item from the backpack.    # This is equivalent to
>>> my_backpack.take("pencils")        # Backpack.take(my_backpack, "pencils")
>>> my_backpack.contents
['notebook']
```

Inheritance

Superclass ⇝ Subclass

```
class Knapsack(Backpack): # Inherit from the Backpack class in the class ↪
    ↪definition
    """Attributes:
        name (str): the name of the knapsack's owner.
        color (str): the color of the knapsack.
        max_size (int): the maximum number of items that can fit inside.
        contents (list): the contents of the backpack.
        closed (bool): whether or not the knapsack is tied shut.
    """
    def __init__(self, name, color, max_size=3):
        """Use the Backpack constructor to initialize the name, color,
        and max_size attributes. A knapsack only holds 3 items by default.
        """
        Backpack.__init__(self, name, color, max_size)
        self.closed = True
```

Inheritance

- all methods defined in the superclass class are available to instances of the subclass.
- methods from the superclass can be changed for the subclass by override
- New methods can be included normally.

```
>>> from object_oriented import Knapsack
>>> my_knapsack = Knapsack("Brady", "brown")

# A Knapsack is a Backpack, but a Backpack is not a Knapsack.
>>> print(issubclass(Knapsack, Backpack), issubclass(Backpack, Knapsack))
True False
>>> isinstance(my_knapsack, Knapsack) and isinstance(my_knapsack, Backpack)
True

# The Knapsack class has a weight() method, but the Backpack class does not.
>>> print(hasattr(my_knapsack, 'weight'), hasattr(my_backpack, 'weight'))
True False
```

Magic Methods

- special methods used to make an object behave like a built-in data type.
- begin and end with two underscores, like the constructor `__init__()`.
- all variables and routines of a class are public
- magic methods are hidden

```
In [1]: %run object_oriented.py
In [2]: b = Backpack("Oscar", "green")
In [3]: b.          # Press 'tab' to see standard methods and attributes.
          color    max_size take()
          contents name
          dump()   put()
In [3]: b.__        # Press 'tab' to see magic methods and hidden attributes.
          __getattribute__   _ _new__()             __class__
          __delattr__         __hash__               __reduce_ex__()
          __dict__            __init__()             __repr__
          __dir__()           __init_subclass__()    __setattr__
          __doc__             __sizeof__()           __reduce__()
          __str__             __format__()           __module__
          __subclasshook__()  __weakref__
```

Magic Methods

Method	Arithmetic Operator
__add__()	+
__sub__()	-
__mul__()	*
__pow__()	**
__truediv__()	/
__floordiv__()	//

Method	Comparison Operator
__lt__()	<
__le__()	<=
__gt__()	>
__ge__()	>=
__eq__()	==
__ne__()	=

Operator overloading:

```python
class Backpack:
    def __add__(self, other):
        return len(self.contents) + len(other.contents)
```

```python
class Backpack(object)
    def __lt__(self, other):
        return len(self.contents) < len(other.contents)
```

Static Attributes and Methods

Static attributes and methods are defined without `self` and can be accessed both with and without instantiation

```
class Backpack:
    # ...
    brand = "Adidas"                # Backpack.brand is a static attribute.
```

```
class Backpack:
    # ...
    @staticmethod
    def origin():                   # Do not use 'self' as a parameter.
        print("Manufactured by " + Backpack.brand + ", inc.")
```

More Magic Methods and Hashing

Method	Operation	Trigger Function
`__bool__()`	Truth value	`bool()`
`__len__()`	Object length or size	`len()`
`__repr__()`	Object representation	`repr()`
`__getitem__()`	Indexing and slicing	`self[index]`
`__setitem__()`	Assignment via indexing	`self[index] = x`
`__iter__()`	Iteration over the object	`iter()`
`__reversed__()`	Reverse iteration over the object	`reversed()`
`__contains__()`	Membership testing	`in`

A hash value is an integer that uniquely identifies an object.
If the `__hash__()` method is not defined, the default hash value is the object's memory address (accessible via the built-in function `id()`) divided by 16, rounded down to the nearest integer.

```
class Backpack:
    def __hash__(self):
        return hash(self.name) ^ hash(self.color) ^ hash(len(self.contents))
```

Outline

Type Annotations

- Statically typed: type checking at compile-time; requires datatype declarations.
- Dynamically typed: type checking at runtime; does not require datatype declarations.

Python is dynamically typed. Type annotations are used to indicate the datatypes of variables and input/outputs of functions and methods.

```python
# This is how you declare the type of a variable type in Python 3.6
age: int = 1

# You don't need to initialize a variable to annotate it
a: int  # Ok (no value at runtime until assigned)

# The latter is useful in conditional branches
child: bool
if age < 18:
    child = True
else:
    child = False
```

Built-in Types (I)

```python
# For simple built-in types, just use the name of the type
x: int = 1
x: float = 1.0
x: bool = True
x: str = "test"
x: bytes = b"test"

# For collections, the type of the collection item is in brackets
# (Python 3.9+)
x: list[int] = [1]
x: set[int] = {6, 7}
# In Python 3.8 and earlier, the name of the collection type is
# capitalized, and the type is imported from the 'typing' module
from typing import List, Set, Dict, Tuple, Optional
x: List[int] = [1]
x: Set[int] = {6, 7}
```

Built-in Types (II)

```python
from typing import List, Set, Dict, Tuple, Optional
# For mappings, we need the types of both keys and values
x: dict[str, float] = {"field": 2.0}  # Python 3.9+
x: Dict[str, float] = {"field": 2.0}

# For tuples of fixed size, we specify the types of all the elements
x: tuple[int, str, float] = (3, "yes", 7.5)  # Python 3.9+
x: Tuple[int, str, float] = (3, "yes", 7.5)

# For tuples of variable size, we use one type and ellipsis
x: tuple[int, ...] = (1, 2, 3)  # Python 3.9+
x: Tuple[int, ...] = (1, 2, 3)

# Use Optional[] for values that could be None
x: Optional[str] = some_function()
```

Types in Functions

```
def add(x: int, y: int) -> int:
    return x + y
```

To allow multiple datatypes, we can use type union operators. Pre-Python 3.10 this would look like:

```
from typing import Union
def add(x: Union[int, float], y: Union[int, float]) -> Union[int, float]:
    return x + y
```

Here, we allow either int or float datatypes!
With Python 3.10, we can replace Union with the new union operator |:

```
def add(x: int | float, y: int | float) -> int | float:
    return x + y
```

Type Checking

Python does not check types but IDE do and can provide warnings. From command line we can use the `mypy` tool

```
pip install mypy
mypy main.py
```

```python
# Mypy understands a value can't be None in an if-statement
if x is not None:
    print(x.upper())
# If a value can never be None due to some invariants, use an assert
assert x is not None
print(x.upper())
```

Summary