SP01_l1_23-Python

November 4, 2023

1 Podstawy Pythona

- Python jest językiem **interpretowanym** - interpreter Pythona wykonuje program, uruchamiając kolejne instrukcje kodu
- Do przetwarzania danych za pomocą Pythona nie trzeba pisać programów/aplikacji w Pythonie - wystarczy skrypt

1.1 Semantyka

- Wcięcia zamiast nawiasów - zazwyczaj 4 spacje
- Instrukcje nie muszą kończyć się średnikiem
- Średnik rozdziela wiele instrukcji zapisanych w jednej linii (niezalecane - słaba czytelność kodu)
- Wielkość liter ma znaczenie (case-sensitive)
- Komentarze jednoliniowe rozpoczynają się od # - zazwyczaj pojawiają się w oddzielnej linii, przed opisywanym kodem

[1]:
```
x = 1
y = 2
if x > 3:
    x += 4
    y = 5
y
```

[1]:
```
2
```

Użycie odwołania do obiektu prezentuje informacje na temat tego obiektu.
Zazwyczaj są one lepiej sformatowane niż uzyskane za pomocą funkcji `print`

1.2 Model obiektowy

Każda wartość, każda funkcja, klasa czy moduł jest obiektem.

Mają one informacje o typie, metody i wewnętrzne dane.

[2]:
```
a = [1, 2, 3]
type(a)
```

[2]:
```
list
```

[3]:
```
# Wykorzystanie ? do przeszukiwania przestrzeni nazw powłoki
a.*?
```

```
a.__add__
a.__class__
a.__class_getitem__
a.__contains__
a.__delattr__
a.__delitem__
a.__dir__
a.__doc__
a.__eq__
a.__format__
a.__ge__
a.__getattribute__
a.__getitem__
a.__gt__
a.__hash__
a.__iadd__
a.__imul__
a.__init__
a.__init_subclass__
a.__iter__
a.__le__
a.__len__
a.__lt__
a.__mul__
a.__ne__
a.__new__
a.__reduce__
a.__reduce_ex__
a.__repr__
a.__reversed__
a.__rmul__
a.__setattr__
a.__setitem__
a.__sizeof__
a.__str__
a.__subclasshook__
a.append
a.clear
a.copy
a.count
a.extend
a.index
a.insert
a.pop
```

```
a.remove
a.reverse
a.sort
```

[4]:
```
import numpy as np
```

[5]:
```
type(np)
```

[5]:
```
module
```

[6]:
```
# Introspekcja - wybrane ogólne informacje o obiekcie
np?
```

```
Type:        module
String form: <module 'numpy' from '/opt/conda/lib/python3.10/site-packages/numpy/
 ↪__init__.py'>
File:        /opt/conda/lib/python3.10/site-packages/numpy/__init__.py
Docstring:
NumPy
=====

Provides
  1. An array object of arbitrary homogeneous items
  2. Fast mathematical operations over arrays
  3. Linear Algebra, Fourier Transforms, Random Number Generation

How to use the documentation
----------------------------
Documentation is available in two forms: docstrings provided
with the code, and a loose standing reference guide, available from
`the NumPy homepage <https://www.scipy.org>`_.

We recommend exploring the docstrings using
`IPython <https://ipython.org>`_, an advanced Python shell with
TAB-completion and introspection capabilities.  See below for further
instructions.

The docstring examples assume that `numpy` has been imported as `np`::

  >>> import numpy as np

Code snippets are indicated by three greater-than signs::

  >>> x = 42
  >>> x = x + 1

Use the built-in ``help`` function to view a function's docstring::
```

```
  >>> help(np.sort)
  … # doctest: +SKIP

For some objects, ``np.info(obj)`` may provide additional help.  This is
particularly true if you see the line "Help on ufunc object:" at the top
of the help() page.  Ufuncs are implemented in C, not Python, for speed.
The native Python help() does not know how to view their help, but our
np.info() function does.

To search for documents containing a keyword, do::

  >>> np.lookfor('keyword')
  … # doctest: +SKIP

General-purpose documents like a glossary and help on the basic concepts
of numpy are available under the ``doc`` sub-module::

  >>> from numpy import doc
  >>> help(doc)
  … # doctest: +SKIP

Available subpackages
---------------------
doc
    Topical documentation on broadcasting, indexing, etc.
lib
    Basic functions used by several sub-packages.
random
    Core Random Tools
linalg
    Core Linear Algebra Tools
fft
    Core FFT routines
polynomial
    Polynomial tools
testing
    NumPy testing tools
f2py
    Fortran to Python Interface Generator.
distutils
    Enhancements to distutils with support for
    Fortran compilers support and more.

Utilities
---------
test
    Run numpy unittests
show_config
```

```
    Show numpy build configuration
dual
    Overwrite certain functions with high-performance SciPy tools.
    Note: `numpy.dual` is deprecated.  Use the functions from NumPy or Scipy
    directly instead of importing them from `numpy.dual`.
matlib
    Make everything matrices.
__version__
    NumPy version string

Viewing documentation using IPython
-----------------------------------
Start IPython with the NumPy profile (``ipython -p numpy``), which will
import `numpy` under the alias `np`.  Then, use the ``cpaste`` command to
paste examples into the shell.  To see which functions are available in
`numpy`, type ``np.<TAB>`` (where ``<TAB>`` refers to the TAB key), or use
``np.*cos*?<ENTER>`` (where ``<ENTER>`` refers to the ENTER key) to narrow
down the list.  To view the docstring for a function, use
``np.cos?<ENTER>`` (to view the docstring) and ``np.cos??<ENTER>`` (to view
the source code).

Copies vs. in-place operation
-----------------------------
Most of the functions in `numpy` return a copy of the array argument
(e.g., `np.sort`).  In-place versions of these functions are often
available as array methods, i.e. ``x = np.array([1,2,3]); x.sort()``.
Exceptions to this rule are documented.
```

1.3 Typy

Python jest językiem silnie typowanym. Każdy obiekt ma swój typ i zna swój typ.
Nie ma możliwości zmiany typu przez obiekt (przypadkowo czy celowo).

Poniższa konstrukcja nie zmienia typów obiektów.

[7]:
```
a = 1
a = "a"
```

Python, w odróżnieniu do innych języków, nie posiada zmiennych, które posiadają zarówno typ jak i nazwę.

Python posiada *nazwy*, które wskazują na *obiekty* (mające swój typ).
Powyżej zmieniliśmy jedynie to na co wskazuje nazwa `a`.

Przypisywanie nazw do innych nazw tworzy odwołanie do tego samego obiektu (w innych językach byłoby to powiązane z kopiowaniem).
Przypisywanie w Pythonie określane jest mianem *wiązania* (nazwy z obiektem).

[8]:
```
a = ["a", "b", "c"]
b = a
a.append("d")
b
```

[8]:
```
['a', 'b', 'c', 'd']
```

Analogiczna sytuacja ma miejsce w przypadku argumentów funkcji. Nazwy (zmienne) lokalne odwoływują się do oryginalnych obiektów.

[9]:
```
def add_item(l, item):
    l.append(item)

add_item(a, "e")
b
```

[9]:
```
['a', 'b', 'c', 'd', 'e']
```

Aby nie wprowadzać za dużo zamieszania z nazwami i zmiennymi (lub ich brakiem w Pythonie), przyjmuje się, że:

- **zmienne** są nazwami obiektów znajdujących się w określonej przestrzeni nazw, natomiast
- **informacja o typie** obiektu jest zapisana w samym obiekcie.

Do sprawdzenie czy dwie zmienne są tym samym obiektem (wskazują na ten sam obiekt) można wykorzystać operator `is`.
Funkcja `id` pozwala uzyskać identyfikator obiektu.
Do porównania wartości obiektów służy operator `==`.

[10]:
```
a is b
```

[10]:
```
True
```

[11]:
```
print(id(a), "is the same as", id(b))
```

```
140087542548736 is the same as 140087542548736
```

[12]:
```
# funkcja list tworzy nową listę (kopię)
c = list(a)
c
```

[12]:
```
['a', 'b', 'c', 'd', 'e']
```

[13]:
```
a is c
```

[13]:
```
False
```

[14]:
```
print(id(a), "is not the same as", id(c))
```

```
140087542548736 is not the same as 140086518697664
```

[15]:
```
a == c
```

[15]:
```
True
```

1.4 Obiekty modyfikowalne i niemodyfikowalne

Większość typów w Pythonie jest modyfikowalna (*mutable*). Obiekty takich typów mogą zmieniać swoje wartości - zmienne (nazwy) nie mają nic do tego.
Przykłady:

- listy (`list`)
- zbiory (`set`)
- słowniki (`dict`)

Są jednak takie typy, które są niemodyfikowalne (*immutable*).
Przykłady:

- `bool`
- `int`
- `float`
- `str`
- `tuple`
- `frozenset`

[16]:
```
x = 1
y = 1
x is y
```

[16]:
```
True
```

[17]:
```
x += 1
print(id(x), "is not the same as", id(y))
```

```
140087589798160 is not the same as 140087589798128
```

[18]:
```
# immutable?
t = (1, 2, [3, 4, 5])
t[2].append(6)
t
```

[18]:
```
(1, 2, [3, 4, 5, 6])
```

[19]:
```
t[1] = "a"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [19], in <cell line: 1>()
```

```
----> 1 t[1] = "a"

TypeError: 'tuple' object does not support item assignment
```

1.5 Wywoływanie funkcji i metod

Funkcje mogą być wywoływane za pomocą nawiasów okrągłych.

Nagłówki funkcji/metod mogą określać dla swoich parametrów wartości domyślne. Parametry bez wartości domyślnych nie mogą występować po parametrach z wartościami domyślnymi.

Wartości parametrów mogą być przekazywane za pomocą pozycji lub za pomocą nazwy.

Od wersji 3.5 Pythona istnieje możliwość sugerowania typów obiektów przekazywanych jako parametry oraz typów obiektów zwracanych przez funkcje.

[20]:
```
def ducklist(item: int=2, how_many:int=10) -> list:
    a = []
    for _ in range(how_many):
        a.append(item)
    return a
```

[21]:
```
# Introspekcja
ducklist?
```

```
Signature: ducklist(item: int = 2, how_many: int = 10) -> list
Docstring: <no docstring>
File:      /tmp/ipykernel_16486/1123748225.py
Type:      function
```

[22]:
```
c = ducklist("a", 6)
c
```

[22]:
```
['a', 'a', 'a', 'a', 'a', 'a']
```

[23]:
```
d = ducklist(how_many=2)
d
```

[23]:
```
[2, 2]
```

2 Przepływ sterowania

2.1 `if`, `elif`, `else`

Kolejność warunków złożonych ma znaczenie. Przykładowo w wyrażeniu `if 2>1 or 3<2:`, drugi warunek nie będzie sprawdzany.

[24]:
```
if 1>2:
    a = '1>2'
elif 2>3:
    a = '2>3'
else:
    a = 'else'

a
```

[24]:
```
'else'
```

2.2 `for`

Służy do iterowania po zbiorze, liście lub krotce.
Wyjście z pętli możliwe jest za pomocą instrukcji `break` (przerywa najbardziej wewnętrzna pętlę). Przerwanie wykonania bieżącej iteracji i rozpoczęcie wykonania następnej możliwe jest za pomocą `continue`

Funkcją, która często jest wykorzystywana przy okazji pętli `for` jest `range(start,stop[,step])`. Tworzy ona iterator będący ciągiem wartości całkowitoliczbowych o równych odstępach np. `range(5, 0, -1)`

[25]:
```
# abc - Abstract Base Classes
from collections.abc import Iterable
t = (1, 2, [3, 4, 5], range(5, 0, -1))
s = 0
for elem in t:
    if isinstance(elem, Iterable):
        for e in elem:
            print(e);
            if e >= 4:
                continue
            else:
                pass #konieczne bo else musi mieć instrukcję - "nope"
            s += e
    else:
        print(elem);
        s += elem
s
```

```
1
2
3
4
5
5
4
3
```

```
2
1
```

[25]: 12

2.3 while

[26]:
```
i = ord('a')
str1 = ""
while len(str1)<10:
    str1 += chr(i)
    i += 1
str1
```

[26]: 'abcdefghij'

2.4 wyrażenia trójargumentowe

Pozwalają (w dość nieczytelny) połączyć w jedną linię intrukcję `if else` zwracającą wartość.
`wartość = wyrażenie_true if warunek else wyrażenie_false`

[27]:
```
a = '1>2' if 1>2 else '2>3' if 2>3 else 'else'
a
```

[27]: 'else'

3 Najważniejsze typy proste

Przykłady:

- `NoneType` - z jedyną instancją `None`
- `str` - kodowanie UTF-8
- `bytes` - surowe bajty ASCII
- `float` - precyzja 64 bity
- `bool` - dwie instancje `True` i `False`
- `int`

3.1 str

[28]:
```
a = 'abcd'
b = "abcd"
c = "ab'cd"
d = """a
b
c
d"""
e = "abc\nd"
f = r"abc\nd"
```

```
g = "{0:s} as str, {1:.2f} as float, {2:d} as digit"
h = "Poznań"
```

[29]:
```
d.count("\n")
```

[29]: 3

[30]:
```
list(c)
```

[30]: ['a', 'b', "'", 'c', 'd']

[31]:
```
# rozdzielanie - dotyczy sekwencyjnych typów danych
b[:2]
```

[31]: 'ab'

[32]:
```
e + f
```

[32]: 'abc\ndabc\\nd'

[33]:
```
d
```

[33]: 'a\nb\nc\nd'

[34]:
```
g.format("Hola",3.1415,3)
```

[34]: 'Hola as str, 3.14 as float, 3 as digit'

3.2 bytes

[35]:
```
h_utf8 = h.encode('utf-8')
```

[36]:
```
type(h_utf8)
```

[36]: bytes

[37]:
```
h_utf8
```

[37]: b'Pozna\xc5\x84'

[38]:
```
h2_byte = b'Pozna\xc5\x84 - tylko ASCII'
```

[39]:
```
h2 = h2_byte.decode('utf-8')
h2
```

[39]: 'Poznań - tylko ASCII'

[40]:
```
type(h2)
```

[40]:
```
str
```

3.3 Daty i czas

Do obsługi daty i czasu wykorzystywany jest wbudowany moduł `datetime`

[41]:
```
from datetime import datetime, date, time
```

[42]:
```
dt = datetime(2022, 1, 26, 12, 46, 45)
```

[43]:
```
dt.strftime('%Y-%m-%d %H:%M')
```

[43]:
```
'2022-01-26 12:46'
```

[44]:
```
dt.replace(minute=0, second=0)
```

[44]:
```
datetime.datetime(2022, 1, 26, 12, 0)
```

[45]:
```
delta = datetime.now() - dt
delta
```

[45]:
```
datetime.timedelta(days=647, seconds=3423, microseconds=370972)
```

[46]:
```
delta.days
```

[46]:
```
647
```

4 Najważniejsze typy złożone

4.1 Krotki

- Niemodyfikowalna sekwencja obiektów o stałej długości.

4.1.1 Tworzenie

[47]:
```
t1 = 1, 2, 3, 'a'
t1
```

[47]:
```
(1, 2, 3, 'a')
```

[48]:
```
t2 = (1, 2, 3), ("a", "b", "c")
t2
```

[48]:
```
((1, 2, 3), ('a', 'b', 'c'))
```

[49]:
```
t3 = tuple("1234abcd")
t3
```

[49]:
```
('1', '2', '3', '4', 'a', 'b', 'c', 'd')
```

[50]:
```
t4 = t2 + t3
t4
```

[50]:
```
((1, 2, 3), ('a', 'b', 'c'), '1', '2', '3', '4', 'a', 'b', 'c', 'd')
```

4.1.2 Rozpakowywanie

[51]:
```
t1 = 1, 2, 3, 'a'
a, b, c, d = t1
a
```

[51]:
```
1
```

[52]:
```
a, b = b, a
a
```

[52]:
```
2
```

4.1.3 Przykładowe metody, funkcje i operatory

[53]:
```
t1 = 1, 2, 2, 1, 3, 4
t1.index(3)
```

[53]:
```
4
```

[54]:
```
t1.count(2)
```

[54]:
```
2
```

[55]:
```
len(t1)
```

[55]:
```
6
```

[56]:
```
max(t1)
```

[56]:
```
4
```

[57]:
```
sum(t1)
```

[57]:
```
13
```

[58]:
```
t1 = 1, 2
t2 = t1 * 4
t2
```

[58]:
```
(1, 2, 1, 2, 1, 2, 1, 2)
```

4.2 Listy

- Zmienna długość
- Modyfikowalna zawartość

4.2.1 Tworzenie

[59]:
```
l1 = [1, 2, 3, 'a']
l1
```

[59]:
```
[1, 2, 3, 'a']
```

[60]:
```
l2 = list(t2)
l2
```

[60]:
```
[1, 2, 1, 2, 1, 2, 1, 2]
```

4.2.2 Modyfikacja

[61]:
```
l2[3] = 3
l2
```

[61]:
```
[1, 2, 1, 3, 1, 2, 1, 2]
```

[62]:
```
l2.append(7)
l2
```

[62]:
```
[1, 2, 1, 3, 1, 2, 1, 2, 7]
```

[63]:
```
l2.insert(2,44)
l2
```

[63]:
```
[1, 2, 44, 1, 3, 1, 2, 1, 2, 7]
```

[64]:
```
l2.pop(2)
```

[64]:
```
44
```

[65]:
```
l2
```

[65]:
```
[1, 2, 1, 3, 1, 2, 1, 2, 7]
```

4.2.3 Przykładowe metody, funkcje i operatory

[66]:
```
44 in l2
```

[66]: False

[67]:
```
l2 + l1
```

[67]: [1, 2, 1, 3, 1, 2, 1, 2, 7, 1, 2, 3, 'a']

[68]:
```
l2.sort()
l2
```

[68]: [1, 1, 1, 1, 2, 2, 2, 3, 7]

[69]:
```
l2.sort(key = lambda v : -v)
l2
```

[69]: [7, 3, 2, 2, 2, 1, 1, 1, 1]

4.2.4 Wycinki

[70]:
```
l2[2:6]
```

[70]: [2, 2, 2, 1]

[71]:
```
l2[2:4] = [4, 5, 6]
l2
```

[71]: [7, 3, 4, 5, 6, 2, 1, 1, 1, 1]

[72]:
```
l2[:2]
```

[72]: [7, 3]

[73]:
```
l2[-2:]
```

[73]: [1, 1]

[74]:
```
# krok - ujemny iteruje od końca
l2[::3]
```

[74]: [7, 5, 1, 1]

4.3 Słownik

- Obiekt klasy `dict` jest odpowiednikiem mapy (tablicy asocjacyjnej).
- Dla poszczególnych kluczy przechowywane są wartości

- Obiekt `mutable` (modyfikowalny na poziomie wartości)
- Poprawne klucze to wartości hashowalne (niezmienne z niezmiennymi wartościami)

4.3.1 Tworzenie

[75]:
```
d1 = {'jan': 1000, 'marek': 2000, 'anna': 3000}
d1
```

[75]:
```
{'jan': 1000, 'marek': 2000, 'anna': 3000}
```

4.3.2 Modyfikacja

[76]:
```
d1['krzysztof'] = 4000
d1
```

[76]:
```
{'jan': 1000, 'marek': 2000, 'anna': 3000, 'krzysztof': 4000}
```

[77]:
```
del d1['anna']
d1
```

[77]:
```
{'jan': 1000, 'marek': 2000, 'krzysztof': 4000}
```

4.3.3 Metody

[78]:
```
d1.pop('jan')
```

[78]:
```
1000
```

[79]:
```
d1
```

[79]:
```
{'marek': 2000, 'krzysztof': 4000}
```

[80]:
```
list(d1.values())
```

[80]:
```
[2000, 4000]
```

[81]:
```
values = list(d1.keys())
values
```

[81]:
```
['marek', 'krzysztof']
```

[82]:
```
# wartość domyślna
d1.get('anna',3000)
```

[82]:
```
3000
```

[83]:
```
howmany = {}
for value in values:
    for letter in value:
        howmany[letter] = howmany.get(letter,0) + 1
howmany
```

[83]:
```
{'m': 1,
 'a': 1,
 'r': 2,
 'e': 1,
 'k': 2,
 'z': 2,
 'y': 1,
 's': 1,
 't': 1,
 'o': 1,
 'f': 1}
```

[84]:
```
d2 = {}

for l in howmany:
    d2.setdefault(howmany[l], []).append(l)
d2
```

[84]:
```
{1: ['m', 'a', 'e', 'y', 's', 't', 'o', 'f'], 2: ['r', 'k', 'z']}
```

4.4 Zbiór

- Zbiór nieuporządkowanych unikalnych elementów.
- Obiekt modyfikowalny

4.4.1 Tworzenie

[85]:
```
s1 = set([1, 2, 3, 1, 2, 3, 4, 6])
s1
```

[85]:
```
{1, 2, 3, 4, 6}
```

[86]:
```
s2 = {1, 2, 3, 4, 5}
s2
```

[86]:
```
{1, 2, 3, 4, 5}
```

[87]:
```
s1 | s2
```

[87]:
```
{1, 2, 3, 4, 5, 6}
```

[88]:
```
s1 & s2
```

[88]:
```
{1, 2, 3, 4}
```

4.4.2 Metody

[89]:
```
s1.add(7)
s1
```

[89]:
```
{1, 2, 3, 4, 6, 7}
```

[90]:
```
# jeśli jest wszystko z s1 jest w s2
s1.issubset(s2)
```

[90]:
```
False
```

[91]:
```
{1, 2, 3, 4, 5}.isdisjoint({6, 7})
```

[91]:
```
True
```

[92]:
```
s1.update(s2)
s1
```

[92]:
```
{1, 2, 3, 4, 5, 6, 7}
```

[93]:
```
s1.remove(1)
```

[94]:
```
#usuwa dowolny
s1.pop()
```

[94]:
```
2
```

[95]:
```
s1
```

[95]:
```
{3, 4, 5, 6, 7}
```

4.5 Funkcje dla sekwencji

[96]:
```
d2 = {}
l1 = [1, 2, 1, 2, 3, 4]
for p, v in enumerate(l1):
    d2.setdefault(v, []).append(p)
d2
```

[96]:
```
{1: [0, 2], 2: [1, 3], 3: [4], 4: [5]}
```

[97]:
```
# enumerate
d2 = {}
l1 = [1, 2, 1, 2, 3, 4]
for p, v in enumerate(sorted(l1)):
    d2.setdefault(v, []).append(p)
d2
```

[97]:
```
{1: [0, 1], 2: [2, 3], 3: [4], 4: [5]}
```

[98]:
```
# zip
names = ['jan', 'marek', 'anna']
values = [1000, 2000, 3000]

z1 = zip(names, values)

tuples = []

for n, v in list(z1):
    tuples.append((n,v))
tuples
```

[98]:
```
[('jan', 1000), ('marek', 2000), ('anna', 3000)]
```

[99]:
```
# reversed
for p, v in enumerate(reversed(tuples)):
    tuples[p] = tuple(reversed(v))
tuples
```

[99]:
```
[(3000, 'anna'), (2000, 'marek'), ('anna', 3000)]
```

4.6 Składanie (list, słowników, zbiorów)

Pozwala na tworzenie nowych sekwencji

- z istniejących sekwencji
- spełniających określone warunki
- zmienionych zgodnie z określonym wyrażeniem

`[<wyrażenie> for zmienna in lista if <warunek>]`

[100]:
```
l1 = list(range(6)) + list(range(7))
l1
```

[100]:
```
[0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5, 6]
```

[101]:
```
l2 = [x + 10 for x in l1 if x % 2 == 0]
l2
```

[101]:
```
[10, 12, 14, 10, 12, 14, 16]
```

[102]:
```
z1 = {x % 5 for x in l2 if x > 10}
z1
```

[102]:
```
{1, 2, 4}
```

[103]:
```
d2 = { k*4 : v+20 for k, v in enumerate(z1) if v % 2 == 0 }
d2
```

[103]:
```
{4: 22, 8: 24}
```

4.6.1 Zagnieżdżone składanie list

[104]:
```
l1 = [ ['jan', 'marek', 'anna'], [2000, 2000, 3000], ['jan', 'marek', 'anna',␣
 ↪'ole'] ]
l2 = [x for l in l1 if len(l) <= 3 for x in l if len(str(x)) > 3]
l2
```

[104]:
```
['marek', 'anna', 2000, 2000, 3000]
```

[105]:
```
l1 = [ ['jan', 'marek', 'anna'], [2000, 2000, 3000], ['jan', 'marek', 'anna',␣
 ↪'ole'] ]
l2 = [ {x for x in l if len(str(x)) > 3 } for l in l1 if len(l) <= 3]
l2
```

[105]:
```
[{'anna', 'marek'}, {2000, 3000}]
```

5 Funkcje

- definiuje się za pomocą słowa kluczowego `def`
- zawierają instrukcje `return`, które zwracają wynik działania funkcji. Brak oznacza wynik równy `None`

5.1 Przestrzenie nazw

[106]:
```
# zmienna zdefniowana w globalnej sprzestrzeń nazw
a = [8,9]

def func():
    # zmienna zdefniowana w lokalnej przestrzeni nazw
    a = []
    for i in range(5):
        a.append(i)
    print(a)
```

```
func()
a
```

```
[0, 1, 2, 3, 4]
```

[106]:
```
[8, 9]
```

5.2 Funkcje jako obiekty

Funkcje w Pythonie są obiektami. Pozwala to między innymi przekazywać je jako parametry, lub przypisywać do zmiennych

[107]:
```
def removeb(x):
    return str.replace(x, "b", "")

def apply_ops(l, ops):
    for op in ops:
        for i, v in enumerate(l):
            l[i] = op(v)
```

[108]:
```
my_ops = [removeb, str.upper, str.strip]

l1 = ["anna  ", "  bolek", " bernard " ]

apply_ops(l1, my_ops)

l1
```

[108]:
```
['ANNA', 'OLEK', 'ERNARD']
```

5.3 Wyrażenia lambda

W Pythonie, podobnie jak w wielu nowoczesnych językach programowania istnieje możliwość definiowania funkcji anonimowych.

[109]:
```
removeb = lambda x: str.replace(x, "b", "")

my_ops = [removeb, str.upper, str.strip]

l1 = ["anna  ", "  bolek", " bernard " ]

apply_ops(l1, my_ops)

l1
```

[109]:
```
['ANNA', 'OLEK', 'ERNARD']
```

[110]:
```
my_ops = [lambda x: str.replace(x, "b", ""), str.upper, str.strip]

l1 = ["anna  ", "  bolek", " bernard " ]

apply_ops(l1, my_ops)

l1
```

[110]:
```
['ANNA', 'OLEK', 'ERNARD']
```

5.4 Generatory

Jeśli w funkcji zamiast `return` wykorzystamy `yield` wówczas możemy stworzyć generator. Generatory tworzą iteratory i w odróżnieniu od zwykłych funkcji, które dają cały wynik w jednym momencie, generują kolejne wyniki *na żądanie*.

[111]:
```
def is_prime(x):
    if x < 2:
        return False
    else:
        for n in range(2,x):
            if x % n == 0:
                return False
        return True

def fun_gen():
    for i in range(1000000):
        if is_prime(i):
            yield i
```

[112]:
```
# Interpreter po wywołaniu generatora nie wykonuje żadnego kodu
gen = fun_gen()
gen
```

[112]:
```
<generator object fun_gen at 0x7f686e7ce2d0>
```

[113]:
```
# kod jest wykonywany (i kontynuowany) tylko w przypadku żądania elementów
# uruchom poniższe 3 x
i = 0
for p in gen:
    print(p)
    i += 1
    if i > 10:
        break
```

```
2
3
5
```

```
7
11
13
17
19
23
29
31
```

6 Klasy

Poniższy kod to definicja klasy `Complex`. Wyjaśnienia kolejnych fragmentów kodu:

`class Complex:` - Rozpoczyna definicję klasy o nazwie "Complex".

`def __init__(self, realpart, imagpart=0.0):` - To jest konstruktor klasy `Complex`. Przyjmuje dwa argumenty:

- `realpart` (część rzeczywista) i
- `imagpart` (część urojona).

Argument `imagpart` ma domyślną wartość 0.0, co oznacza, że możesz tworzyć obiekty `Complex` podając tylko jedną wartość jako część rzeczywistą, a część urojoną zostanie ustawiona na 0.0, jeśli nie jest podana.

`self.r = realpart` - Ustawia atrybut `r` obiektu `Complex` na podaną wartość `realpart`.

`self.i = imagpart` - Ustawia atrybut `i` obiektu `Complex` na podaną wartość `imagpart`.

`def __add__(self, other):` - To jest specjalna metoda w Pythonie, która umożliwia operatorowi `+` wykonywanie operacji dodawania na obiektach tej klasy. Metoda ta przyjmuje dwa argumenty: `self` (aktualny obiekt) i `other` (inny obiekt, który chcemy dodać). Jeśli `other` jest liczbą (`float` lub `int`), to ta metoda dodaje tę liczbę do części rzeczywistej obiektu i zwraca nowy obiekt `Complex`. Jeśli `other` jest obiektem klasy `Complex`, to metoda dodaje odpowiednie części rzeczywiste i urojone obu obiektów i zwraca nowy obiekt `Complex`.

`def __gt__(self, other):` - To jest specjalna metoda, która umożliwia operatorowi `>` wykonywanie porównań na obiektach tej klasy. Jednak ta metoda jest zdefiniowana jako metoda prywatna `_illegal`, która wyświetla komunikat o nielegalnej operacji.

`def dump(self):` - Metoda `dump` wypisuje atrybuty obiektu `Complex` na ekranie.

`def __str__(self):` - To jest specjalna metoda, która zwraca reprezentację obiektu jako ciąg znaków, która jest używana, gdy obiekt jest konwertowany na napis za pomocą funkcji `str()`. Ta metoda zwraca łańcuch znaków, który reprezentuje obiekt `Complex` w postaci "(część rzeczywista, część urojona)".

`def __repr__(self):` - To jest specjalna metoda, która zwraca reprezentację obiektu w postaci łańcucha znaków, która jest używana, gdy obiekt jest konwertowany na napis za pomocą funkcji repr(). Ta metoda zwraca łańcuch znaków, który reprezentuje obiekt `Complex` w postaci "Complex(część rzeczywista, część urojona)".

`def _illegal(self, op):` - To jest metoda prywatna, która jest wywoływana, gdy próbujemy wykonać nielegalną operację porównania na obiektach `Complex`. Wyświetla komunikat o nielegalnej operacji.

[114]:
```
class Complex:
    """Complex class with some methods."""
    def __init__(self, realpart, imagpart=0.0):
        self.r = realpart
        self.i = imagpart

    def __add__(self, other):
        if isinstance(other, (float,int)):
            return Complex(self.r + other, self.i)
        return Complex(self.r + other.r,
                       self.i + other.i)

    def __gt__(self, other):  self._illegal('>')

    def dump(self):
        print(self.__dict__)

    def __str__(self):
        return '(%g, %g)' % (self.r, self.i)

    def __repr__(self):
        return 'Complex' + str(self)

    def _illegal(self, op):
        print('illegal operation "%s" for complex numbers' % op)
```

[115]:
```
c1 = Complex(2,-1)
print(c1)
```

```
(2, -1)
```

[116]:
```
c2 = Complex(1)
print(c2)
```

```
(1, 0)
```

[117]:
```
c1 + c2
```

[117]:
```
Complex(3, -1)
```

[118]:
```
c1 + 4
```

[118]:
```
Complex(6, -1)
```

[119]:
```
c2 > 3
```

```
illegal operation ">" for complex numbers
```

[120]:
```
help(c1)
```

```
Help on Complex in module __main__ object:

class Complex(builtins.object)
 |  Complex(realpart, imagpart=0.0)
 |
 |  Complex class with some methods.
 |
 |  Methods defined here:
 |
 |  __add__(self, other)
 |
 |  __gt__(self, other)
 |      Return self>value.
 |
 |  __init__(self, realpart, imagpart=0.0)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __repr__(self)
 |      Return repr(self).
 |
 |  __str__(self)
 |      Return str(self).
 |
 |  dump(self)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

[121]:
```
c1.__doc__
```

[121]:
```
'Complex class with some methods.'
```

[122]:
```
print(dir(c1))
```

```
['__add__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
```

```
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_illegal', 'dump',
'i', 'r']
```

6.1 Dziedziczenie

[123]:
```
class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)
```

[124]:
```
class Student(Person):
    def __init__(self, fname, lname, grade):
        super().__init__(fname, lname)
        self.grade = grade # zmienne oddzielne dla każdej instancji
```

[125]:
```
p1 = Person("Anna","Kowalska")
p2 = Person("Stefan","Kowalski")
p1.printname()
```

```
Anna Kowalska
```

[126]:
```
s = Student("Jan","Nowak",5)
s.printname()
```

```
Jan Nowak
```

[127]:
```
s.grade
```

[127]:
```
5
```

7 Najważniejsze biblioteki przetwarzania danych

Popularność Pythona oraz jego wszechstronność wynika w dużej mierze z bogactwa różnorodnych bibliotek.

Niektóre z nich mają kluczowe znaczenie podczas przetwarzania i wizualizacji danych.
Poniżej kilka przykładów.

7.1 NumPy

`NumPy` to biblioteka będąca podstawą numerycznych obliczeń w Pythonie wykorzystywana przez wiele pakietów ukierunkowanych na dokonywanie obliczeń.

Zaimplementowana w C, przez co jej wydajność jest bardzo wysoka.

Kluczowe elementy pakietu `NumPy`:

- `ndarray` - wydajna implementacja tablic wielowymiarowych

- funkcje matematyczne do wykonywania operacji na całych tablicach
- narzędzia do odczytu i zapisu danych tablicowych z/do plików
- obsługa algebry liniowej, transformacji Fouriera i generowania liczb losowych

7.1.1 Wydajność

[128]:
```
import numpy as np
howmany = 100 * 1000 * 1000
np_array = np.arange(howmany)
typical_list = list(range(howmany))
```

[129]:
```
%time np_array2 = np_array % 1000
```

```
CPU times: user 1.04 s, sys: 461 ms, total: 1.5 s
Wall time: 1.5 s
```

[130]:
```
%time typical_list2 = [x % 1000 for x in typical_list]
```

```
CPU times: user 5.85 s, sys: 7.22 s, total: 13.1 s
Wall time: 13.1 s
```

7.1.2 `ndarray`

[131]:
```
import numpy as np
randdata = np.random.randn(3, 4)
randdata
```

[131]:
```
array([[ 0.15132273, -0.84053036,  1.83288837, -1.25338177],
       [ 1.3792694 ,  0.72162195,  0.28140556,  0.4588333 ],
       [ 1.13223319,  0.31837858, -0.44136728, -0.07182222]])
```

[132]:
```
randdata * 2
```

[132]:
```
array([[ 0.30264546, -1.68106071,  3.66577673, -2.50676355],
       [ 2.75853881,  1.4432439 ,  0.56281112,  0.9176666 ],
       [ 2.26446639,  0.63675716, -0.88273456, -0.14364445]])
```

[133]:
```
randdata + randdata
```

[133]:
```
array([[ 0.30264546, -1.68106071,  3.66577673, -2.50676355],
       [ 2.75853881,  1.4432439 ,  0.56281112,  0.9176666 ],
       [ 2.26446639,  0.63675716, -0.88273456, -0.14364445]])
```

[134]:
```
randdata.shape
```

[134]:
```
(3, 4)
```

[135]:
```
np.identity(4)
```

[135]:
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

[136]:
```
randdata[1,1:3] = 10
randdata
```

[136]:
```
array([[ 0.15132273, -0.84053036,  1.83288837, -1.25338177],
       [ 1.3792694 , 10.        , 10.        ,  0.4588333 ],
       [ 1.13223319,  0.31837858, -0.44136728, -0.07182222]])
```

7.1.3 Funkcje

[137]:
```
a = np.random.randn(4)
b = np.random.randn(4)
print(a)
print(b)
```

```
[ 0.85693341  0.13778204  0.20981159 -0.28212736]
[ 0.32239723 -0.49092115 -0.47236798 -0.79576453]
```

[138]:
```
np.maximum(a,b)
```

[138]:
```
array([ 0.85693341,  0.13778204,  0.20981159, -0.28212736])
```

[139]:
```
bsqrt = np.sqrt(b)
bsqrt
```

```
/tmp/ipykernel_16486/1603099121.py:1: RuntimeWarning: invalid value encountered
in sqrt
  bsqrt = np.sqrt(b)
```

[139]:
```
array([0.56780034,        nan,        nan,        nan])
```

[140]:
```
bsqrtisf = np.isfinite(np.sqrt(b))
bsqrtisf
```

```
/tmp/ipykernel_16486/3913250624.py:1: RuntimeWarning: invalid value encountered
in sqrt
  bsqrtisf = np.isfinite(np.sqrt(b))
```

[140]:
```
array([ True, False, False, False])
```

[141]:
```
sqrt_or_org = np.where(bsqrtisf,bsqrt,b)
sqrt_or_org
```

[141]:
```
array([ 0.56780034, -0.49092115, -0.47236798, -0.79576453])
```

[142]:
```
print(randdata)
randdata.sum(axis=1)
```

```
[[ 0.15132273 -0.84053036  1.83288837 -1.25338177]
 [ 1.3792694  10.          10.          0.4588333 ]
 [ 1.13223319  0.31837858 -0.44136728 -0.07182222]]
```

[142]:
```
array([-0.10970103, 21.8381027 ,  0.93742227])
```

[143]:
```
randdata.cumsum(axis=1)
```

[143]:
```
array([[ 0.15132273, -0.68920762,  1.14368074, -0.10970103],
       [ 1.3792694 , 11.3792694 , 21.3792694 , 21.8381027 ],
       [ 1.13223319,  1.45061178,  1.0092445 ,  0.93742227]])
```

[144]:
```
randdata > 0
```

[144]:
```
array([[ True, False,  True, False],
       [ True,  True,  True,  True],
       [ True,  True, False, False]])
```

[145]:
```
(randdata > 0).sum(axis=1)
```

[145]:
```
array([2, 4, 2])
```

[146]:
```
randdata.sort(axis=1)
randdata
```

[146]:
```
array([[-1.25338177, -0.84053036,  0.15132273,  1.83288837],
       [ 0.4588333 ,  1.3792694 , 10.        , 10.        ],
       [-0.44136728, -0.07182222,  0.31837858,  1.13223319]])
```

7.1.4 Operacje na plikach

[147]:
```
np.save('randdata',randdata)
```

[148]:
```
randdata = 1
randdata = np.load('randdata.npy')
randdata
```

[148]:
```
array([[-1.25338177, -0.84053036,  0.15132273,  1.83288837],
       [ 0.4588333 ,  1.3792694 , 10.        , 10.        ],
       [-0.44136728, -0.07182222,  0.31837858,  1.13223319]])
```

7.2 Pandas

Biblioteka posiadająca narzędzia oraz struktury do wszechstronnego przetwarzania danych. Wykorzystywana często w połączeniu z innymi narzędziami do przetwarzania danych czy wizualizacji: `NumPy`, `SciPy`, `sikit-learn`, `matplot`.

Dwa najważniejsze typy obiektów to:

- `Series` - jednowymiarowa indeksowana tablica
- `DataFrame` - dwuwymiarowa tabela

O ile `NumPy` koncentruje się na przetwarzaniu wartości numerycznych, o tyle **`pandas`** pozwala na korzystanie z danych heterogenicznych o bogatym zestawie typów wartości

7.2.1 Series

Przypomina słownik

Dokumentacja Series - (https://pandas.pydata.org/docs/reference/series.html)

[149]:
```
d1 = {'jan': 1000, 'marek': 2000, 'anna': 3000}
d1
```

[149]:
```
{'jan': 1000, 'marek': 2000, 'anna': 3000}
```

[150]:
```
import pandas as pd
from pandas import Series, DataFrame
se1 = pd.Series([1000, 2000, 3000])
se1
```

[150]:
```
0    1000
1    2000
2    3000
dtype: int64
```

[151]:
```
se1.values
```

[151]:
```
array([1000, 2000, 3000])
```

[152]:
```
se1.index
```

[152]:
```
RangeIndex(start=0, stop=3, step=1)
```

[153]:
```
se1.index = ['jan','marek','anna']
print(se1.index)
se1
```

```
Index(['jan', 'marek', 'anna'], dtype='object')
```

[153]:
```
jan      1000
marek    2000
anna     3000
dtype: int64
```

[154]:
```
se1.name = 'salary'
se1.index.name = 'name'
se1
```

[154]:
```
name
jan      1000
marek    2000
anna     3000
Name: salary, dtype: int64
```

[155]:
```
d1['krzysztof'] = 4000
se2 = pd.Series(d1)
se2
```

[155]:
```
jan          1000
marek        2000
anna         3000
krzysztof    4000
dtype: int64
```

[156]:
```
se1 + se2
```

[156]:
```
anna         6000.0
jan          2000.0
krzysztof       NaN
marek        4000.0
dtype: float64
```

7.2.2 DataFrame

Tabela danych. Uporządkowany zbiór kolumn, każda z wartością określonego typu.

Konstruktory `DataFrame` obsługują wiele typów danych, od macierzy `ndarray` poprzez słowniki, listy i ich kombinacje.

Dokumentacja DataFrame - (https://pandas.pydata.org/docs/reference/frame.html)

[157]:
```
wojewodztwa = {
    'teryt': [2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32],
    'nazwa':␣
 ↪['dolnośląskie','kujawsko-pomorskie','lubelskie','lubuskie','łódzkie','małopolskie','mazowi
```

```
    'stolica': ['Wrocław','Bydgoszcz','Lublin','Gorzów␣
↪Wielkopolski','Łódź','Kraków','Warszawa','Opole','Rzeszów','Białystok','Gdańsk','Katowice',
    'powierzchnia':␣
↪[19947,17972,25122,13988,18219,15183,35558,9412,17846,20187,18321,12333,11711,24173,29826,2
    'ludnosc':␣
↪[2898525,2069273,2103342,1010177,2448713,3413931,5428031,980771,2125901,1176576,2346717,450
    'gestosc': [145.45,115.62,84.29,72.53,135.37,223.98,151.96,104.82,119.3,58.
↪53,127.44,367.59,106.02,59.11,117.14,74.31],
    'urbanizacja': [68.61,59.09,46.46,64.95,62.55,48.19,64.4,53.3,41.09,60.
↪79,63.66,76.73,44.86,58.99,54.27,68.5],
    'bezrobocie': [5.4,9.2,8.4,6.2,6.3,4.9,5.1,6.6,9.1,8.1,5.2,4.5,8.7,10.9,3.
↪3,7.8],
    'pkb':␣
↪[57228,41875,35712,42755,48126,47272,83123,41080,36088,37077,50001,53654,36970,36306,56496,
    'symbol_rej':␣
↪['D','C','L','F','E','K','W','O','R','B','G','S','T','N','P','Z']}
woj = pd.DataFrame(wojewodztwa)
woj
```

[157]:

	teryt	nazwa	stolica	powierzchnia	ludnosc	\
0	2	dolnośląskie	Wrocław	19947	2898525	
1	4	kujawsko-pomorskie	Bydgoszcz	17972	2069273	
2	6	lubelskie	Lublin	25122	2103342	
3	8	lubuskie	Gorzów Wielkopolski	13988	1010177	
4	10	łódzkie	Łódź	18219	2448713	
5	12	małopolskie	Kraków	15183	3413931	
6	14	mazowieckie	Warszawa	35558	5428031	
7	16	opolskie	Opole	9412	980771	
8	18	podkarpackie	Rzeszów	17846	2125901	
9	20	podlaskie	Białystok	20187	1176576	
10	22	pomorskie	Gdańsk	18321	2346717	
11	24	śląskie	Katowice	12333	4508078	
12	26	świętokrzyskie	Kielce	11711	1230044	
13	28	warmińsko-mazurskie	Olsztyn	24173	1420514	
14	30	wielkopolskie	Poznań	29826	3500361	
15	32	zachodniopomorskie	Szczecin	22897	1693219	

	gestosc	urbanizacja	bezrobocie	pkb	symbol_rej
0	145.45	68.61	5.4	57228	D
1	115.62	59.09	9.2	41875	C
2	84.29	46.46	8.4	35712	L
3	72.53	64.95	6.2	42755	F
4	135.37	62.55	6.3	48126	E
5	223.98	48.19	4.9	47272	K
6	151.96	64.40	5.1	83123	W
7	104.82	53.30	6.6	41080	O

```
8    119.30        41.09         9.1  36088          R
9     58.53        60.79         8.1  37077          B
10   127.44        63.66         5.2  50001          G
11   367.59        76.73         4.5  53654          S
12   106.02        44.86         8.7  36970          T
13    59.11        58.99        10.9  36306          N
14   117.14        54.27         3.3  56496          P
15    74.31        68.50         7.8  43150          Z
```

[158]:
```
woj.head()
```

[158]:
```
   teryt              nazwa              stolica  powierzchnia  ludnosc  \
0      2       dolnośląskie              Wrocław         19947  2898525
1      4  kujawsko-pomorskie            Bydgoszcz         17972  2069273
2      6           lubelskie               Lublin         25122  2103342
3      8            lubuskie  Gorzów Wielkopolski         13988  1010177
4     10             łódzkie                Łódź         18219  2448713

   gestosc  urbanizacja  bezrobocie    pkb symbol_rej
0   145.45        68.61         5.4  57228          D
1   115.62        59.09         9.2  41875          C
2    84.29        46.46         8.4  35712          L
3    72.53        64.95         6.2  42755          F
4   135.37        62.55         6.3  48126          E
```

[159]:
```
pd.DataFrame(woj, columns=['nazwa', 'powierzchnia', 'ludnosc']).head()
```

[159]:
```
               nazwa  powierzchnia  ludnosc
0       dolnośląskie         19947  2898525
1  kujawsko-pomorskie         17972  2069273
2           lubelskie         25122  2103342
3            lubuskie         13988  1010177
4             łódzkie         18219  2448713
```

[160]:
```
# dostajemy obiekt Series
woj['nazwa'].head()
```

[160]:
```
0         dolnośląskie
1    kujawsko-pomorskie
2             lubelskie
3              lubuskie
4               łódzkie
Name: nazwa, dtype: object
```

[161]:
```
woj.index = woj['nazwa'].values
```

[162]:
```
woj.head()
```

[162]:
```
                     teryt              nazwa              stolica  \
dolnośląskie             2       dolnośląskie              Wrocław
kujawsko-pomorskie       4  kujawsko-pomorskie            Bydgoszcz
lubelskie                6           lubelskie               Lublin
lubuskie                 8            lubuskie  Gorzów Wielkopolski
łódzkie                 10             łódzkie                Łódź

                     powierzchnia  ludnosc  gestosc  urbanizacja  bezrobocie  \
dolnośląskie                19947  2898525   145.45        68.61         5.4
kujawsko-pomorskie          17972  2069273   115.62        59.09         9.2
lubelskie                   25122  2103342    84.29        46.46         8.4
lubuskie                    13988  1010177    72.53        64.95         6.2
łódzkie                     18219  2448713   135.37        62.55         6.3

                       pkb symbol_rej
dolnośląskie         57228          D
kujawsko-pomorskie   41875          C
lubelskie            35712          L
lubuskie             42755          F
łódzkie              48126          E
```

[163]:
```
woj['stolica'].head()
```

[163]:
```
dolnośląskie                      Wrocław
kujawsko-pomorskie              Bydgoszcz
lubelskie                         Lublin
lubuskie              Gorzów Wielkopolski
łódzkie                             Łódź
Name: stolica, dtype: object
```

[164]:
```
woj.columns
```

[164]:
```
Index(['teryt', 'nazwa', 'stolica', 'powierzchnia', 'ludnosc', 'gestosc',
       'urbanizacja', 'bezrobocie', 'pkb', 'symbol_rej'],
      dtype='object')
```

[165]:
```
woj.loc['wielkopolskie']
```

[165]:
```
teryt                    30
nazwa           wielkopolskie
stolica               Poznań
powierzchnia           29826
ludnosc              3500361
gestosc               117.14
urbanizacja            54.27
bezrobocie               3.3
pkb                    56496
```

```
symbol_rej                  P
Name: wielkopolskie, dtype: object
```

[166]:
```
woj[woj.bezrobocie < 5]
```

[166]:
```
                teryt          nazwa   stolica  powierzchnia  ludnosc  gestosc  \
małopolskie        12    małopolskie    Kraków         15183  3413931   223.98
śląskie            24        śląskie  Katowice         12333  4508078   367.59
wielkopolskie      30  wielkopolskie    Poznań         29826  3500361   117.14

                urbanizacja  bezrobocie    pkb symbol_rej
małopolskie           48.19         4.9  47272          K
śląskie               76.73         4.5  53654          S
wielkopolskie         54.27         3.3  56496          P
```

[167]:
```
woj.query('bezrobocie < 5')
```

[167]:
```
                teryt          nazwa   stolica  powierzchnia  ludnosc  gestosc  \
małopolskie        12    małopolskie    Kraków         15183  3413931   223.98
śląskie            24        śląskie  Katowice         12333  4508078   367.59
wielkopolskie      30  wielkopolskie    Poznań         29826  3500361   117.14

                urbanizacja  bezrobocie    pkb symbol_rej
małopolskie           48.19         4.9  47272          K
śląskie               76.73         4.5  53654          S
wielkopolskie         54.27         3.3  56496          P
```

[168]:
```
woj[woj.bezrobocie < 5].iloc[:2,2:5]
```

[168]:
```
              stolica  powierzchnia  ludnosc
małopolskie    Kraków         15183  3413931
śląskie      Katowice         12333  4508078
```

[169]:
```
woj[woj.bezrobocie < 5].iloc[:2,[1,2,7]]
```

[169]:
```
                   nazwa   stolica  bezrobocie
małopolskie  małopolskie    Kraków         4.9
śląskie          śląskie  Katowice         4.5
```

[170]:
```
woj.shape
```

[170]:
```
(16, 10)
```

7.2.3 Funkcje apply i map

[171]:
```
x = woj.iloc[:,3:8].apply(lambda x: x.max() - x.min())
print(type(x))
x
```

```
<class 'pandas.core.series.Series'>
```

[171]:
```
powierzchnia      26146.00
ludnosc         4447260.00
gestosc             309.06
urbanizacja          35.64
bezrobocie            7.60
dtype: float64
```

[172]:
```
# funkcja nie musi zwracać jednej wartości
def f(x):
    return pd.Series([x.max(),x.min(),x.mean()], index=['min', 'max','avg'])

x = woj.iloc[:,3:8].apply(f)
print(type(x))
x
```

```
<class 'pandas.core.frame.DataFrame'>
```

[172]:

	powierzchnia	ludnosc	gestosc	urbanizacja	bezrobocie
min	35558.0000	5.428031e+06	367.59000	76.7300	10.90000
max	9412.0000	9.807710e+05	58.53000	41.0900	3.30000
avg	19543.4375	2.397136e+06	128.96625	58.5275	6.85625

[173]:
```
x = woj['pkb'].map(lambda x: x * 2)
x.name = 'bigpkb'
x
```

[173]:
```
dolnośląskie           114456
kujawsko-pomorskie      83750
lubelskie               71424
lubuskie                85510
łódzkie                 96252
małopolskie             94544
mazowieckie            166246
opolskie                82160
podkarpackie            72176
podlaskie               74154
pomorskie              100002
śląskie                107308
świętokrzyskie          73940
warmińsko-mazurskie     72612
```

```
wielkopolskie          112992
zachodniopomorskie      86300
Name: bigpkb, dtype: int64
```

7.2.4 Sortowanie i ranking

[174]:
```
woj.sort_values(by='bezrobocie').head()
```

[174]:
```
                teryt          nazwa   stolica  powierzchnia  ludnosc  gestosc  \
wielkopolskie      30  wielkopolskie    Poznań         29826  3500361   117.14
śląskie            24        śląskie  Katowice         12333  4508078   367.59
małopolskie        12    małopolskie    Kraków         15183  3413931   223.98
mazowieckie        14    mazowieckie  Warszawa         35558  5428031   151.96
pomorskie          22      pomorskie    Gdańsk         18321  2346717   127.44

                urbanizacja  bezrobocie    pkb symbol_rej
wielkopolskie         54.27         3.3  56496          P
śląskie               76.73         4.5  53654          S
małopolskie           48.19         4.9  47272          K
mazowieckie           64.40         5.1  83123          W
pomorskie             63.66         5.2  50001          G
```

[175]:
```
woj['bezrobocie'].rank().sort_values()
```

[175]:
```
wielkopolskie           1.0
śląskie                 2.0
małopolskie             3.0
mazowieckie             4.0
pomorskie               5.0
dolnośląskie            6.0
lubuskie                7.0
łódzkie                 8.0
opolskie                9.0
zachodniopomorskie     10.0
podlaskie              11.0
lubelskie              12.0
świętokrzyskie         13.0
podkarpackie           14.0
kujawsko-pomorskie     15.0
warmińsko-mazurskie    16.0
Name: bezrobocie, dtype: float64
```

7.2.5 Statystyki

[176]:
```
woj.describe()
```

[176]:
```
            teryt  powierzchnia       ludnosc     gestosc  urbanizacja  \
count   16.000000     16.000000  1.600000e+01   16.000000    16.000000
mean    17.000000  19543.437500  2.397136e+06  128.966250    58.527500
std      9.521905   6836.338883  1.281880e+06   75.869689     9.815478
min      2.000000   9412.000000  9.807710e+05   58.530000    41.090000
25%      9.500000  14884.250000  1.372896e+06   81.795000    52.022500
50%     17.000000  18270.000000  2.114622e+06  116.380000    59.940000
75%     24.500000  23216.000000  3.027376e+06  137.890000    64.537500
max     32.000000  35558.000000  5.428031e+06  367.590000    76.730000

        bezrobocie           pkb
count    16.00000     16.000000
mean      6.85625  46682.062500
std       2.09570  12129.130392
min       3.30000  35712.000000
25%       5.17500  37050.250000
50%       6.45000  42952.500000
75%       8.47500  50914.250000
max      10.90000  83123.000000
```

7.2.6 Korelacje

[177]:
```
woj['bezrobocie'].corr(woj['gestosc'])
```

[177]:
```
-0.5510191870112093
```

[178]:
```
woj['urbanizacja'].corr(woj['pkb'])
```

[178]:
```
0.4545066039439765
```

[179]:
```
woj['powierzchnia'].corr(woj['bezrobocie'])
```

[179]:
```
-0.09954361441267383
```

[180]:
```
woj.iloc[:,3:8].corr()
```

[180]:
```
              powierzchnia   ludnosc   gestosc  urbanizacja  bezrobocie
powierzchnia      1.000000  0.481466 -0.246039     0.078817   -0.099544
ludnosc           0.481466  1.000000  0.697606     0.309029   -0.638406
gestosc          -0.246039  0.697606  1.000000     0.337647   -0.551019
urbanizacja       0.078817  0.309029  0.337647     1.000000   -0.368414
bezrobocie       -0.099544 -0.638406 -0.551019    -0.368414    1.000000
```

[181]:
```
# jeśli atrybut jest poprawną nazwą Pythona to możemy się do niego odwołać za
↪pomocą bardziej zwięzłej notacji
woj.iloc[:,3:8].corrwith(woj.pkb)
```

[181]:
```
powierzchnia    0.549019
ludnosc         0.851206
gestosc         0.394859
urbanizacja     0.454507
bezrobocie     -0.685165
dtype: float64
```

[182]:
```
woj.iloc[:,3:8].corr().applymap(lambda x: '%.2f' % x)
```

[182]:

	powierzchnia	ludnosc	gestosc	urbanizacja	bezrobocie
powierzchnia	1.00	0.48	-0.25	0.08	-0.10
ludnosc	0.48	1.00	0.70	0.31	-0.64
gestosc	-0.25	0.70	1.00	0.34	-0.55
urbanizacja	0.08	0.31	0.34	1.00	-0.37
bezrobocie	-0.10	-0.64	-0.55	-0.37	1.00

7.2.7 Grupowanie

[183]:
```
woj['region'] =␣
 ↪['południe','zachód','wschód','zachód','centrum','południe','centrum','południe',
                 ␣
 ↪'południe','wschód','północ','południe','południe','północ','zachód','północ']
woj
```

[183]:

	teryt	nazwa	stolica	\
dolnośląskie	2	dolnośląskie	Wrocław	
kujawsko-pomorskie	4	kujawsko-pomorskie	Bydgoszcz	
lubelskie	6	lubelskie	Lublin	
lubuskie	8	lubuskie	Gorzów Wielkopolski	
łódzkie	10	łódzkie	Łódź	
małopolskie	12	małopolskie	Kraków	
mazowieckie	14	mazowieckie	Warszawa	
opolskie	16	opolskie	Opole	
podkarpackie	18	podkarpackie	Rzeszów	
podlaskie	20	podlaskie	Białystok	
pomorskie	22	pomorskie	Gdańsk	
śląskie	24	śląskie	Katowice	
świętokrzyskie	26	świętokrzyskie	Kielce	
warmińsko-mazurskie	28	warmińsko-mazurskie	Olsztyn	
wielkopolskie	30	wielkopolskie	Poznań	
zachodniopomorskie	32	zachodniopomorskie	Szczecin	

	powierzchnia	ludnosc	gestosc	urbanizacja	bezrobocie	\
dolnośląskie	19947	2898525	145.45	68.61	5.4	
kujawsko-pomorskie	17972	2069273	115.62	59.09	9.2	
lubelskie	25122	2103342	84.29	46.46	8.4	
lubuskie	13988	1010177	72.53	64.95	6.2	

```
łódzkie                    18219  2448713   135.37        62.55         6.3
małopolskie                15183  3413931   223.98        48.19         4.9
mazowieckie                35558  5428031   151.96        64.40         5.1
opolskie                    9412   980771   104.82        53.30         6.6
podkarpackie               17846  2125901   119.30        41.09         9.1
podlaskie                  20187  1176576    58.53        60.79         8.1
pomorskie                  18321  2346717   127.44        63.66         5.2
śląskie                    12333  4508078   367.59        76.73         4.5
świętokrzyskie             11711  1230044   106.02        44.86         8.7
warmińsko-mazurskie        24173  1420514    59.11        58.99        10.9
wielkopolskie              29826  3500361   117.14        54.27         3.3
zachodniopomorskie         22897  1693219    74.31        68.50         7.8

                          pkb symbol_rej    region
dolnośląskie            57228          D  południe
kujawsko-pomorskie      41875          C    zachód
lubelskie               35712          L    wschód
lubuskie                42755          F    zachód
łódzkie                 48126          E   centrum
małopolskie             47272          K  południe
mazowieckie             83123          W   centrum
opolskie                41080          O  południe
podkarpackie            36088          R  południe
podlaskie               37077          B    wschód
pomorskie               50001          G    północ
śląskie                 53654          S  południe
świętokrzyskie          36970          T  południe
warmińsko-mazurskie     36306          N    północ
wielkopolskie           56496          P    zachód
zachodniopomorskie      43150          Z    północ
```

[184]:
```
# takie funkcje jak mean pomijają kolumny nienumeryczne
woj.drop(columns="teryt").groupby("region").mean().sort_values(by='bezrobocie')
```

[184]:
```
           powierzchnia       ludnosc     gestosc  urbanizacja  bezrobocie  \
region
centrum    26888.500000  3.938372e+06  143.665000    63.475000    5.700000
zachód     20595.333333  2.193270e+06  101.763333    59.436667    6.233333
południe   14405.333333  2.526208e+06  177.860000    55.463333    6.533333
północ     21797.000000  1.820150e+06   86.953333    63.716667    7.966667
wschód     22654.500000  1.639959e+06   71.410000    53.625000    8.250000

                    pkb
region
centrum    65624.500000
zachód     47042.000000
południe   45382.000000
```

```
północ    43152.333333
wschód    36394.500000
```

8 Wizualizacja

Wizualizacja jest kluczowym elementem analizy danych.

Python posiada biblioteki, które to umożliwiają. W szczególności zalicza się do nich:

- `matplotlib`,
- wspomniana `pandas` oraz
- `seaborn` (oparty na `matplotlib`)

[ ]:
```
%%sh
pip install yfinance
```

[190]:
```
# pip install yfinance
import yfinance as yf
import pandas as pd

start_date = '2022-01-15'
end_date = '2022-02-03'
tickers = ['AAPL', 'AMZN', 'MSFT', 'GOOG']

all_data = {}
for ticker in tickers:
    stock_data = yf.download(ticker, start=start_date, end=end_date)
    all_data[ticker] = stock_data

print(all_data['AAPL'].head())
```

```
[**********100%%**********]  1 of 1 completed
[**********100%%**********]  1 of 1 completed
[**********100%%**********]  1 of 1 completed
[**********100%%**********]  1 of 1 completed
                 Open        High         Low       Close   Adj Close  \
Date
2022-01-18  171.509995  172.539993  169.410004  169.800003  168.103012
2022-01-19  170.000000  171.080002  165.940002  166.229996  164.568680
2022-01-20  166.979996  169.679993  164.179993  164.509995  162.865845
2022-01-21  164.419998  166.330002  162.300003  162.410004  160.786865
2022-01-24  160.020004  162.300003  154.699997  161.619995  160.004776

               Volume
Date
2022-01-18   90956700
2022-01-19   94815000
2022-01-20   91420500
```

```
2022-01-21  122848900
2022-01-24  162294600
```

[196]:
```
# na wszelki wypadek, można zapisać te dane na przyszłość
allData = {}
from pathlib import Path
for ticker in all_data:
    filepath = Path('datafiles/yahoo/' + ticker.lower() + '.csv')
    print(filepath)
    filepath.parent.mkdir(parents=True, exist_ok=True)
    all_data[ticker].to_csv(filepath)
```

```
datafiles/yahoo/aapl.csv
datafiles/yahoo/amzn.csv
datafiles/yahoo/msft.csv
datafiles/yahoo/goog.csv
```

[197]:
```
# i jeśli takie wypadek wystąpi (gdyby yfinance nie działał poprawnie)␣
 ↪zaczytamy te dane z lokalnych plików
allData = {}
for ticker in ['AAPL', 'AMZN', 'MSFT', 'GOOG']:
    filepath = Path('datafiles/yahoo/' + ticker.lower() + '.csv')
    print(filepath)
    allData[ticker] = pd.read_csv(filepath)
allData['AAPL'].head()
```

```
datafiles/yahoo/aapl.csv
datafiles/yahoo/amzn.csv
datafiles/yahoo/msft.csv
datafiles/yahoo/goog.csv
```

[197]:

	Date	Open	High	Low	Close	Adj Close	Volume
0	2022-01-18	171.509995	172.539993	169.410004	169.800003	168.103012	90956700
1	2022-01-19	170.000000	171.080002	165.940002	166.229996	164.568680	94815000
2	2022-01-20	166.979996	169.679993	164.179993	164.509995	162.865845	91420500
3	2022-01-21	164.419998	166.330002	162.300003	162.410004	160.786865	122848900
4	2022-01-24	160.020004	162.300003	154.699997	161.619995	160.004776	162294600

[198]:
```
price = pd.DataFrame({ticker: data['Adj Close'] for ticker, data in allData.
 ↪items()})
```

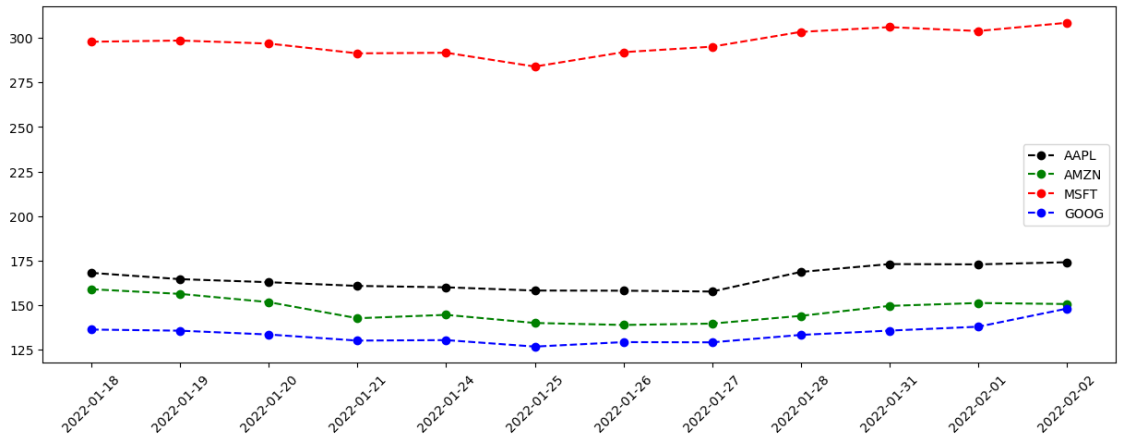
```
volume = pd.DataFrame({ticker: data['Volume'] for ticker, data in allData.
  ↪items()})
price.index = allData["MSFT"]['Date'].values
volume.index = allData["MSFT"]['Date'].values
price
```

[198]:

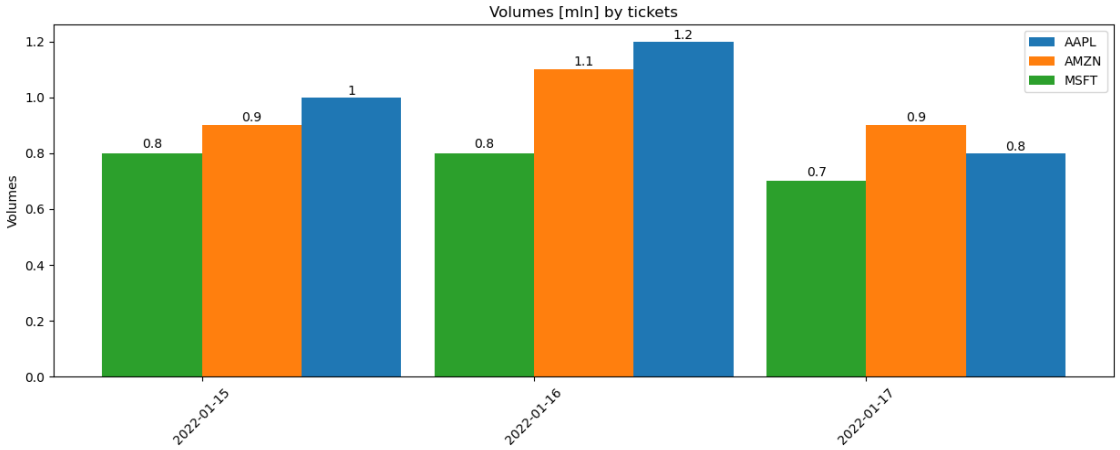
	AAPL	AMZN	MSFT	GOOG
2022-01-18	168.103012	158.917496	297.808350	136.290497
2022-01-19	164.568680	156.298996	298.477478	135.651993
2022-01-20	162.865845	151.667496	296.775208	133.506500
2022-01-21	160.786865	142.643005	291.294281	130.091995
2022-01-24	160.004776	144.544006	291.628815	130.371994
2022-01-25	158.183151	139.985992	283.874908	126.735497
2022-01-26	158.094040	138.872498	291.963379	129.240005
2022-01-27	157.628723	139.637497	295.043304	129.121002
2022-01-28	168.627716	143.977997	303.328644	133.289505
2022-01-31	173.033234	149.573502	306.005157	135.698502
2022-02-01	172.864944	151.193497	303.820679	137.878494
2022-02-02	174.082626	150.612503	308.445435	148.036499

8.1 matplotlib

[199]:
```
import matplotlib.pyplot as plt

plt.figure(figsize=(15,5))
plt.plot(price['AAPL'], color='k', linestyle='dashed', marker='o', label='AAPL')
plt.plot(price['AMZN'], color='g', linestyle='dashed', marker='o', label='AMZN')
plt.plot(price['MSFT'], color='r', linestyle='dashed', marker='o', label='MSFT')
plt.plot(price['GOOG'], color='b', linestyle='dashed', marker='o', label='GOOG')
plt.xticks(rotation=45)
plt.legend(loc='best')
plt.savefig('prices.svg')
plt.show()
```



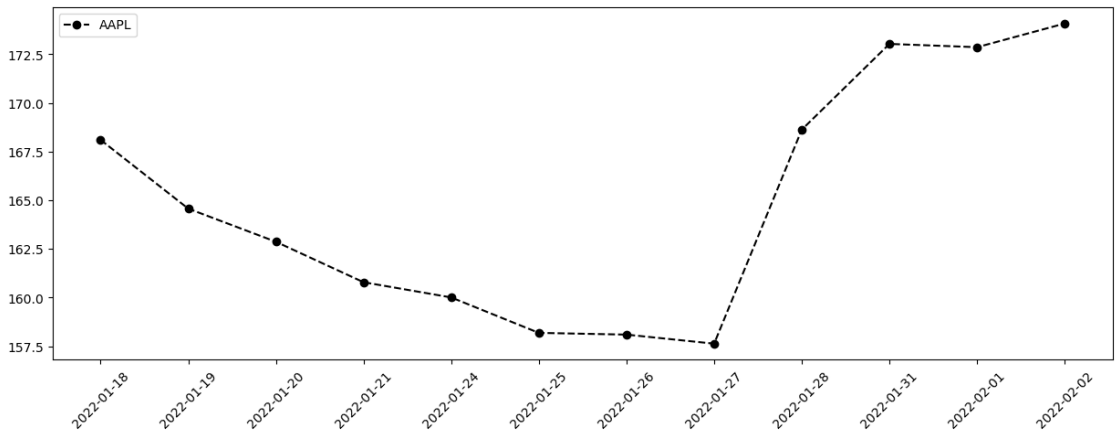
[201]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from datetime import datetime

# Przykładowe dane - zastąp to swoimi danymi
volume = pd.DataFrame({'AAPL': [1000000, 1200000, 800000],
                       'AMZN': [900000, 1100000, 950000],
                       'MSFT': [750000, 850000, 700000]},
                      index=['2022-01-15', '2022-01-16', '2022-01-17'])

lbls = [datetime.strptime(label, "%Y-%m-%d").strftime("%Y-%m-%d") for label in␣
 ↪volume.index]
noseries = len(volume.columns)
x = np.arange(len(lbls))  # the label locations
width = 0.9/noseries  # the width of the bars

fig = plt.figure(figsize=(15, 5))
ax = fig.add_subplot()

for i, l in enumerate(volume.columns):
    r = ax.bar(x - width * i + (width * noseries / 2), volume[l].map(lambda x:␣
 ↪round(x/1000000, 1)), width, label=l)
    ax.bar_label(r, padding=i)

ax.set_ylabel('Volumes')
ax.set_title('Volumes [mln] by tickets')
ax.set_xticks(x, lbls)
ax.tick_params(axis='x', rotation=45)

ax.legend()
plt.show()
```



Środowisko `matplotlib` jest domyślne skonfigurowane. Możemy jednak dokonywać korekt tego źrodowiska wykorzystując metodę `rc`.
Przykładowo, zamiast za każdym razem kofigurować wielkość rysunków w powyższych przykładach można było na początku skonfigurować wielkość rysunków za pomocą: `plt.rc('figure', figsize=(15, 5))`

[202]:
```
plt.rc('figure', figsize=(15, 5))

plt.plot(price['AAPL'], color='k', linestyle='dashed', marker='o', label='AAPL')
plt.xticks(rotation=45)
plt.legend(loc='best')
```

[202]:
```
<matplotlib.legend.Legend at 0x7f66404f5390>
```



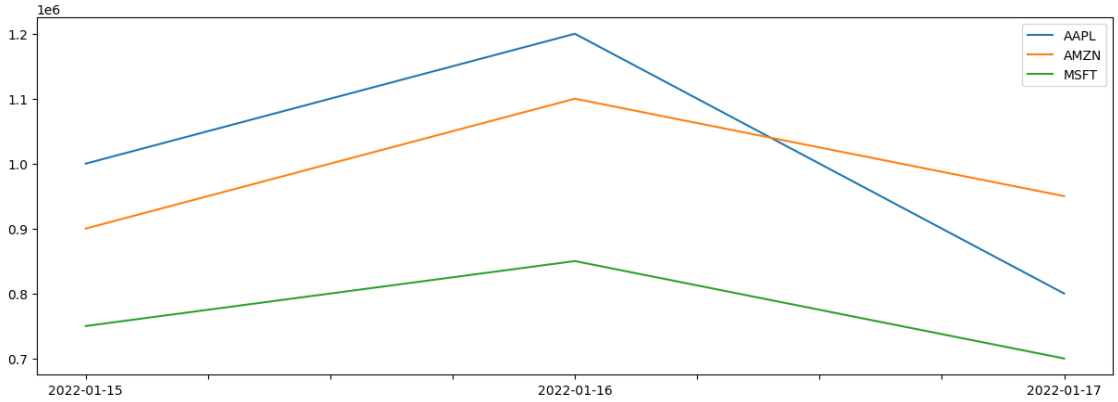
8.2 pandas

Biblioteka `matplotlib`, jak doskonale było to widać, to dość niskopoziomowe narzędzie. Rysunki są tam składane z wykresów, kolumn, legendy, osi, etykiet itp.

Na szczęście są biblioteki, które wewnętrznie korzystając z biblioteki `matplotlib`, proces tworzenia wykresów znacząco ułatwiają. Przykładem jest sam `pandas` czy `seaborn`

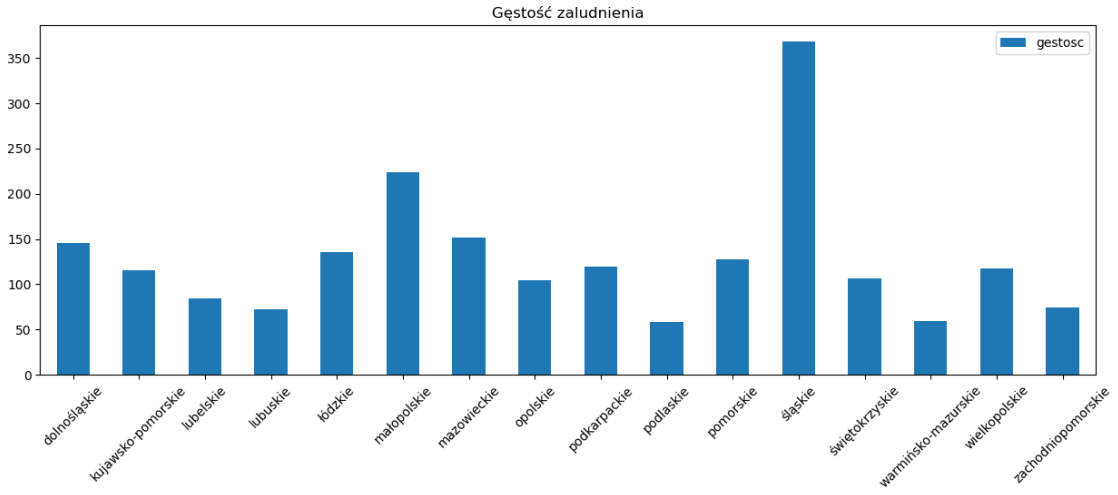
[203]:
```
import pandas as pd

volume.plot() #chyba nie da się prościej
```

[203]: <AxesSubplot:>



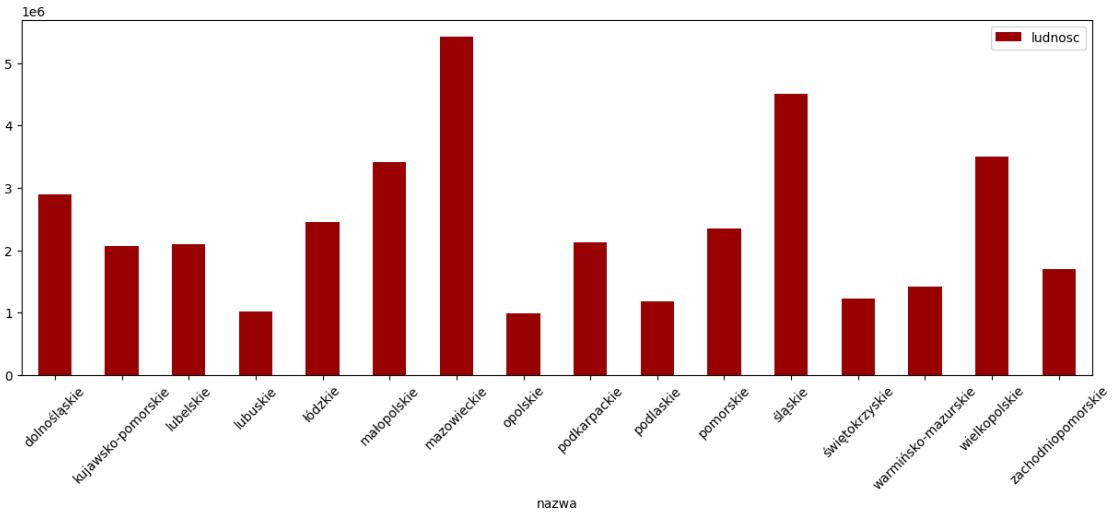
[204]:
```
df = pd.DataFrame(woj["gestosc"])
df.plot(title="Gęstość zaludnienia",
        kind="bar", rot=45,
        sort_columns=True)
```

[204]: <AxesSubplot:title={'center':'Gęstość zaludnienia'}>

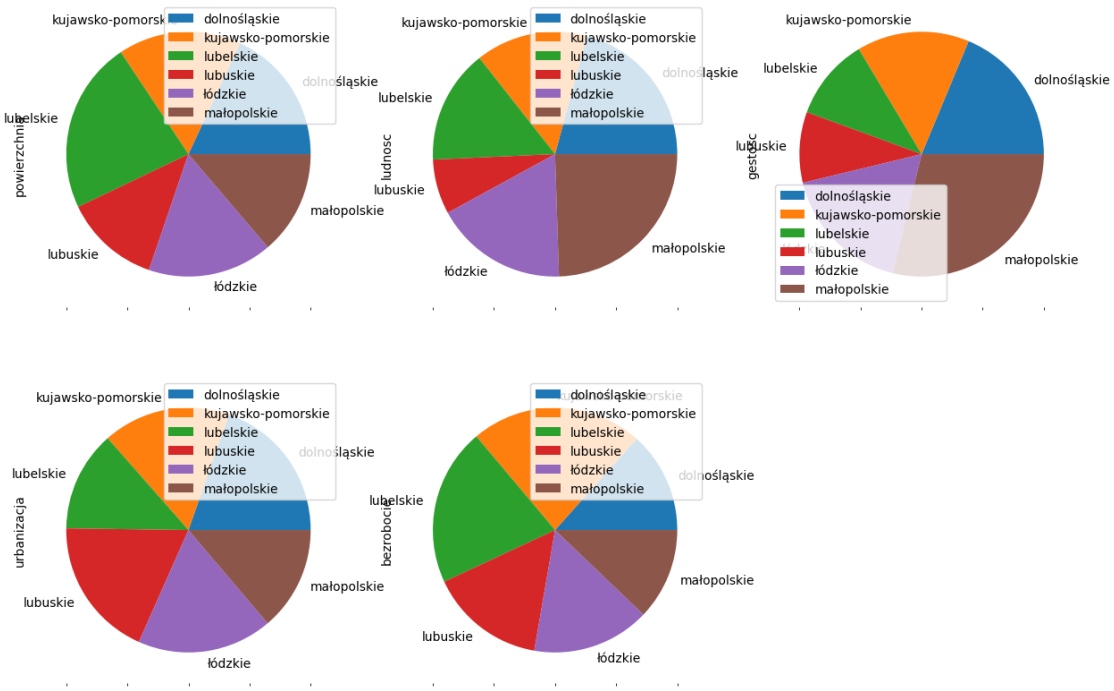


[205]:
```
woj.plot.bar(x="nazwa",y="ludnosc", color="#990000", rot=45)
```

[205]: <AxesSubplot:xlabel='nazwa'>



[206]:
```
plot = woj.iloc[:6,3:8].plot.pie(subplots=True, grid=True,
                                 figsize=(15, 10), layout=(2, 3), rot=30,
                                 sharex=True, sharey=False, legend=True)
```



8.3 `seaborn`

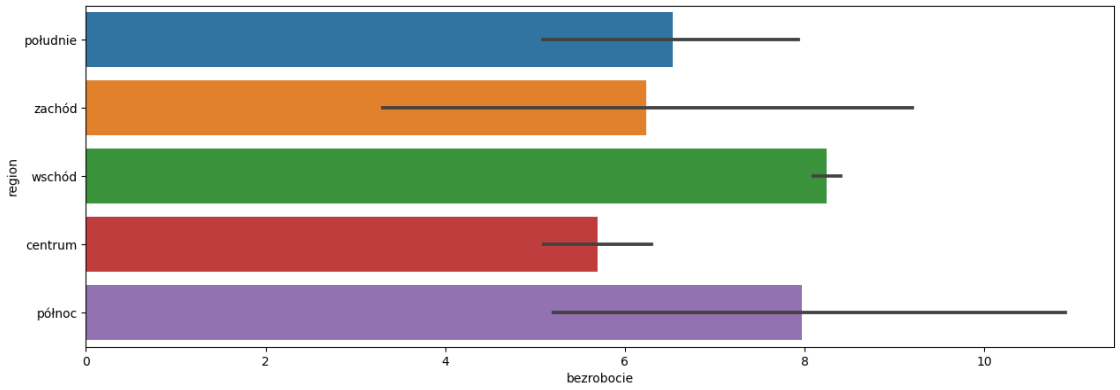
To kolejna biblioteka oparta o `matplotlib`
Jej główne funkcjonalności obejmują prezentację:

- zależności i powiązań pomiędzy danymi
- rozkładu, trendów, danych odstających (często wymagane na początkowym etapie analizy danych)
- danych posiadających wartości kategoryczne
- regresji liniowej
- analiza danych wielowymiarowych na serii wykresów
- z wykorzystaniem szeregu predefiniowanych stylów i zestawów kolorów

Dokumentacja pakietu (https://seaborn.pydata.org/index.html)

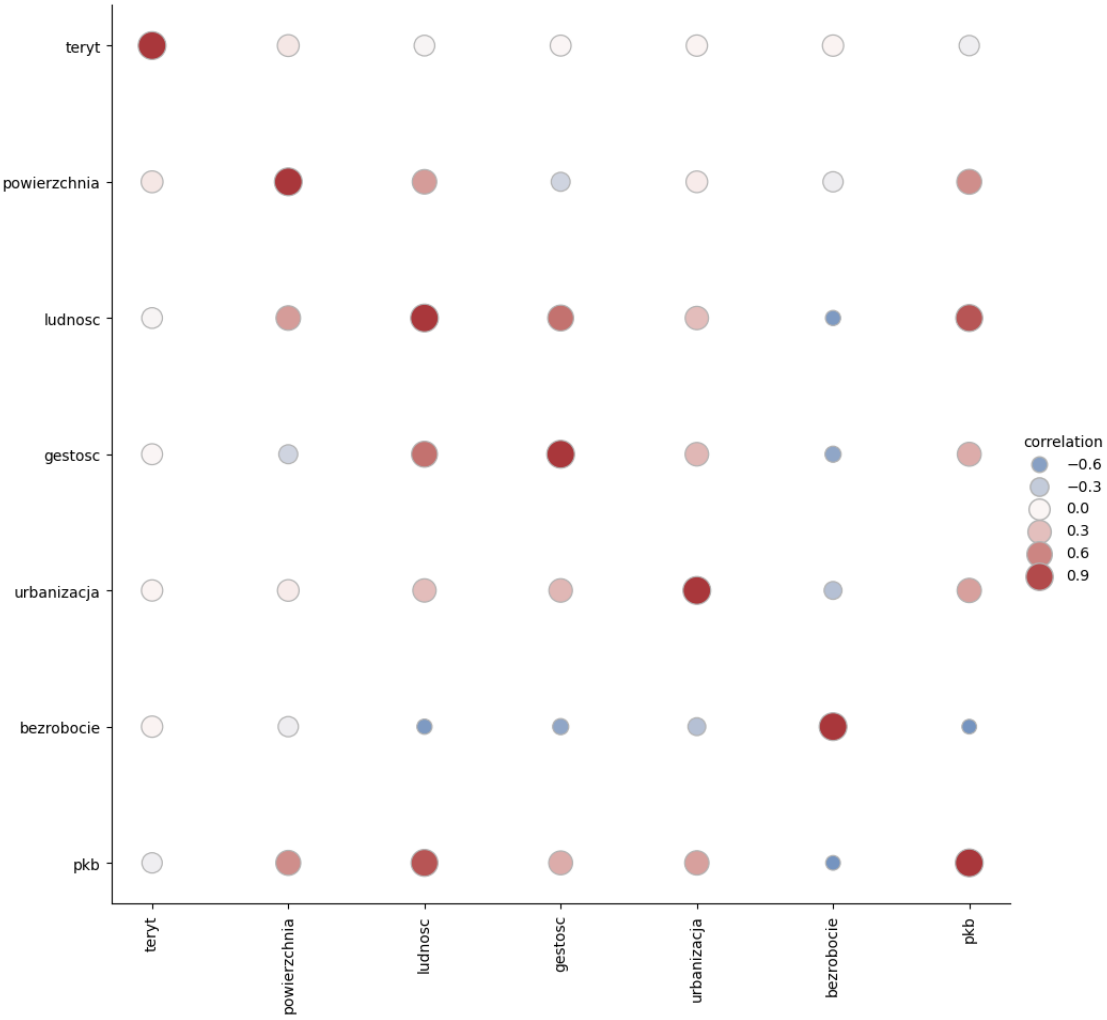
[207]:
```
import seaborn as sns

sns.barplot(x="bezrobocie", y="region", data=woj, orient="h")
```

[207]:
```
<AxesSubplot:xlabel='bezrobocie', ylabel='region'>
```



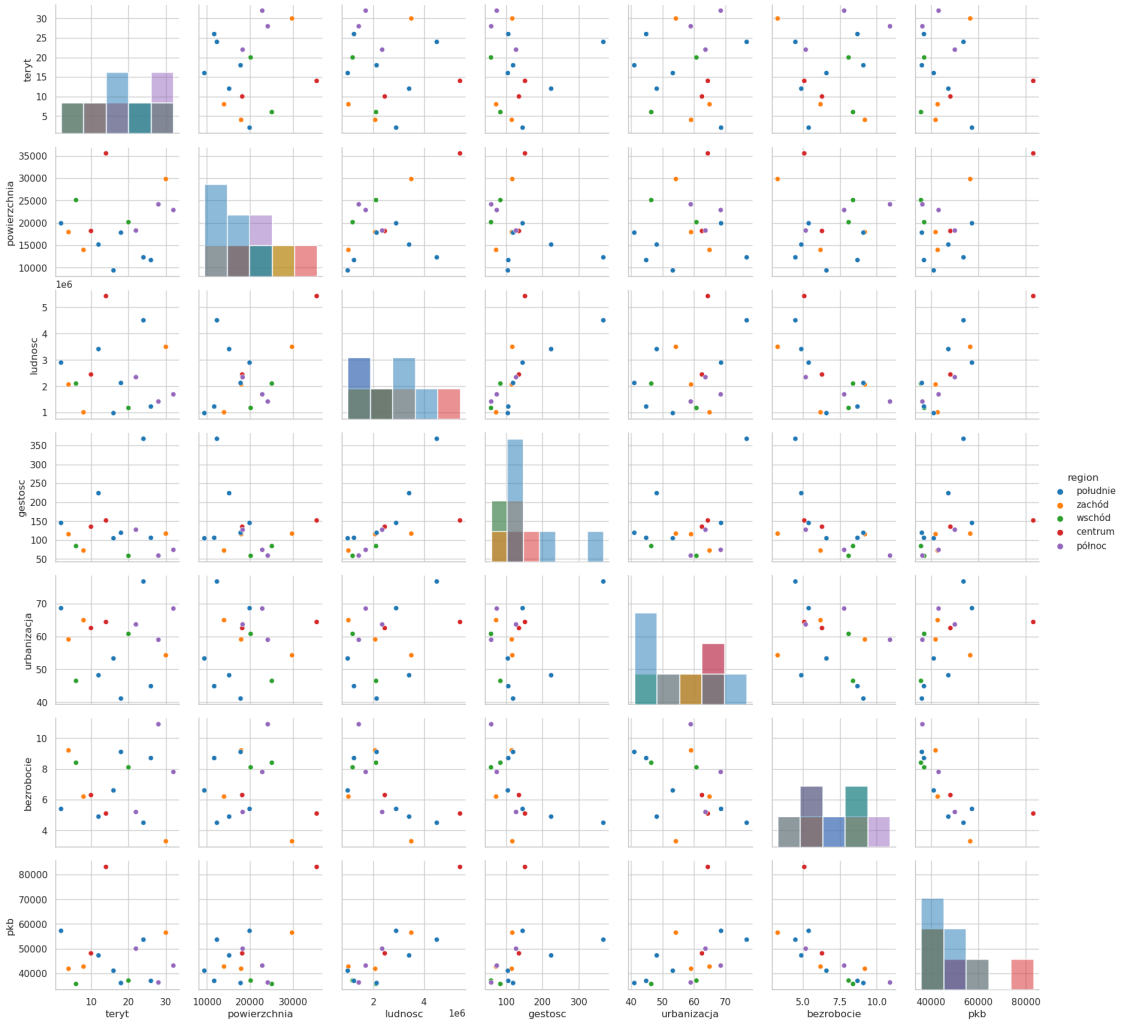
[208]:
```
corr_mat = woj.corr().stack().reset_index(name="correlation")
g = sns.relplot(
    data=corr_mat,
    x="level_0", y="level_1", hue="correlation", size="correlation",
    palette="vlag", hue_norm=(-1, 1), edgecolor=".7",
    height=10, sizes=(50, 350), size_norm=(-1, 1),
)

g.set(xlabel="", ylabel="", aspect="equal")
g.despine(left=False, bottom=False)
g.ax.margins(.05)
```

```
for label in g.ax.get_xticklabels():
    label.set_rotation(90)
for elem in g.legend.legendHandles:
    elem.set_edgecolor(".7")
```



[209]:
```
sns.set_style("whitegrid")
sns.set_context("notebook") # paper, notebook, talk, poster.
g = sns.PairGrid(woj, hue="region")
g.map_diag(sns.histplot)
g.map_offdiag(sns.scatterplot)
g.add_legend()
```

[209]: <seaborn.axisgrid.PairGrid at 0x7f66402f7850>



9 Prawie podsumowanie

Python ma swoje wady. Często, gdy kluczowa jest wydajność, w środowiskach Big Data będziemy korzystali z języków natywnych, *first class citizen* dla tych rozwiązań (Java, Scala, Kotlin). Bywa również tak, że niektóre narzędzia nie dają nam w tym zakresie większego wyboru.

Jeśli jednak zależy nam na:

- szybkim prototypowaniu rozwiązania
- obróbce końcowych danych z uwzględnieniem ich złożonej wizualizacji
- przetwarzaniu w sposób niedostępny za pomocą innych języków programowania (a dostępny w Pythonie dzięki bogactwu bibliotek)

wówczas Python jest trudny do zastąpienia, w szczególności w tych dwóch ostatnich aspektach.

[ ]: