Laboratorium programistyczne 1 - wprowadzenie do Pythona

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

2022-03-31

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 podłączenie 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 informacja mi temat korzystania z Colab (patrz Rysunek 1).

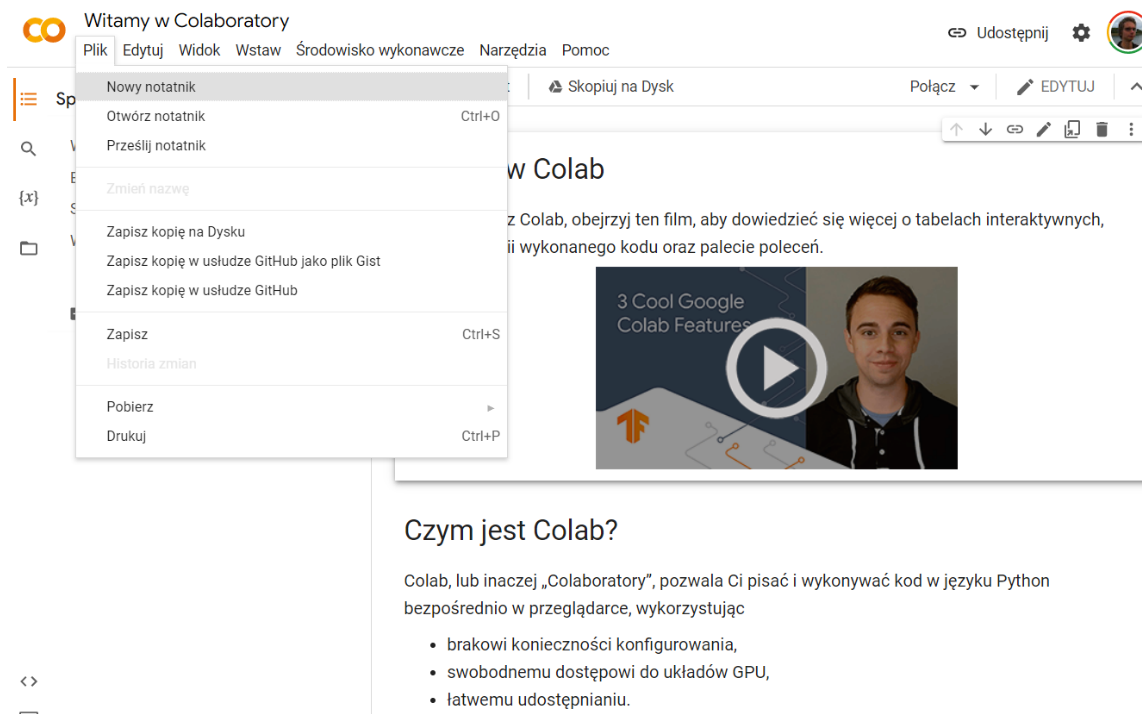
Polecam zajrzeć do niego w domu, jednak teraz czas utworzyć nasz pierwszy notatnik. W tym celu w Plik → Nowy notatnik. W lewym górnym rogu ekranu znajduje się nazwa twojego notatnika. Możesz ją zmienić klikając na nazwie domyślnej i wpisując własną. Sugeruję, w nazwie pliku podać swoje imię i nazwisko,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

W tak utworzonym notatniku możecie dodawać pola tekstowe klikając ”+test” w górnym lewym roku 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:



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

```
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 linijkach 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.

Zadanie 1.2.2

Spróbuj przy pomocy wcześniej poznanych operatorów zaokrąglić w dół następujące liczby: `10.7`, `1.1234`, `3.5`.

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).

Ilekroć 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 którym 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`.

2.2.1 Wybór podlisty

Innymi przydatnymi operacjami na listach jest wycinanie podlisty przy użyciu dwukropka:

```
l = [0, 1, 2, 3, 4, 5, 6]
print(l[2:5], l[3:], l[:3])
```

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

W powyższym przykładzie:

- komenda `l[2:5]` zwraca podlistę od elementu nr 2 (**włącznie**) do elementu nr 5 (**wyłącznie**) listy `l`,
- komenda `l[3:]` zwraca podlistę od elementu nr 3 (**włącznie**) do końca,
- komenda `l[:3]` zwraca podlistę od początku do elementu nr 3 (**wyłącznie**).

Podobnie jak w `range()` możemy podać trzeci argument określający krok. Pozwala to na wybieranie co n-tego elementu z listy, zarówno zaczynając od początku jak i końca:

```
l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(l[::2], l[1::2])
print(l[::-1])
print(l[::-3])
print(l[::-1][::3], l[::3][::-1])
```

```
[0, 2, 4, 6, 8] [1, 3, 5, 7, 9]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
[9, 6, 3, 0]
[9, 6, 3, 0] [9, 6, 3, 0]
```

W powyższym przykładzie:

- komenda `l[::2]` zwraca co drugi element listy `l`,
- komenda `l[1::2]` zwraca co drugi element od elementu nr 1,
- komenda `l[::-1]` zwraca listę od tyłu,
- komenda `l[::-3]` zwraca co 3 element z listy od tyłu (warto zauważyć że nie zawsze jest to równoważne wypisaniu listy złożonej z co 3 elementu od tyłu).

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 kilkukrotnie.

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.

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, '-- nie prawda')
sprawdz(1)
sprawdz(0)
```

```
1 -- prawda
0 -- nie prawda
```

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 linijki 2, czyli jeśli wartość zmiennej `x` będzie odpowiadała prawdzie.
- Linijka 5 zostanie wykonana, jeśli warunek z linijki 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 `if`a 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).

Zadanie 4.2.1

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.

5 Zadania dodatkowe

Zadanie 5.0.1

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

Zadanie 5.0.2

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
```

6 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.

licealisci.pracownia@icm.edu.pl wpisując jako temat wiadomości `Lx PD1`, gdzie `x` to numer grupy, np. `L3 PD1` dla grupy L3, itd.

Zadanie 6.0.1 — 1 pkt

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

Zadanie 6.0.2 — 1 pkt

Napisz pętlę, która wypisze 20 elementów ciągu geometrycznego, który zaczyna się od 65536 i ma iloczyn -1/2. Kolejne liczby powinny być wypisane w jednym wierszu i porozdzielane pojedynczymi przecinkami.

Kilka pierwszych elementów tego ciągu: 65536, -32768, 16384, -8192, 4096 itd.

Zadanie 6.0.3 — 1 pkt

Wypisz 20 pierwszych liczb ciągu Fibonacciego. Liczby powinny być wypisane po jednej w każdym wierszu.

Wskazówka: jeśli macie zajęcia przed wykładem, to na wykładzie poznacie ciąg Fibonacciego, czyli ciąg 0, 1, 1, 2, 3, 5, 8, ..., w którym każdy kolejny wyraz jest sumą dwóch poprzednich.

Zadanie 6.0.4 — 1 pkt

Napisz pętlę, która obliczy liczbę wszystkich możliwych kombinacji „6” w grze liczbowej „Lotto”. Dla przypomnienia losujemy tam zbiór 6 liczb ze zbioru 49 liczb.

Zadanie 6.0.5 — 1 pkt

Napisz pętlę, która wypisze liczby od 1 do 10, każdą w oddzielnej linijce. 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 6.0.6 — 2 pkt

Oblicz szansę trafienia „3” w grze liczbowej „Lotto”. Dla przypomnienia losujemy tam zbiór 6 liczb ze zbioru 49 liczb.

Zadanie 6.0.7— 2 pkt

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

Zadanie 6.0.8 — 3 pkt

Używając dwóch pętli `for` (jedna wewnątrz drugiej), napisz program, który dla wypisze trójkąt Pascala do wybranej liczby linijek `n`.
Dla n = 6 wynik będzie następujący:

```
1,
1, 1,
1, 2, 1
1, 3, 3, 1
1, 4, 6, 4, 1
1, 5, 10, 10, 5, 1
```