

Laboratorium programistyczne 1 - wprowadzenie do Pythona

Projekt „Matematyka dla Ciekawych Świata”,
Piotr Morawiecki, Robert Paciorek

2025-01-02

Spis treści

1	Praca z Pythonem	1
1.1	Praca z Google Colab	2
1.2	Python jako kalkulator	2
1.3	Pętla for	3
2	Listy	4
2.1	Lista kolejnych liczb naturalnych	4
2.2	Operacje na listach	5
3	Tworzenie własnych funkcji	6
4	Typ logiczny	7
4.1	Instrukcja warunkowa if	7
4.2	Pętla while	9
5	Generowanie liczb losowych	9
6	Rysowanie wykresów	9
7	Zadania dodatkowe	11
8	Praca domowa nr 1	12

1 Praca z Pythonem

Na zajęciach będziemy programować w języku Python w wersji 3. Pythona można używać na jeden z dwóch sposobów:

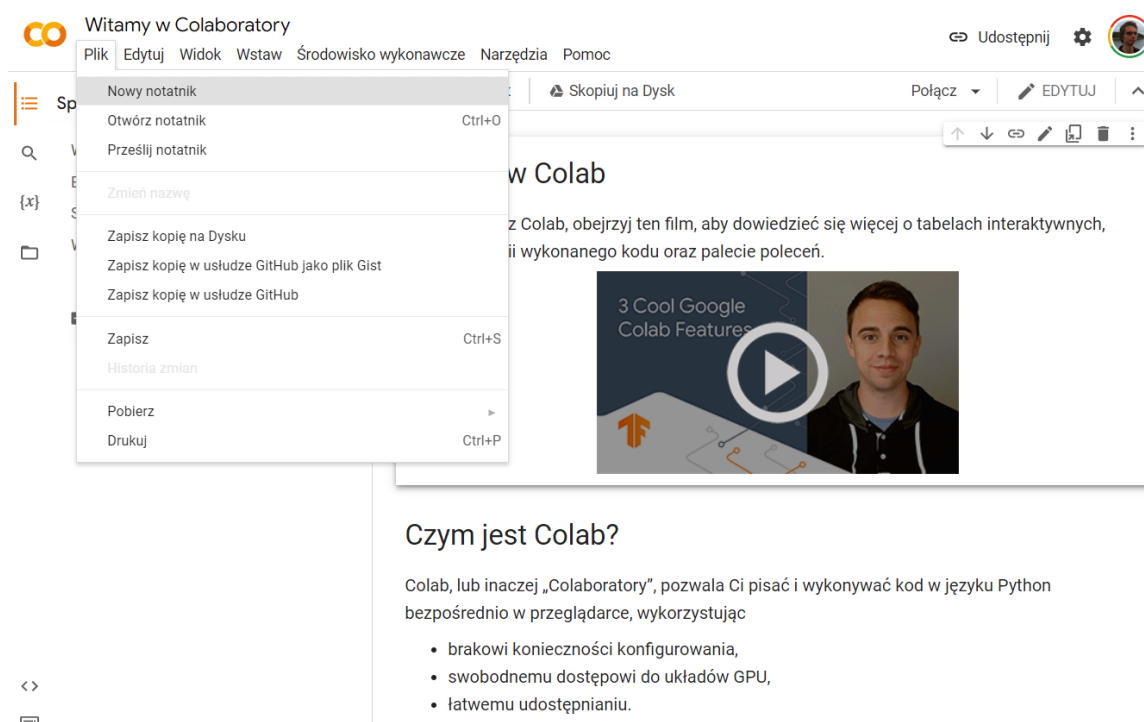
Zainstalowany interpreter Pythona. Interpreter Pythona może być zainstalowany na komputerze, z którego korzystamy. Można go bezpłatnie pobrać ze strony <https://www.python.org/downloads/> i zainstalować na swoich domowych komputerach. Zwróćcie uwagę, aby zainstalować wersję o numerze rozpoczynającym się od 3 (np. 3.x), a nie starszą, ale wciąż używaną wersję 2. Wersje te różnią się tak znacząco, że programy, które będziemy pisać na zajęciach nie będą działać w starszej, drugiej wersji Pythona.

Interpreter Pythona online. Istnieje wiele interpreterów Pythona online. Na zajęciach będziemy korzystali z interpretera Google Colab znajdującego się na stronie <https://colab.research.google.com>. Interpretera tego będziecie mogli również używać na swoich domowych komputerach bez instalacji żadnego dodatkowego oprogramowania — wystarczy dowolna przeglądarka i połączenie do internetu.

1.1 Praca z Google Colab

Jeśli nie posiadasz jeszcze konta Google to musisz je założyć poprzez stronę <https://accounts.google.com/>. Jeśli posiadasz już konto to otwórz stronę <https://colab.research.google.com/notebooks/welcome.ipynb?hl=pl>. Google Colaboratory (w skrócie Colab) pozwala pisać i wykonywać programy w Pythonie. Korzysta on z tzw. notatników (notebooków), w których zarówno można pisać fragmenty programów oraz wstawki tekstowe (które ułatwią Ci pisanie notatek lub objaśnianie poszczególnych części kodu prowadzącym). Po otwarciu powyższej strony będziecie mogli obejrzeć przykładowy notatnik, z wieloma praktycznymi informacjami na temat korzystania z Colab (patrz Rysunek 1).

Polecam zajrzeć do niego w domu, jednak teraz czas utworzyć nasz pierwszy notatnik. W tym celu w Plik → Nowy notatnik. W lewym górnym rogu ekranu znajduje się nazwa twojego notatnika. Możesz ją zmienić klikając na nazwie domyślnej i wpisując własną. Sugeruję, w nazwie pliku podać swoje imię i nazwisko, np. "Pierwsze kroki w Pythonie - Jan Kowalski" - ułatwi to prowadzącym sprawdzanie waszych skryptów.



Rysunek 1: Widok notatnika powitalnego dostępnego na stronie <https://colab.research.google.com/notebooks/welcome.ipynb?hl=pl>

W tak utworzonym notatniku możecie dodawać pola tekstowe klikając "+test" w górnym lewym rogu notatnika. W polu tekstowym możesz pisać własne notatki. Żeby napisać skrypt w Pythonie należy kliknąć "+kod".

Kod też możesz udostępnić prowadzącemu, np. wysyłając mu zadania domowe lub w przypadku uczestniczenia w laboratorium online. W tym celu kliknij "Udostępnij" znajdujące się w prawym górnym rogu okna. Następnie w sekcji "Pobierz link" możesz skopiować link i wysłać prowadzącemu zajęcia przez Discorda. Warto zmienić domyślną opcję "Przeglądający" na "Komentujący" lub w przypadku laboratoriów online na "Edytor". Wówczas prowadzący będzie mógł nanosić odpowiednio komentarze lub edytować wasz notebook.

1.2 Python jako kalkulator

Najprostszym sposobem użycia Pythona jest użycie jego jako kalkulatora — wpisujemy działanie do obliczenia, naciskamy shift+Enter (lub klikając trójkąt po lewej stronie pola z kodem) i poniżej otrzymujemy wynik działania. Przykładowe działania, które możemy wykonać w Pythonie znajdują się poniżej:

```
2 + 2 * 2
(2 + 2) * 2
2 ** 7
47 / 10
47 // 10
47 % 10
```

W powyższym przykładzie:

- Znak `**` oznacza podnoszenie do potęgi.
- Znak `/` oznacza dzielenie.
- Znak `//` oznacza dzielenie całkowite.
- Znak `%` oznacza branie reszty z dzielenia.

W oknie kodu możemy też wpisać kilka komend w osobnych liniach i wykonać je jednocześnie. Jednak wówczas zostanie wypisany jedynie wynik ostatniej komendy. Jeśli chcemy wyświetlić wyniki wszystkich operacji wyświetlić na ekranie należy umieścić dane działanie w komendzie `print`, np.:

```
print(2 + 2 * 2)
print((2 + 2) * 2)
```

Podobnie jak w kalkulatorze możemy korzystać z *pamięci*, w Pythonie możemy zapisywać wartości w *zmiennych*:

```
x = 3
y = 4
x**2 + y**2
```

W pierwszych dwóch liniach następuje *przypisanie* wartości 3 do zmiennej `x` oraz wartości 4 do zmiennej `y`. Od tej pory możemy korzystać z tych zmiennych, np. do obliczenia wartości wyrażenia $(x^2 + y^2)$.

Zadanie 1.2.1

Korzystając z Pythona jak z kalkulatora, znajdź wszystkie dziesięciocyfrowe potęgi dwójki.

1.3 Pętla `for`

Załóżmy, że chcemy obliczyć kwadraty wszystkich liczb od 1 do 5. Zgodnie z dotychczasową wiedzą, w tym celu musimy wykonać 5 działań:

```
print(1 * 1)
print(2 * 2)
print(3 * 3)
print(4 * 4)
print(5 * 5)
```

Widzimy jednak, że te działania są bardzo podobne i chciałoby się je wykonać „za jednym zamachem”. Do wykonywania wielokrotnie tego samego (lub podobnego) kodu służą pętle. Najprostszym rodzajem pętli jest pętla `for`, która dla danej *listy* i operacji do wykonania wykonuje tę operację po kolei na każdym elemencie listy.

Do wykonania powyższego zadania służy pętla `for` w następującej postaci:

```
for x in [1, 2, 3, 4, 5]:
    print(x * x)
```

Spróbuj przepisać tę pętlę do pola kodu i uruchomić. Po chwili powinien pojawić się wynik działania naszego programu:

```
1
4
9
16
25
```

Zwróć uwagę na dwie rzeczy:

- Na końcu pierwszej linijki jest dwukropek.
- Druga linijka musi być *wcięta*, tzn. rozpoczynać się od spacji, kilku spacji lub znaku tabulacji. Jeśli zapomnimy o wcięciu, interpreter zgłosi błąd i całą komendę wielolinijkową będziemy musieli pisać od początku (można pomóc sobie, naciskając strzałkę w górę i przeglądając stare komendy).

Ilekoć w niniejszych materiałach pojawią się dwie ramki, jedna obok drugiej, w lewej ramce znajdował się będzie kod programu, a w prawej efekt jego działania:

```
for x in [1, 2, 3, 4, 5]:
    print(x * x)
```

```
1
4
9
16
25
```

Powyższa pętla wypisała każdą liczbę w osobnej linijce. Dzieje się tak, ponieważ funkcja `print(...)` domyślnie przechodzi do następnej linii po każdym wywołaniu. Można jednak zmienić to zachowanie, dodając wewnątrz `print(...)` po przecinku końcówkę `end = X`, gdzie `X` to otoczony apostrofami ciąg znaków, który chcemy wypisywać zamiast przejścia do nowej linii.

Przykłady:

```
for x in [1, 2, 3, 4, 5]:
    print(x * x, end = ' ')
```

```
1 4 9 16 25
```

```
for x in [1, 2, 3, 4, 5]:
    print(x * x, end = '')
```

```
1491625
```

```
for x in [1, 2, 3, 4, 5]:
    print(x * x, end = ', ')
```

```
1, 4, 9, 16, 25,
```

Zadanie 1.3.1

Wykorzystując pętlę `for` oblicz oraz wypisz na ekran wartość wyrażenia $(1 + 1/n)^n$ dla $n = 1, 2, 3, 4, 5, 10, 100, 1000, 10000$.

2 Listy

2.1 Lista kolejnych liczb naturalnych

Często potrzebujemy, aby pętla przeszła po liście kilku kolejnych liczb naturalnych. W tym celu możemy oczywiście podać wprost kolejne elementy listy (tak jak w powyższym przykładzie), jednak istnieje wygodniejsze rozwiązanie, mianowicie polecenie `range()`:

```
for x in range(7):  
    print(x, end = ', ')
```

0, 1, 2, 3, 4, 5, 6,

```
for x in range(5, 10):  
    print(x, end = ', ')
```

5, 6, 7, 8, 9,

```
for x in range(10, 20, 3):  
    print(x, end = ', ')
```

10, 13, 16, 19,

Na powyższych przykładach widzimy, że polecenie `range()` występuje w trzech wersjach:

- `range(kon)` generuje listę kolejnych liczb od 0 (**włącznie**) do kon (**wyłącznie**).
- `range(pocz, kon)` generuje listę kolejnych liczb od pocz (**włącznie**) do kon (**wyłącznie**).
- `range(pocz, kon, krok)` generuje listę liczb od pocz (**włącznie**) do kon (**wyłącznie**), przeskakując w każdym kroku o krok.

Do zapamiętania: Wszystkie przedziały w Pythonie są domknięte z lewej strony i otwarte z prawej strony, tzn. zawierają swój lewy koniec i nie zawierają swojego prawego końca.

Zadanie 2.1.1

Korzystając z Pythona wypisz nieparzyste liczby niezerowe, których moduł jest mniejszy niż 20, w kolejności od liczb z najmniejszym modułem, do liczb z największym modułem, tzn.:

```
-1  
1  
-3  
3  
-5  
5  
itd.
```

2.2 Operacje na listach

Do tej pory listy traktowaliśmy głównie jako zbiór elementów po którym iterujemy. Zastosowanie list jest jednak znacznie szersze. Lista stanowi pewnego rodzaju kontener do przechowywania innych zmiennych, w których elementy zorganizowane są na zasadzie określenia ich (względnej) kolejności. Lista może zawierać elementy różnych typów.

Na listach możemy wykonywać m.in. operacje modyfikowania, czy też usuwania jej elementów:

```
l = [13, 17, 0, 8]  
l[0] = 15  
del l[2]  
l.append(5)  
print(l)  
for x in l:  
    print(x)
```

```
[15, 17, 8, 5]  
15  
17  
8  
5
```

W powyższym przykładzie widzimy:

- Modyfikację pierwszego elementu listy (`l[0] = 15`), z użyciem odwołania poprzez numer elementu. Elementy list numerujemy od zera. Ujemne wartości oznaczają numerowanie od końca listy, czyli -1

jest ostatnim elementem listy, -2 przedostatnim, itd.

- Usunięcie trzeciego elementu listy (`del l[2]`). Powoduje to zmianę numeracji kolejnych elementów.
- Możemy dodać nowy element do listy komendą `append`.

Jednak jeżeli chcemy modyfikować elementy listy iterując po niej, to konieczne jest iterowanie po indeksach (a nie jak dotychczas po wartościach):

```
for i in range(len(l)):
    print(l[i])
    l[i] = 0
print(l)
```

```
15
17
8
[0, 0, 0]
```

Dzieje się tak gdyż przypisanie do zmiennej `x` jakiejś wartości w ramach konstrukcji `for x in lista`: modyfikuje tylko zmienną `x`, a nie element listy który został do niej pobrany.

Zauważ że użyliśmy `len()` aby uzyskać ilość elementów w liście i `range()` aby uzyskać w oparciu o nią listę wszystkich indeksów elementów listy `l`.

3 Tworzenie własnych funkcji

Często będziemy chcieli wielokrotnie wykorzystać raz napisany fragment kodu. W tym celu będziemy tworzyć własne *funkcje*.

Definicja funkcji ma następującą postać:

```
def <nazwa_funkcji>(<argumenty>):
    <pierwsze_polecenie>
    <drugie_polecenie>
    ...
```

Zwróć uwagę na podobieństwa definicji funkcji do pętli `for`:

- Na końcu pierwszej linijki jest dwukropek.
- Druga linijka musi być *wcięta*, tzn. rozpoczynać się od spacji, kilku spacji lub znaku tabulacji.
- Jeżeli w ramach funkcji chcemy wykonać kilka instrukcji muszą one mieć taki sam poziom wcięcia.
- „Wnętrze” funkcji kończymy wracając do takiego samego poziomu wcięcia na jakim ją rozpoczęliśmy (takiego wcięcia jakie miała linijka z słowem kluczowym `def`).

Jest to typowy sposób wyznaczania bloku kodu w Pythonie, z którym spotkaliśmy się już kilkakrotnie.

Gdy umieszczamy inną konstrukcję korzystającą z bloku kodu we wnętrzu jakiegoś innego bloku (np. pętlę `for` wewnątrz funkcji), blok tej instrukcji musi być „bardziej” wcięty od bloku w którym jest zawarty, powrót do poziomu wcięcia zewnętrznego bloku oznacza zakończenie bloku tej instrukcji i kontynuowanie zewnętrznego bloku.

Porada

Na funkcję można patrzeć jak na nazwany kawałek kodu, który możemy wywołać z innego miejsca ze odmiennymi wartościami zmiennych stanowiących jej argumenty.

Polecenie wywołania funkcji ma postać `nazwa_funkcji(argumenty)` i możemy napisać je w tym samym pliku, poniżej definicji tej funkcji. Typowo ilość i kolejność argumentów w definicji, jak i w wywołaniu powinny być takie same. Jeżeli nasza funkcja nie potrzebuje przyjmować argumentów nawiasy okrągłe w jej definicji i wywołaniu pozostawiamy puste. Jeżeli potrzebujemy więcej argumentów rozdzielamy je w obu przypadkach przecinkami (tak jak miało to miejsce w korzystaniu z funkcji `print`).

Przykład Napiszmy funkcję, która wypisuje swój argument podniesiony do kwadratu i wywołajmy ją:

```
def kwadrat(x):  
    print(x * x)  
  
kwadrat(7)  
kwadrat(2 + 3)
```

```
49  
25
```

Zwróć uwagę, iż wywołania funkcji w powyższym przykładzie nie są wcięte — są poza blokiem funkcji.

Funkcja też może zwrócić wybraną wartość używając komendy `return`. Wówczas można zwracaną wartość wykorzystać poza funkcją.

```
def kwadrat(x):  
    return(x * x)  
  
kwadrat(8)  
  
print(kwadrat(10))  
  
a = kwadrat(3)  
b = kwadrat(4)  
print(a+b)
```

```
100  
25
```

W tym przykładzie wartość zwrócona przez komendę `kwadrat(8)` nie jest ani wypisywana na ekran ani zapisywana. Żeby ją wypisać należy użyć komendy `print`, jak w przykładzie komendy `print(kwadrat(10))`. Zwracane wartości można też zapisywać jako zmienne i wykorzystać później w kodzie, np. dodać je do siebie jak w komendzie `print(a+b)`.

4 Typ logiczny

Jak już się przekonaliśmy można używać Pythona jako kalkulatora. Możemy go także użyć do obliczania wartości wyrażeń logicznych. Służy do tego wbudowany dwuwartościowy typ logiczny z wartościami:

- **True** oznaczającą logiczną jedynkę / prawdę
- **False** oznaczającą logiczne zero / fałsz

Operacje na tym typie wykonujemy z użyciem słów kluczowych: **and**, **or**, **not** oznaczających odpowiednio: iloczyn logiczny (aby był prawdą oba warunki muszą być spełnione), sumę logiczną (aby wynik był prawdą co najmniej jednej z warunków musi być spełniony) oraz negację logiczną. Podobnie jak w zwykłych operacjach arytmetycznych możemy grupować ich fragmenty (celem wymuszenia kolejności działań) przy pomocy nawiasów okrągłych.

Wartościom tego typu mogą odpowiadać wybrane wartości innych typów (np. liczba całkowita 0 odpowiada **False**, a pozostałe liczby całkowite **True**). Wartościami tego typu są też wyniki różnego rodzaju porównań, takich jak: `<` (mniejsze), `>` (większe), `<=` (mniejsze równe), `>=` (większe równe), `==` (równe), `!=` (nierówne).

4.1 Instrukcja warunkowa **if**

Często chcemy, aby program zachowywał się w różny sposób w zależności od tego, czy jakiś warunek jest spełniony, czy nie. W Pythonie (jak w większości języków programowania) służy do tego instrukcja warunkowa **if**.

Przypuśćmy, że chcemy napisać funkcję, która dla podanej wartości sprawdzi czy odpowiada ona logicznej prawdzie czy fałszowi i wypisuje odpowiedni komunikat. Zatem kod będzie wyglądał następująco:

```
def sprawdz(x):
    if x:
        print(x, '-- prawda')
    else:
        print(x, '-- nieprawda')
sprawdz(1)
sprawdz(0)
```

```
1 -- prawda
0 -- nieprawda
```

Zwróć uwagę na następujące rzeczy:

- **if** to po polsku „jeśli”, **else** to po polsku „w przeciwnym przypadku”.
- Linijki rozpoczynające się od **if** i **else** (podobnie jak linijki rozpoczynające się np. od **def**) kończą się dwukropkiem.
- „Wnętrze” **if**-a i **else**-a (linijki 3 i 5) jest wcięte (bardziej niż samo wnętrze definicji funkcji `sprawdz`).
- Linijka 3 zostanie wykonana, jeśli spełniony będzie warunek z liniiki 2, czyli jeśli wartość zmiennej `x` będzie odpowiadała prawdzie.
- Linijka 5 zostanie wykonana, jeśli warunek z liniiki 2 nie będzie spełniony.

W powyższym przykładzie użyliśmy konstrukcji **if/else** do rozróżnienia pomiędzy dwoma przypadkami. Używając komendy **elif** (skrót od **else if**) możemy stworzyć bardziej skomplikowany kod do rozróżnienia pomiędzy kilkoma różnymi przypadkami:

```
for x in range(0, 5):
    if x < 1 or x == 4:
        print('mniejsze od 1 lub równe 4')
    elif x in [0,2,3]:
        print('0 2 lub 3')
    else:
        print('nic ciekawego')
```

```
mniejsze od 1 lub równe 4
nic ciekawego
0 2 lub 3
0 2 lub 3
mniejsze od 1 lub równe 4
```

Ten kod składa się z trzech bloków, które są wykonywane w zależności od spełnienia poszczególnych warunków: **if**, **elif**, **else**. Mamy dużą dowolność w konstruowaniu tego typu fragmentów kodu: bloków **elif** może być dowolnie wiele, blok **else** może występować jako ostatni blok, ale może też go nie być w ogóle.

W powyższym przykładzie widzimy również, że w roli warunków sprawdzanych w ramach **ifa** mogą występować bardziej złożone wyrażenia. Możemy tutaj użyć dowolnego wyrażenia którego wynik odpowiada wartości logicznej **True/False**, najczęściej spotkamy się z wyrażeniami złożonymi z poznanych już operatorów porównań (<, >, <=, >=, ==, !=) i operacji logicznych (**and**, **or**, **not**).

Zwróć uwagę na warunek postaci „A **in** B”. Taki warunek sprawdza, czy wartość reprezentowana przez A jest elementem B, a jego wynik oczywiście także jest wartością logiczną. W naszym przykładzie sprawdzaliśmy, czy wartość zmiennej `x` występuje w podanej liście liczb, czyli czy jest 1, 2 lub 3.

Zauważ, że dla `x` wynoszącego 0 spełnione są dwa warunki (pierwszy i środkowy), w takim wypadku decydująca jest kolejność warunków i w konstrukcji **if/elif** wykonany zostanie jedynie kod związany z pierwszym pasującym warunkiem.

Zadanie 4.1.1

Sprawdź, czy punkty o współrzędnych równych: $p_x^1 = 10$ $p_y^1 = 2$, $p_x^2 = 10$ $p_y^2 = 5$, $p_x^3 = 8$ $p_y^3 = 8$, są wewnątrz okręgu, który ma środek w $o_x = 5$ $o_y = 5$ i promień $r=5$. Jeśli punkt jest wewnątrz wypisz "Wewnątrz", jeżeli jest na okręgu to wypisz "Na okręgu", a w przeciwnym przypadku "Na zewnątrz".

4.2 Pętla `while`

Do tej pory korzystaliśmy z pętli `for`, która pozwala na iterowanie po liście elementów. Innym istotnym rodzajem pętli jest pętla `while`, która powoduje wykonywanie zawartego w niej kodu dopóki podany warunek jest spełniony.

```
a, b = 0, 1
while a <= 40:
    print(a, end=" ")
    a, b = b, a + b
```

```
0 1 1 2 3 5 8 13 21 34
```

Zwróć uwagę, że wewnątrz pętli `while` (tak samo jak innych konstrukcji używających wciętego bloku - takich jak `for`, czy `if`) może znajdować się więcej niż jedno polecenie. Trzeba tylko pamiętać, aby wszystkie były poprzedzone takim samym wcięciem.

Pętla `while` jest też naturalnym wyborem gdy w Pythonie chcemy przechodzić przez jakiś zakres liczb z krokiem nie całkowitym (wcześniej poznana instrukcja `range`, stosowana do iterowania po zakresie liczbowym w pętli `for`, wymaga aby krok był całkowity).

5 Generowanie liczb losowych

Do generowania liczb losowych w Pythonie posłużyć może np. pakiet `random`. Aby go używać, na początku programu musimy go zaimportować poleceniem:

```
import random
```

Pakiet ten zawiera wiele komend umożliwiających generowanie liczb losowych. Na naszych zajęciach będziemy korzystać z następujących.

```
random.random()           # generuje liczbę z przedziału [0,1]
random.uniform(-3,2)       # generuje liczbę z przedziału [-3,2]
random.randint(-3,2)       # generuje liczbę całkowitą z przedziału [-3,2]
random.choice(["orzeł", "reszka"]) # generuje losową wartość z listy
```

Zadanie 5.0.1

Wypisz na konsoli 10 losowo wygenerowanych liczb całkowitych z przedziału od 1 do 6.

Zadanie 5.0.2

Napisz program, który z prawdopodobieństwem 23% wypisze "wygrana". W przeciwnym wypadku powinna ona wypisać "przegrana".

Wskazówka: Możesz tu wykorzystać funkcję `random.random()` generuje losową wartość z przedziału $[0,1]$. Na przykład, jakie jest prawdopodobieństwo, że warunek `random.random() < 0.36` zostanie spełniony?

6 Rysowanie wykresów

Do rysowania wykresów w Pythonie posłużyć może np. pakiet `pyplot`. Aby go używać, na początku programu musimy go zaimportować i wybrać odpowiedni tryb poleceniami:

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

Do tworzenia wykresów będzie nam służyła funkcja `plot()` z biblioteki zaimportowanej jako `plt`. Będziemy ją wywoływać w sposób następujący: `plt.plot(x, y, format)`, gdzie `x` jest listą kolejnych wartości osi X, `y` jest listą odpowiadających im wartości osi Y, a format określa typ rysowanego wykresu. Na przykład:

```
plt.plot([1, 2, 3, 4], [1, 2, 4, -2])
```

Python pozwala nam również ustawić własne nazwy osi pionowej i poziomej, na przykład:

```
plt.xlabel("oś pozioma")  
plt.ylabel("oś pionowa")
```

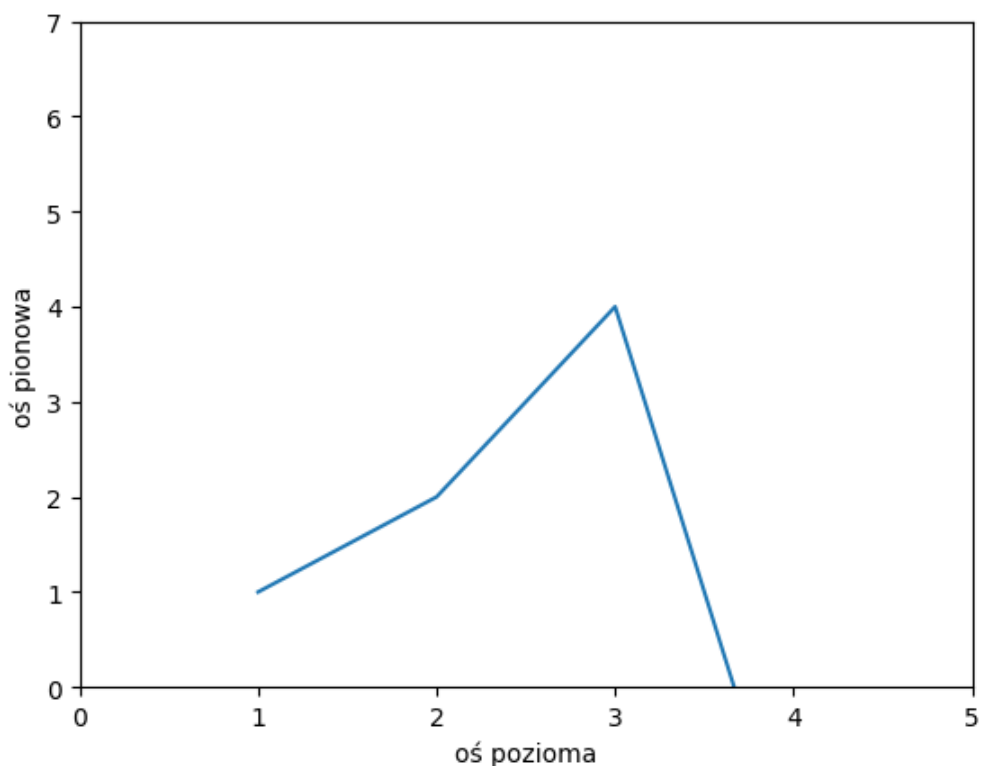
Możemy też ustawić zakres wartości na osiach za pomocą komend:

```
plt.xlim([0,5])  
plt.ylim([0,7])
```

Wykres możemy wyświetlić za pomocą komendy `plt.show()`:

```
plt.show()
```

Po wykonaniu powyższych komend powinieneś dostać następujący wykres:



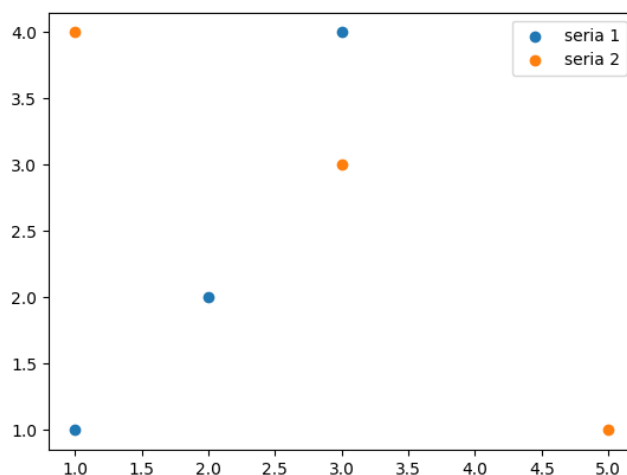
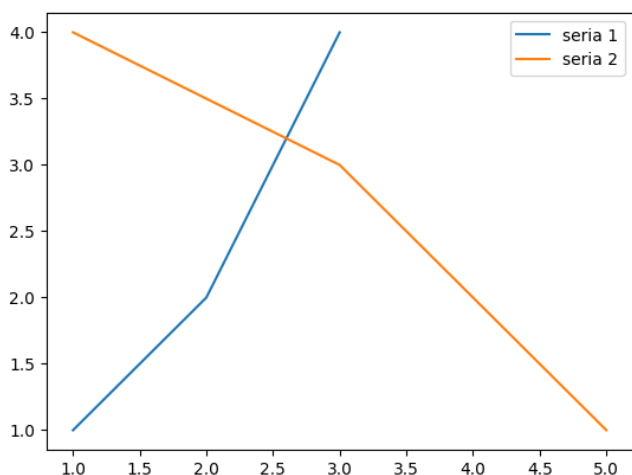
Biblioteka `pyplot` pozwala nam narysować więcej niż jedną serię danych. W tym celu wystarczy dodać więcej instrukcji `plot()` przed wyświetleniem wykresu. Warto dla czytelności dodać legendę komendą `legend`, będzie ona używać etykiety wykresu przekazanej w argumencie `label` funkcji `plot`:

```
plt.plot([1, 2, 3], [1, 2, 4], label='seria 1')  
plt.plot([1, 3, 5], [4, 3, 1], label='seria 2')  
  
plt.legend()  
plt.show()
```

Zmieniając komendę `plt.plot` na `plt.scatter` zostanie narysowany wykres punktowy, zamiast liniowego.

```
plt.scatter([1, 2, 3], [1, 2, 4], label='seria 1')
plt.scatter([1, 3, 5], [4, 3, 1], label='seria 2')

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



Zadanie 6.0.1

Narysuj wykres przedstawiający pierwsze pięć potęg dwójki.

Jeśli nie chcemy dostać wykres punktowy, na którym punkty nie są łączone linią wystarczy zastąpić komendą `plt.plot`, komendą `plt.scatter`. Na przykład:

```
plt.scatter([1, 2, 3, 4], [1, 2, 4, -2])
```

Zadanie 6.0.2

Narysuj wykres x - y , na którym zostanie umieszczone 20 losowo wygenerowanych punktów w obrębie kwadratu o wierzchołkach w punktach $(-1, -1)$, $(1, -1)$, $(1, 1)$ i $(-1, 1)$.

Zadanie 6.0.3

Napisz funkcję, przyjmującą dwa argumenty a i b , która używając pętli **while**, obliczy i zwróci sumę liczb całkowitych większych od a i mniejszych od b .

7 Zadania dodatkowe

Zadanie 7.0.1

Oblicz sumę $1^2 + 2^2 + 3^2 + \dots + 99^2 + 100^2$.

Zadanie 7.0.2

Używając dwóch pętli **for**, jedna wewnątrz drugiej, napisz program, który wypisze na ekranie *trójkąt z siódemek*, taki jak poniżej:

7
77
777
7777
77777
777777
7777777
77777777
777777777

Zadanie 7.0.3 (symulacja gry w wojnę)

1. Napisz program symulujący karcianą grę w wojnę (zasady są opisane poniżej).
2. Narysuj wykres, na którym będzie przedstawiona zmiana liczby kart każdego z graczy w kolejnych rundach.
3. Zbadaj czy po wygraniu rundy bardziej opłaca się swoją (zwycięską) kartę umieszczać jako pierwszą czy jako drugą na spodzie swojej talii.

Zasady gry. Dwóch graczy dzieli po równo talię 52 kart. Następnie odkrywają wierzchnią talię swojej talii. Gracz z wyższą kartą wygrywa odkrytą kartę przeciwnika, i kładzie obie karty na spodzie swojej talii. Jeśli obie odkryte karty są równo następuje "wojna". Wówczas każdy gracz kładzie na stole następną kartę ze swojej talii (zakrytą) i odkrywa kolejną kartę. Gracz z wyższą kartą wygrywa wszystkie karty na stole. Jeśli odkryte karty są ponownie równej wartości, "wojna" jest powtarzana. Gra kończy się kiedy jeden z graczy nie ma więcej kart do odkrycia.

8 Praca domowa nr 1

Rozwiązania zadań domowych należy przesłać do czasu następnych ćwiczeń. Prowadzący może również zmienić ostateczny termin przesłania pracy domowej, np. na inną godzinę lub dzień.

Za pracę domową można maksymalnie dostać 6 punktów. Można wybierać zarówno łatwiejsze zadania, jak i trudniejsze, za większą liczbę punktów. Można również rozwiązać zadania za większą liczbę punktów, np. za 10. Jeśli wtedy poprawnie będą rozwiązane zadania za 5 punktów, to zostanie przydzielone 5 punktów. Jeśli będą poprawnie rozwiązane zadania za więcej niż 6 punktów, np. 8 czy 10, to taka osoba otrzyma maksymalnie 6 punktów.

Pamiętaj, żeby notebook był czytelny – każde zadanie powinno być umieszczone w osobnym bloku tekstowym oraz poprzedzone numerem (i opcjonalnie opisem) zadania. Komentarze są mile widziane.

W tym celu wysłania pracy domowej kliknij "Udostępnij" znajdujące się w prawym górnym rogu okna. Następnie w sekcji "Pobierz link" zmień domyślną opcję "Przeglądający" na "Komentujący", a następnie skopiuj podany w oknie link i wyślij do prowadzącemu zajęcia przez Discorda.

Zadanie 8.0.1 — 1 pkt

Napisz pętlę, która wypisze wszystkie dwucyfrowe liczby podzielne przez 7. Kolejne liczby powinny być wypisane w jednym wierszu i porozielane pojedynczymi spacjami.

Zadanie 8.0.2 — 1 pkt

W grze „Lotto” gracze obstawiają 6 liczb z przedziału od 1 do 49. Następnie losowane jest sześć losowych liczb z tego przedziału. Im więcej wylosowanych liczb gracz zgadł tym wyższa jest jego wygrana.

Napisz pętlę, która obliczy prawdopodobieństwo trafienia wszystkich 6 wylosowanych liczb.

Zadanie 8.0.3 — 1 pkt

Ciąg Fibonacciego to ciąg, w którym dwa pierwsze wyrazy to 0 i 1, a każdy kolejny wyraz jest sumą dwóch poprzednich:

0, 1, 1, 2, 3, 5, 8, ...

Napisz program, który znajduje i wypisuje 20 pierwszych liczb ciągu Fibonacciego. Liczby powinny być wypisane po jednej w każdym wierszu.

Zadanie 8.0.4 — 2 pkt

Napisz pętlę, która wypisze liczby od 1 do 10, każdą w oddzielnej linii. Jeśli jest ona podzielna przez 2, niech wypisze ją 2 razy rozdzielając liczby przecinkiem np. 4, 4. Jeśli jest ona podzielna przez 3, niech wypisze ją 3 razy, rozdzielane przecinkami. Jeśli jest ona podzielna naraz przez 2 i 3, to niech wypisze ją jeden raz.

Zadanie 8.0.5 — 2 pkt

Wypisz wszystkie liczby pierwsze, które są mniejsze niż 1000, każdą w oddzielnej linii.

Zadanie 8.0.6 — 1 pkt (test parzystości)

Napisz funkcję `parzystosc(x)`, który dla podanej liczby x wypisze:

- liczba parzysta, jeśli podana liczba x jest całkowita i parzysta,
- liczba nieparzysta, jeśli podana liczba x jest całkowita i nieparzysta,
- liczba nie jest całkowita, jeśli podana liczba x nie jest liczbą całkowitą.

Zadanie 8.0.7 — 1 pkt (rzut kością)

Rozważ sześcienną kostkę, na której trzech ściankach są litery "a", na dwóch ściankach litery "b", a na jednej ściance litera "c". Napisz program, który zasymuluje 20 rzutów tą kością i wypisze ich wyniki na ekranie.

Zwróć uwagę, że litera "a" powinna statystycznie pojawiać się częściej niż b i c.

Zadanie 8.0.8 — 1 pkt (silnia)

Napisz funkcję `silnia(n)`, która dla podanej liczby całkowitej n zwróci iloczyn wszystkich liczb naturalnych od 1 do n . Na przykład `silnia(5)` powinna zwrócić $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$.

Zadanie 8.0.9 — 2 pkt (średnia z losowania)

Napisz funkcję `losuj(n)`, która wygeneruje n losowych liczb z przedziału od 0 do 10, i obliczy z nich średnią arytmetyczną.

Zadanie 8.0.10 — 2 pkt (losowe punkty w kole)

Narysuj na wykresie x - y sto losowo wygenerowanych punktów w obrębie koła o środku w punkcie $(0, 0)$ i promieniu 1.

Wskazówka: Współrzędne x i y dla punktów w obrębie tego koła spełniają warunek $x^2 + y^2 \leq 1$.

Zadanie 8.0.11 — 2 pkt (wykres funkcji kwadratowej)

Za pomocą skryptu w Pythonie narysuj wykres funkcji

$$y(x) = x^2 - 2x + 1$$

w zakresie od $x = -3$ do $x = 3$. Pamiętaj o tym, aby odpowiednio nazwać osie tego wykresu.

Wskazówka: Utwórz listy x i y , w których będą zapisane wartości x i y , przez które wykres funkcji będzie przechodzić. Możesz sam zdecydować ile punktów zostanie naniesione na wykres.