Python

Technologie Informacyjne i Komunikacyjne R

Michał Parniak

mparniak@fuw.edu.pl www.qodl.eu

2026/27

Na podstawie materiałów: Katarzyna Grzelak, Krzysztof Miernik, Paweł Klimczewski

Po co nam Python i co nas czeka

1. **Podstawy:** kalkulator, zmienne, `if`, pętle.
2. **Struktura:** funkcje, moduły, listy, słowniki, pliki, wyjątki.
3. **Liczby i wykresy:** NumPy, Matplotlib, liczby zespolone.
4. **Analiza danych:** dopasowania, niepewności, χ^2.

Potem sprawdzian, a po nim sterowanie przyrządami.

Cel

Na końcu semestru napiszesz program, który sam steruje generatorem i oscyloskopem, zmierzy filtr RC, opracuje dane i przygotuje wykresy do sprawozdania.

Każdy slajd to ściąga: wolno z niej korzystać na ćwiczeniach i na sprawdzianie.

Jak uruchamiać Pythona

Sesja interaktywna: do prób.

```
$ python3
>>> 2 + 2
4
>>> exit()
```

Skrypt: program w pliku.

```
$ cat witaj.py
print("Witaj, atomie!")
$ python3 witaj.py
Witaj, atomie!
```

- Pisz w **VS Code**: podpowiedzi, kolory, przycisk uruchamiania.
- Python jest językiem **interpretowanym**: nie ma kompilacji, błąd wychodzi dopiero, gdy wykonanie dojdzie do tej linii.
- Notatnik **Jupyter** działa jak notatnik Mathematica (Shift+Enter).
- Dokumentacja: `https://docs.python.org/3/`, a w sesji `help(print)`.

Python jako kalkulator

```
>>> 7 / 2
3.5
>>> 7 // 2
3
>>> 7 % 2
1
>>> 2 ** 10
1024
>>> -4 ** 2
-16
>>> (-4) ** 2
16
```

```
>>> import math
>>> math.sqrt(2)
1.4142135623730951
>>> math.sin(math.pi / 2)
1.0
>>> math.log(math.e), math.log10(1000)
(1.0, 3.0)
```

- / to dzielenie, // całkowite, % reszta, ** potęga (**nie** ^).
- Funkcje matematyczne są w module `math`.

Zmienne i typy

```
>>> n = 87
>>> type(n)
<class 'int'>
>>> lam = 780e-9
>>> type(lam)
<class 'float'>
>>> atom = "Rb"
>>> type(atom)
<class 'str'>
>>> jest_zimny = True
>>> z = 3 + 4j
>>> abs(z)
5.0
```

- Zmiennej nie deklarujemy: typ ma wartość, nie nazwa.
- `int` ma dowolną wielkość; `float` ma około 16 cyfr.
- Dlatego liczb rzeczywistych nie porównujemy przez ==:

```
>>> 0.1 + 0.2
0.30000000000000004
>>> 0.1 + 0.2 == 0.3
False
>>> math.isclose(0.1 + 0.2, 0.3)
True
```

Napisy i formatowanie

```
>>> atom = "Rb"
>>> atom + "-87"
'Rb-87'
>>> len(atom), atom.lower(), atom * 3
(2, 'rb', 'RbRbRb')
>>> "87;3/2;6.83".split(";")
['87', '3/2', '6.83']
>>> tau = 26.2345
>>> f"tau = {tau:.2f} ns"
'tau = 26.23 ns'
>>> f"{1234.5:.2e} | {7:4d} | {tau:8.3f}"
'1.23e+03 |    7 |   26.235'
```

`f"..."` wstawia wartości w klamry. Po dwukropku format: `.2f` dwa miejsca po przecinku, `.2e` notacja naukowa, `8.3f` szerokość 8, `4d` liczba całkowita. Tego będziemy używać do budowania rozkazów dla przyrządów: `f"frequency {f:.2e}"`.

Wejście, wyjście, instrukcja `if`

```
a = int(input("Podaj liczbę: "))
if a > 0:
    print("dodatnia")
elif a == 0:
    print("zero")
else:
    print("ujemna")
print("koniec")
```

- `input` zawsze zwraca napis: trzeba go zamienić (`int`, `float`).
- Blok tworzy **wcięcie** (4 spacje), po dwukropku.
- Porównania: ==, !=, <, <=, >, >=.
- Łączenie: `and`, `or`, `not`.

Pętle

for: znamy liczbę powtórzeń.

```
>>> suma = 0
>>> for n in range(1, 5):
...     suma += n
>>> suma
10
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> list(range(2, 10, 3))
[2, 5, 8]
```

`range` **nie obejmuje** końca.

while: powtarzamy aż do warunku.

```
>>> atomy = 1000.0
>>> kroki = 0
>>> while atomy > 1:
...     atomy /= 2
...     kroki += 1
>>> kroki
10
```

`break` kończy pętlę, `continue` przechodzi do następnego obrotu. Tak właśnie program czeka na koniec pomiaru:
`while status != 'STOP':`

Czytanie komunikatów o błędach

```
$ python3 program.py
Traceback (most recent call last):
  File "program.py", line 3, in <module>
    print(energia(0))
  File "program.py", line 2, in energia
    return -13.6 / n ** 2
ZeroDivisionError: division by zero
```

Czytamy **od dołu**: co się stało, potem gdzie.

`SyntaxError`	literówka, brak dwukropka lub nawiasu
`IndentationError`	złe wcięcie
`NameError`	nieznana nazwa: literówka albo zmienna jeszcze nie istnieje
`TypeError`	zły typ: `"3" + 4`
`ValueError`	zła wartość: `int("abc")`

Funkcje

```
>>> def energia(n, ry=13.6057):
...     """Energia poziomu wodoru [eV]."""
...     return -ry / n ** 2
>>> energia(2)
-3.401425
>>> energia(n=1, ry=13.6)
-13.6
```

Funkcję można przekazać jako argument:

```
>>> def tabela(f, punkty):
...     return [f(x) for x in punkty]
>>> tabela(energia, [1, 2])
[-13.6057, -3.401425]
>>> tabela(lambda x: x ** 2, [1, 2])
[1, 4]
```

- `return` oddaje wynik; `print` tylko wypisuje. Funkcja, która liczy, **nie powinna** wypisywać.
- Wartości domyślne: `ry=13.6057`.
- Pierwszy napis to opis (`help(energia)`).
- Zmienne z wnętrza funkcji nie istnieją na zewnątrz.
- `lambda` to krótka funkcja bez nazwy, jak `#&` w Mathematica.

Moduły

Plik `atom.py`:

```
RY = 13.6057

def energia(n):
    return -RY / n ** 2

if __name__ == '__main__':
    print(energia(1))   # test
```

Plik `program.py` obok:

```
import atom

print(atom.energia(2))
print(atom.RY)
```

```
import numpy as np         # skrót
from math import pi, sqrt # wybrane
```

- Moduł to zwykły plik `.py`; można go wykonać albo zaimportować.
- Blok `if __name__ == '__main__':` wykonuje się tylko, gdy uruchamiamy plik bezpośrednio, nie przy imporcie.
- Nie pisz `from numpy import *`: nie wiadomo, skąd co się wzięło.
- Stałe piszemy WIELKIMI literami, na początku pliku.

Listy

```
>>> p = [2, 3, 5, 7, 11]
>>> p[0], p[-1], len(p)
(2, 11, 5)
>>> p[1:3]
[3, 5]
>>> p[::2]
[2, 5, 11]
>>> p.append(13)
>>> sum(p), max(p), 5 in p
(41, 13, True)
>>> sorted([3, 1, 2], reverse=True)
[3, 2, 1]
```

Indeksy od **zera**; `p[1:3]` bez elementu 3.

Wyrażenie listowe:

```
>>> [n ** 2 for n in range(5)]
[0, 1, 4, 9, 16]
>>> [n for n in p if n % 3 == 1]
[7, 13]
```

Dwa przydatne narzędzia:

```
>>> for i, x in enumerate("ab"):
...     print(i, x)
0 a
1 b
>>> for a, b in zip([1, 2], "xy"):
...     print(a, b)
1 x
2 y
```

Pułapka: przypisanie nie kopiuje

```
>>> a = [1, 2, 3]
>>> b = a
>>> b.append(4)
>>> a
[1, 2, 3, 4]
>>> c = a.copy()
>>> c.append(5)
>>> a
[1, 2, 3, 4]
```

- Nazwa to etykieta na obiekcie: `b = a` dokleja drugą etykietę, kopii nie ma.
- Listy i słowniki można zmieniać (*mutable*), liczby i napisy nie.
- Funkcja, która dostanie listę, może ją zmienić: tak działa `results.append(...)` w programie pomiarowym.

Krotki i słowniki

Krotka: niezmienna lista.

```
>>> punkt = (3.0, 4.0)
>>> x, y = punkt
>>> a, b = 1, 2
>>> a, b = b, a
>>> a
2
```

Funkcja może zwrócić kilka wartości: tak naprawdę zwraca krotkę.

Słownik: wartości dostępne po nazwie.

```
>>> rb = {"masa": 87, "spin": 1.5}
>>> rb["masa"]
87
>>> rb["tau"] = 26.2
>>> sorted(rb)
['masa', 'spin', 'tau']
>>> "spin" in rb
True
>>> for k, v in sorted(rb.items()):
...     print(k, v)
masa 87
spin 1.5
tau 26.2
```

Słowniki w słownikach i format JSON

```
>>> pomiar = {
...     "freq": 100.0,
...     "ch1": {"scale": 0.5,
...             "u": [0.1, 0.2, 0.3]},
... }
>>> pomiar["ch1"]["u"][1]
0.2
>>> import json
>>> json.dumps(pomiar["ch1"])
'{"scale": 0.5, "u": [0.1, 0.2, 0.3]}'
```

Zapis i odczyt z pliku:

```
with open("w.json", "w") as f:
    json.dump(wyniki, f)

with open("w.json") as f:
    wyniki = json.load(f)
```

JSON zapisuje liczby, napisy, listy i słowniki, a plik da się przeczytać w edytorze. Właśnie w tej postaci program pomiarowy zapisuje przebiegi z oscyloskopu.

Pliki tekstowe

```
with open("wyniki.txt", "w") as plik:
    plik.write("f U\n")
    for f, u in zip(czest, napiecia):
        plik.write(f"{f} {u}\n")

with open("izotopy.txt") as plik:
    for linia in plik:
        if linia.startswith("#"):
            continue
        pola = linia.strip().split(";")
        print(pola[0], float(pola[3]))
```

- `"r"` czytanie (domyślnie), `"w"` zapis od nowa, `"a"` dopisywanie.
- `with` **sam zamknie plik**, nawet jeśli w środku wystąpi błąd.
- Z pliku czytamy napisy: liczby trzeba zamienić.
- Dla tabel liczb jest wygodniejsze `np.loadtxt`.

Wyjątki

```
>>> def odwrotnosc(x):
...     try:
...         return 1 / x
...     except ZeroDivisionError:
...         return math.inf
>>> odwrotnosc(4), odwrotnosc(0)
(0.25, inf)
```

`finally` wykonuje się **zawsze**:

```
wyniki = []
try:
    zmierz(wyniki)      # może się nie udać
finally:
    zapisz(wyniki)      # dane ocaleją
```

- Błąd w Pythonie to **wyjątek**: przerywa funkcję i wędruje w górę, aż ktoś go złapie.
- Łap tylko to, co umiesz obsłużyć, i podaj rodzaj błędu. Samo `except:` ukrywa literówki.
- Własny błąd: `raise ValueError("n < 0")`.
- `try/finally` to podstawa programu pomiarowego: kilkanaście minut pomiaru nie może przepaść przez jeden błąd.

NumPy: tablice

```
>>> import numpy as np
>>> a = np.array([1.0, 2.0, 3.0])
>>> a * 2 + 1
array([3., 5., 7.])
>>> a ** 2
array([1., 4., 9.])
>>> np.sin(0 * a)
array([0., 0., 0.])
>>> a * a
array([1., 4., 9.])
>>> float(a @ a)
14.0
>>> [1, 2] * 2
[1, 2, 1, 2]
```

```
>>> np.linspace(0, 1, 5)
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
>>> np.arange(0, 1, 0.25)
array([0.  , 0.25, 0.5 , 0.75])
>>> np.geomspace(1, 100, 3)
array([  1.,  10., 100.])
```

- Działania obejmują **wszystkie elementy**, bez pętli i o wiele szybciej.
- `linspace`: podajemy liczbę punktów. `geomspace`: punkty równomierne na skali logarytmicznej.

Maski, statystyka, tablice dwuwymiarowe

```
>>> y = np.array([1.2, -0.5, 3.3, -2.1])
>>> y > 0
array([ True, False,  True, False])
>>> y[y > 0]
array([1.2, 3.3])
>>> int(np.sum(y > 0))
2
>>> np.where(y < 0, 0, y)
array([1.2, 0. , 3.3, 0. ])
```

`y.mean()`, `y.std(ddof=1)`, `y.min()`, `np.sum(y)`.

```
>>> m = np.array([[1, 2, 3],
...               [4, 5, 6]])
>>> m.shape
(2, 3)
>>> m[0, :], m[:, 1]
(array([1, 2, 3]), array([2, 5]))
>>> m.mean(axis=0)
array([2.5, 3.5, 4.5])
```

Tabela z pliku, kolumny do zmiennych:

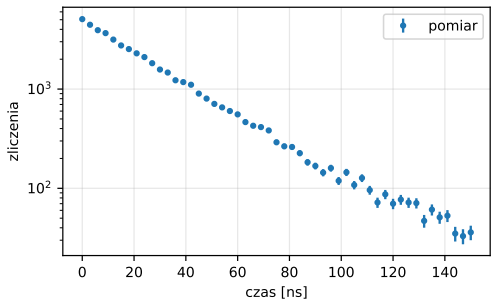
```
t, n, u = np.loadtxt("zanik.dat",
                     unpack=True)
```

Matplotlib: wykres do sprawozdania

```
import matplotlib.pyplot as plt
import numpy as np

t, n, u = np.loadtxt("zanik.dat",
                     unpack=True)

fig, ax = plt.subplots()
ax.errorbar(t, n, yerr=u, fmt="o",
            label="pomiar")
ax.set_yscale("log")
ax.set_xlabel("czas [ns]")
ax.set_ylabel("zliczenia")
ax.legend()
fig.savefig("zanik.pdf")
plt.show()
```



Linie: `ax.plot(x, y, "r--")`, punkty: `ax.plot(x, y, "o")`. Do LaTeX-a zapisuj PDF. Opisy osi z jednostkami są obowiązkowe. Wzory: `r"$\tau$ [ns]"`.

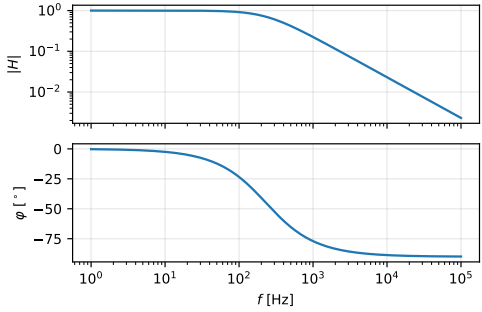
Liczby zespolone i kilka wykresów naraz

```
>>> z = 3 + 4j
>>> z.real, z.imag, abs(z)
(3.0, 4.0, 5.0)
>>> z * 1j
(-4+3j)
>>> float(np.degrees(np.angle(1j)))
90.0
```

Filtr RC, działa na całej tablicy:

```
R, C = 1e3, 690e-9
f = np.geomspace(1, 1e5, 300)
H = 1 / (1 + 2j * np.pi * f * R * C)

fig, (gora, dol) = plt.subplots(
    2, 1, sharex=True)
gora.loglog(f, np.abs(H))
dol.semilogx(f, np.degrees(np.angle(H)))
```



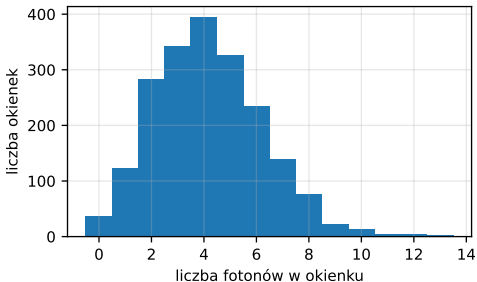
Jednostka urojona to `1j`. Moduł to wzmocnienie, argument to przesunięcie fazy: dlaczego tak, wyjaśnimy na wykładzie o elektronice.

Liczby losowe i histogramy

```
rng = np.random.default_rng(1)
rng.random(3)            # [0, 1)
rng.normal(0, 1, 100)    # Gauss
rng.poisson(4.2, 2000)   # Poisson

fotony = np.loadtxt("fotony.txt")
fig, ax = plt.subplots()
ax.hist(fotony,
        bins=np.arange(-0.5, 14.5))
```

- Ziarno (1) daje powtarzalne wyniki.
- Fotony z lasera: rozkład Poissona, wariancja równa średniej.



Dla liczb całkowitych granice przedziałów ustawiamy na połówkach. Bez tego histogram ma puste przedziały i dziwne schodki.

Dopasowanie: `curve_fit`

```
from scipy.optimize import curve_fit

def model(t, n0, tau, tlo):
    return n0 * np.exp(-t / tau) + tlo

p, cov = curve_fit(model, t, n,
                   p0=(5000, 20, 10),
                   sigma=u,
                   absolute_sigma=True)
niepewnosci = np.sqrt(np.diag(cov))
print(p[1], niepewnosci[1])
```

Szukamy parametrów, które minimalizują

$$\chi^2 = \sum_i \frac{\left(y_i - f\left(x_i; \vec{a}\right)\right)^2}{\sigma_i^2}.$$

- Pierwszy argument modelu to x, dalej parametry.
- p0: punkt startowy.
- cov: macierz kowariancji.

Na przekątnej cov stoją kwadraty niepewności. Elementy poza nią mówią o korelacji parametrów.

`sigma` i `absolute_sigma`: co naprawdę liczymy

`curve_fit(f, x, y, p0)`	wszystkie punkty równie ważne; niepewności z rozrzutu wokół krzywej
`sigma=u`	waga punktów $1/\sigma_i^2$, ale liczy się tylko **stosunek** σ_i: niepewności są przeskalowane tak, by $\chi^2/\mathrm{ndf} = 1$
`sigma=u, absolute_sigma=True`	σ_i traktujemy dosłownie jako niepewności pomiaru

Którą wersję wybrać?

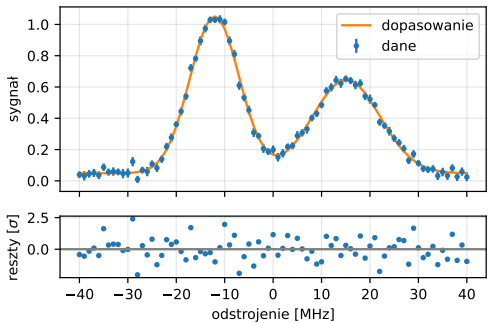
Jeśli wiemy, skąd biorą się σ_i (np. $\sqrt{N}$ dla zliczeń): `absolute_sigma=True`, a potem sprawdzamy χ^2. Jeśli znamy tylko względną jakość punktów albo wcale: bez tej opcji.

Ważne: σ_i opisuje **niezależny** rozrzut punktów. Błąd wspólny dla wszystkich punktów (np. niedokładne wzmocnienie oscyloskopu) nie uśrednia się i nie wolno go tak wpisywać.

Czy dopasowanie jest dobre? χ^2 i reszty

```
reszty = (y - model(x, *p)) / s
chi2 = np.sum(reszty ** 2)
ndf = len(x) - len(p)
print(chi2 / ndf)
```

- $\chi^2/\text{ndf} \approx 1$: model i niepewności zgodne z danymi.
- $\chi^2/\text{ndf} \gg 1$: zły model albo niedoszacowane σ_i.
- $\chi^2/\text{ndf} \ll 1$: przeszacowane σ_i.
- Reszty powinny wyglądać jak szum wokół zera. Struktura w resztach to sygnał, że modelowi czegoś brakuje.
- *p rozpakowuje tablicę na osobne argumenty.



Niepewność wielkości pochodnych

Różnica położeń dwóch linii, parametry m (indeks 1) i n (indeks 4):

```
d = p[4] - p[1]
u_d = np.sqrt(cov[1, 1] + cov[4, 4] - 2 * cov[1, 4])
```

Ogólnie, dla $g(\vec{a})$ i gradientu ∇g:

$$\sigma_g^2 = \nabla g^T \Sigma \nabla g = \sum_{ij} \frac{\partial g}{\partial a_i} \frac{\partial g}{\partial a_j} \Sigma_{ij}.$$

```
gradient = np.array([...])
u_g = np.sqrt(gradient @ cov @ gradient)
```

Korelacja: $\rho_{ij} = \Sigma_{ij} / \sqrt{\Sigma_{ii} \Sigma_{jj}}$. Pominięcie wyrazów pozadiagonalnych daje zły wynik, gdy parametry są skorelowane. Podstawy: wykład z Analizy Niepewności Pomiarowych.

Amplituda i faza sinusoidy: dwie metody

Dopasowanie. Znamy częstotliwość f:

```
def sinus(t, a, phi, b):
    return a * np.sin(
        2 * np.pi * f * t + phi) + b

p, cov = curve_fit(sinus, t, u,
    p0=(np.ptp(u) / 2, 0, 0))
```

`np.ptp` to maksimum minus minimum. Jeśli wyjdzie $a < 0$, zmień znak a i dodaj π do fazy.

Rzut (detekcja fazoczuła, *lock-in*).

```
z = 2 * np.mean(
    u * np.exp(-2j * np.pi * f * t))
amplituda = abs(z)
faza = np.angle(z)
```

$$\frac{2}{T}\int_0^T U_0 \cos(\omega t + \varphi)\, e^{-\mathrm{i}\omega t}\,\mathrm{d}t = U_0 e^{\mathrm{i}\varphi}$$

Działa tylko dla **całkowitej liczby okresów**: inaczej wynik jest zafałszowany. Tu faza liczona jest względem cosinusa.

Najczęstsze kłopoty z dopasowaniem

Krzywa wcale nie pasuje	narysuj model dla `p0` na tle danych: dobry start to połowa sukcesu
`OptimizeWarning`, `inf` w `cov`	parametr nie wpływa na model albo dwa parametry robią to samo
`RuntimeError: ... maxfev`	zły punkt startowy albo zły model
Parametr ujemny, a miał być dodatni	sprawdź znak w modelu; szerokość σ i $-\sigma$ są równoważne
Dane rzędu 10^{-9}	przeskaluj jednostki (ns zamiast s, MHz zamiast Hz)
Dzielenie przez zero w `sigma`	punkt z $\sigma_i = 0$, np. $\sqrt{N}$ dla $N = 0$
Faza skacze o 2π	`np.angle(np.exp(1j * faza))` zawija do $(-\pi, \pi]$

Co dalej

- Po sprawdzianie: **pomiar sterowany przez komputer**.
- Lutujesz filtr RC i piszesz program, który za pomocą generatora i oscyloskopu zmierzy jego charakterystykę.
- Przyda się wszystko z tych czterech zajęć: `f"..."` do rozkazów, pętle, słowniki i JSON na wyniki, `try/finally` na awarie, liczby zespolone do opisu filtru, dopasowania do wyznaczenia RC, Matplotlib do sprawozdania.