# Technologie informacyjne i komunikacyjne

Wykład 9, dn. 11.05.2026

Źródło obrazka: pngtree

# Wstęp do funkcji `def` w Pythonie

W programowaniu funkcje stanowią podstawowy mechanizm abstrakcji i modularyzacji kodu. Pozwalają one:

- wydzielać logiczne fragmenty programu,
- wielokrotnie wykorzystywać ten sam kod,
- zwiększać czytelność i skalowalność aplikacji,
- upraszczać analizę oraz przetwarzanie danych.

W kontekście analizy danych funkcje są szczególnie istotne, ponieważ umożliwiają automatyzację operacji wykonywanych na zbiorach danych, implementację algorytmów statystycznych oraz tworzenie własnych procedur analitycznych.

W języku Python funkcje definiuje się za pomocą słowa kluczowego: `def`

# Czym jest funkcja?

Ogólna składnia funkcji w Pythonie:

```
def nazwa_funkcji(parametry):
    instrukcje
    return wynik
```

Elementy definicji:

- `def` - słowo kluczowe rozpoczynające definicję funkcji,
- `nazwa_funkcji` - identyfikator funkcji,
- `parametry` - dane wejściowe przekazywane do funkcji,
- `return` - instrukcja zwracająca wynik działania funkcji.

# Najprostsza funkcja

```
def przywitaj():
    print("Witaj na zajęciach z Pythona!")
```

Wywołanie funkcji:

```
przywitaj()
```

Wynik:

```
Witaj na zajęciach z Pythona!
```

Należy zauważyć, że sama definicja funkcji nie powoduje wykonania kodu. Funkcja zostaje wykonana dopiero po jej wywołaniu.

# Funkcja zwracająca wartość (`return`)

Bardzo ważny element analizy danych - funkcje często zwracają wynik obliczeń.

```
def suma(a, b):
    return a + b
```

Użycie:

```
wynik = suma(5, 7)
print(wynik)
```

Wynik:

```
12
```

# Funkcja z parametrem

Parametry umożliwiają przekazywanie danych wejściowych do funkcji.

```
def pole_kwadratu(a):
    return a ** 2
```

Wywołanie:

```
wynik = pole_kwadratu(5)
print(wynik)
```

Wynik:

```
25
```

W powyższym przykładzie:

- `a` jest parametrem formalnym,
- `5` jest argumentem przekazywanym do funkcji.

# Wartość zwracana przez funkcję

Instrukcja return kończy działanie funkcji i zwraca wynik.

```
def suma(a, b):
    return a + b

x = suma(10, 15)

print(x)
```

Wynik:

```
25
```

Funkcje bez instrukcji `return` zwracają domyślnie wartość:

```
None
```

# Funkcje w analizie danych

W analizie danych funkcje są używane np. do: liczenia średniej, filtrowania danych, czyszczenia danych, tworzenia wykresów, automatyzacji obliczeń.

Przykład:

```
def srednia(lista):
    return sum(lista) / len(lista)
```

Użycie:

```
oceny = [4, 5, 3, 4, 5]

print(srednia(oceny))
```

Wynik:

```
4.2
```

# Funkcje z wieloma wartościami zwracanymi

Python umożliwia zwracanie wielu wartości jednocześnie.

```
def minimum_maksimum(dane):
    return min(dane), max(dane)

a, b = minimum_maksimum([1, 5, 9, 2])

print(a)

print(b)
```

# Parametry domyślne

Funkcje mogą posiadać wartości domyślne parametrów.

```
def potega(x, n=2):
    return x ** n

print(potega(5))

print(potega(5, 3))
```

Wynik:

```
25

125
```

# Funkcje i biblioteka NumPy

W analizie danych często używamy bibliotek.

Najpopularniejsza:

- NumPy — obliczenia numeryczne,
- pandas — analiza danych tabelarycznych,
- matplotlib — wykresy.

# Funkcje i biblioteka NumPy

Przykład wykorzystania funkcji wraz z biblioteką NumPy:

```
import numpy as np

def statystyki(dane):
    srednia = np.mean(dane)
    mediana = np.median(dane)
    odchylenie = np.std(dane)

    return srednia, mediana, odchylenie

wyniki = [10, 15, 20, 25, 30]

s, m, o = statystyki(wyniki)

print("Średnia:", s)
print("Mediana:", m)
print("Odchylenie:", o)
```

# Funkcje wyższego rzędu

Funkcje w Pythonie są obiektami pierwszej klasy, co oznacza, że mogą być: przekazywane jako argumenty, zwracane przez inne funkcje, przypisywane do zmiennych.

Przykład:

```
def wykonaj(funkcja, x):
    return funkcja(x)

def kwadrat(x):
    return x ** 2

print(wykonaj(kwadrat, 5))
```

# Dobre praktyki programistyczne

Zalecenia:

- stosowanie opisowych nazw funkcji,
- tworzenie funkcji wykonujących jedno zadanie,
- unikanie nadmiernie długich funkcji,
- dokumentowanie funkcji,
- stosowanie typowania.

Przykład:

```
def srednia(dane: list[float]) -> float:
    """
    Funkcja oblicza średnią arytmetyczną.
    """
    return sum(dane) / len(dane)
```

# Typowe błędy

Brak wcięć

Python wykorzystuje wcięcia do definiowania bloków kodu.

```
def test():

print("Błąd")
```

```
def test():

    print("Poprawnie")
```

# Typowe błędy

Brak wywołania funkcji

Sama definicja funkcji nic nie wykonuje.

```
def hello():
    print("Hi")
```

Dopiero:

```
hello()
```

uruchamia funkcję.

# Typowe błędy

Dzielenie przez zero

```
def srednia(dane):
    return sum(dane) / len(dane)
```

Dla pustej listy:

```
[]
```

otrzymamy:

```
ZeroDivisionError
```

Dlatego należy stosować walidację danych wejściowych.

# Lista w Pythonie (`list`)

Co to jest lista?

- Struktura danych przechowująca wiele elementów
- Elementy mogą mieć różne typy danych
- Lista jest uporządkowana i modyfikowalna

Tworzenie listy

```
owoce = ["jabłko", "banan", "gruszka"]

liczby = [1, 2, 3, 4]

mieszana = ["tekst", 10, True]
```

# Numerowanie elementów w liście

W Pythonie elementy listy mają indeksy, liczone od 0.

```
owoce = ["jabłko", "banan", "gruszka"]

print(owoce[0])  # jabłko
print(owoce[1])  # banan
print(owoce[2])  # gruszka
```

Indeksy ujemne

Można liczyć od końca listy:

```
print(owoce[-1])  # gruszka
print(owoce[-2])  # banan
```

# Numerowanie elementów w liście

```
owoce = ["jabłko", "banan", "gruszka"]
```

Schemat numerowania:

| **Element** | `jabłko` | `banan` | `gruszka` |
|---|---|---|---|
| **Indeks** | 0 | 1 | 2 |
| **Od końca** | -3 | -2 | -1 |

# Otrzymywanie zakresu w liście (slicing)

W Pythonie można pobierać część listy używając zakresu indeksów.

```
liczby = [10, 20, 30, 40, 50, 60]
print(liczby[1:4])  # [20, 30, 40]
```

Składnia:

```
lista[start:stop]
```

- `start` – indeks początkowy (włącznie)
- `stop` – indeks końcowy (bez tego elementu)

Przykłady:

```
print(liczby[:3])   # od początku do indeksu 3 → [10, 20, 30]
print(liczby[2:])   # od indeksu 2 do końca → [30, 40, 50, 60]
print(liczby[:])    # cała lista
```

# Otrzymywanie zakresu w liście (slicing)

Krok (co ile elementów)

```
print(liczby[0:6:2])  # co drugi element → [10, 30, 50]
```

Składnia:

```
lista[start:stop:krok]
```

Podsumowanie:

- pozwala wycinać fragmenty listy
- bardzo przydatne w analizie danych
- działa też z ujemnymi indeksami

# Długość listy w Pythonie

Do sprawdzenia liczby elementów w liście używa się funkcji len().

```
owoce = ["jabłko", "banan", "gruszka"]
print(len(owoce))  # 3
```

Jak to działa?

- `len(lista)` zwraca **liczbę elementów** w liście
- działa dla list o dowolnej długości

# Długość listy w Pythonie

Przykład:

| **Lista** | Wynik `len()` |
|---|---|
| `[1, 2, 3]` | `3` |
| `["a", "b"]` | `2` |
| `[]` | `0` |

Zastosowanie:

```
if len(owoce) > 0:
    print("Lista nie jest pusta")
```

# Podsumowanie

Funkcje stanowią fundamentalny element programowania w języku Python i odgrywają kluczową rolę w analizie danych.

Pozwalają one:

- tworzyć modularny kod,
- automatyzować obliczenia,
- implementować algorytmy analityczne,
- przetwarzać dane w sposób powtarzalny i skalowalny.

Znajomość funkcji jest niezbędna w dalszej pracy z:

- różnymi bibliotekami dostępnymi w Pythonie,
- uczeniem maszynowym,
- analizą statystyczną,
- sztuczną inteligencją.

# Za tydzień: Wizualizacja danych

Biblioteka matplotlib

```
import matplotlib.pyplot as plt
```

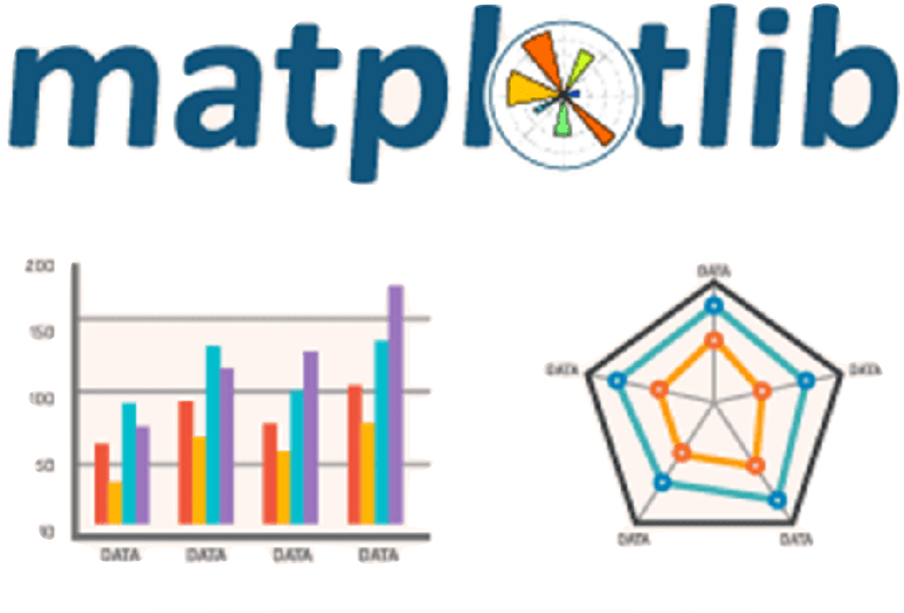
Prosty wykres

```
x = [1, 2, 3, 4]
y = [2, 4, 6, 8]

plt.plot(x, y)

plt.xlabel("X")
plt.ylabel("Y")
plt.title("Przykładowy wykres")

plt.show()
```



# matplotlib

Cheat sheet — Version 3.0.1

## Quick start

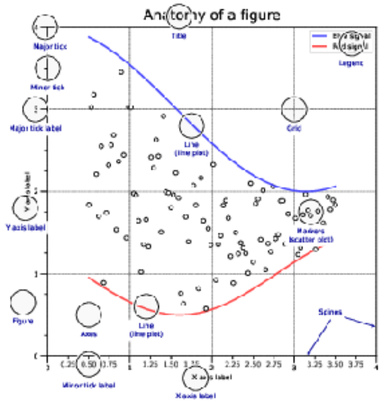
```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt


X = np.linspace(0, 2*np.pi, 100)
Y = np.cos(X)

fig, ax = plt.subplots()
ax.plot(X, Y, color='green')

fig.savefig("figure.pdf")
plt.show()
```

## Anatomy of a figure




## Subplots layout

```
subplot[s](rows, cols, ...)
fig, axs = plt.subplots(3, 3)

G = gridspec(rows,cols, ...)
ax = G[0, :]

ax.inset_axes(extent)

d=make_axes_locatable(ax)
ax = d.new_horizontal('10%')
```

## Getting help

- matplotlib.org
- github.com/matplotlib/matplotlib/issues
- discourse.matplotlib.org
- stackoverflow.com/questions/tagged/matplotlib
- https://gitter.im/matplotlib/matplotlib
- twitter.com/matplotlib
- Matplotlib users mailing list

## Basic plots

```
plot([X], Y, [fmt], ...)
X, Y, fmt, color, marker, linestyle

scatter(X, Y, ...)
X, Y, [s]izes, [c]olors, marker, cmap

bar[h](x, height, ...)
x, height, width, bottom, align, color

imshow(Z, ...)
Z, cmap, interpolation, extent, origin

contour[f]([X], [Y], Z, ...)
X, Y, Z, levels, colors, extent, origin

pcolormesh([X], [Y], Z, ...)
X, Y, Z, vmin, vmax, cmap

quiver([X], [Y], U, V, ...)
X, Y, U, V, C, units, angles

pie(X, ...)
Z, explode, labels, colors, radius

text(x, y, text, ...)
x, y, text, va, ha, size, weight, transform

fill_[between][x](...)
X, Y1, Y2, color, where
```

## Advanced plots

```
step(X, Y, [fmt], ...)
X, Y, fmt, color, marker, where

boxplot(X, ...)
X, notch, sym, bootstrap, widths

errorbar(X,Y,xerr,yerr, ...)
X, Y, xerr, yerr, fmt

hist(X, bins, ...)
X, bins, range, density, weights

violinplot(D, ...)
D, positions, widths, vert

barbs([X], [Y], U, V, ...)
X, Y, U, V, C, length, pivot, sizes

eventplot(positions, ...)
positions, orientation, lineoffsets

hexbin(X, Y, C, ...)
X, Y, C, gridsize, bins
```

## Scales

```
ax.set_[xy]scale(scale, ...)
```

linear — any values
log — values > 0
symlog — any values
logit — 0 < values < 1

## Projections

```
subplot(..., projection=p)
```

p='polar'
p='3d'

```
p=ccrs.Orthographic()
import cartopy.crs as ccrs
```

## Lines

```
linestyle or ls
```

"-" "--" "-." ":" (3,(0.01,2))

```
capstyle or dash_capstyle
```

"butt" "round" "projecting"

## Markers

markevery: 10 | (0, 1) | (25, 5) | (0, 25, 1)

## Colors

'Cn' 'x' 'name' (R,G,B,[A]) '#RRGGBB[AA]' 'x,y'

## Colormaps

```
plt.get_cmap(name)
```

Uniform: viridis, inferno, plasma
Sequential: Greys, YlOrBr, Wistia
Diverging: Spectral, coolwarm, RdGy
Qualitative: tab10, tab20
Cyclic: twilight

## Tick locators

```
from matplotlib import ticker
ax.[xy]axis.set_[minor|major]_locator(locator)

ticker.NullLocator()
ticker.MultipleLocator(0.5)
ticker.FixedLocator([0, 1, 5])
ticker.LinearLocator(numticks=3)
ticker.IndexLocator(base=0.5, offset=0.25)
ticker.AutoLocator()
ticker.MaxNLocator(n=4)
ticker.LogLocator(base=10, numticks=15)
```

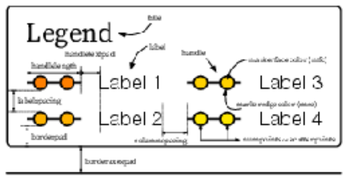
## Tick formatters

```
from matplotlib import ticker
ax.[xy]axis.set_[minor|major]_formatter(formatter)

ticker.NullFormatter()
ticker.FixedFormatter(['zero', 'one', 'two', ...])
ticker.FuncFormatter(lambda x, pos: "[%.2f]" % x)
ticker.FormatStrFormatter('>%d<')
ticker.ScalarFormatter()
ticker.StrMethodFormatter('{x}')
ticker.PercentFormatter(xmax=5)
```

## Ornaments

```
ax.legend(...)
handles, labels, loc, title, frameon
```




```
ax.colorbar(...)
mappable, ax, cax, orientation
```

```
ax.annotate(...)
text, xy, xytext, xycoords, textcoords, arrowprops
```


## Event handling

```
fig, ax = plt.subplots()
def on_click(event):
  print(event)
fig.canvas.mpl_connect(
  'button_press_event', on_click)
```

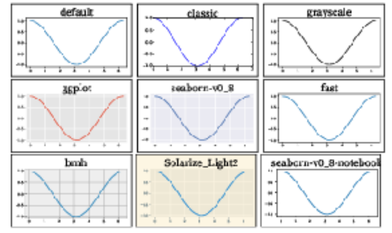
## Animation

```
import matplotlib.animation as mpla

T = np.linspace(0, 2*np.pi, 100)
S = np.sin(T)
line, = plt.plot(T, S)
def animate(i):
    line.set_ydata(np.sin(T+i/50))
anim = mpla.FuncAnimation(
    plt.gcf(), animate, interval=5)
plt.show()
```

## Styles

```
plt.style.use(style)
```




## Quick reminder

```
ax.grid()
ax.set_[xy]lim(vmin, vmax)
ax.set_[xy]label(label)
ax.set_[xy]ticks(ticks, [labels])
ax.set_[xy]ticklabels(labels)
ax.set_title(title)
ax.tick_params(width=10, ...)
ax.set_axis_[on|off]()

fig.suptitle(title)
fig.tight_layout()
plt.gcf(), plt.gca()
mpl.rc('axes', linewidth=1, ...)
[fig|ax].patch.set_alpha(0)
text=r'$\frac{e^{i\pi}}{2^n}$'
```

## Keyboard shortcuts

| | |
|---|---|
| ctrl + s Save | ctrl + w Close plot |
| r Reset view | f Fullscreen 0/1 |
| f View forward | b View back |
| p Pan view | o Zoom to rect |
| x X pan/zoom | y Y pan/zoom |
| g Minor grid 0/1 | G Major grid 0/1 |
| l X axis log/linear | _ Y axis log/linear |

## Ten simple rules

1. Know your audience
2. Identify your message
3. Adapt the figure
4. Captions are not optional
5. Do not trust the defaults
6. Use color effectively
7. Do not mislead the reader
8. Avoid "chartjunk"
9. Message trumps beauty
10. Get the right tool