Powtórzenie wiadomości z Pythona

Cz. I – łańcuchy znaków i listy

Podstawy interaktywnej pracy w Pythonie

Kliknij ikonę IDLE (Python GUI). W oknie pojawi się zapis podobny do poniższego:

```
Python 3.11.4 (tags/v3.11.4:d2340ef, Jun  7 2023, 05:45:37) [MSC v.1934 64
bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

Linia poleceń to trzy znaki „>”:

```
>>>
```

Nasza praca będzie polegała na wpisywaniu poleceń i zatwierdzaniu klawiszem *Enter*, aby wyświetlić wynik na ekranie. Zaczniemy od zapoznania się z poleceniem „*print*”. Polecenie „*print*” mówi interpreterowi, aby wypisał wskazany tekst:

```
>>> print("umie liczyc")
umie liczyc
```

Zadeklarujemy teraz zmienną „*tekst*” i przypiszemy jej wartość w postaci ciągu znaków „*umie liczyc*”, a następnie wywołamy jej wartość poleceniem „*print*”:

```
>>> tekst = "umie liczyc"
>>> print(tekst)
umie liczyc
```

Następnie zadeklarujemy zmienną „*suma*” i przypiszemy jej wynik dodania liczb *4* i *5* oraz wywołamy ją poleceniem „*print*”:

```
>>> suma = 4 + 5
>>> print(suma)
9
```

Ten sam wynik uzyskamy też wpisując nazwę samej zmiennej:

```
>>> suma
9
```

Możemy teraz wypisać tekst „*Suma 4 i 5 to*” wraz z wartością zmiennej „*suma*” za pomocą następującego kodu:

```
>>> print("Suma 4 i 5 to: ", suma)
Suma 4 i 5 to:  9
```

Łańcuchy, liczby i zmienne

W języku Python teksty będziemy nazywać łańcuchami (ang. *string*). Łańcuchy powinny być zawarte między pojedynczymi lub podwójnymi cudzysłowami. Posiadanie dwóch typów cudzysłowów ułatwia tworzenie łańcuchów zawierających te znaki:

```
>>> print(' "Kasia", krzyknal Tomek, "Chodz szybko!" ')
"Kasia", krzyknal Tomek, "Chodz szybko!"
>>> print(" 'Kasia', krzyknal Tomek, 'Chodz szybko!' ")
'Kasia', krzyknal Tomek, 'Chodz szybko!'
```

Jeżeli chcemy zaznaczyć początek nowej linii lub wstawić znak tabulacji korzystamy z ukośnika odwrotnego (ang. *backslash*), po którym następuje odpowiednio *n* lub *t*:

```
>>> print("Tu jest nowa linia,\na tu \ttabulator")
Tu jest nowa linia,
a tu  tabulator
```

Jeśli przed cudzysłowem umieścimy „*r*" znaki nowej linii i tabulacji nie będą interpretowane:

```
>>> print(r"Tu jest nowa linia,\na tu \ttabulator")
Tu jest nowa linia,\na tu \ttabulator
```

Przy pomocy funkcji „*len*" możemy policzyć, ile jest znaków w łańcuchu:

```
>>> len(tekst)
11
```

Natomiast za pomocą metody „*count*" zliczamy ilość wystąpień znaków lub ciągów znaków:

```
>>> tekst.count("u")
1
>>> tekst.count("i")
2
>>> tekst.count("mie")
1
>>> tekst.count("czyc")
1
```

Ostatnia metoda staje się przydatna do zliczania poszczególnych zasad w sekwencji DNA lub RNA oraz wyszukiwania powtarzających się sekwencji nukleotydów:

```
>>> dna = "CACCCAGAAGTATATATTTTGATTCTACCAGGATTTGGTATAATT"
>>> dna.count("T")
17
>>> dna.count("A")
14
>>> dna.count("TTTT")
1
```

Podobnie możemy operować na sekwencjach aminokwasowych i zliczać poszczególne aminokwasy w takiej sekwencji:

```
>>> bialko = "GMGSYQLLVPPPEALSKPLSVPTRLLLGPGPSNLAPRVLAAGSLRMIGHMQKEMLQIMEE"
```

```
>>> bialko.count("P")
8
>>> bialko.count("N")
1
>>> bialko.count("L")
11
```

Z kolei metoda „*replace*" pozwala na zamianę znaków w łańcuchu:

```
>>> rna = dna.replace("T","U")
>>> rna
'CACCCAGAAGUAUAUAUUUUGAUUCUACCAGGAUUUGGUAUAAUU'
```

W ten sposób możemy przepisać DNA na RNA. Ponadto, każdy ciąg znaków w Pythonie stanowi tablicę (jednowymiarową), co oznacza, że można odwoływać się do poszczególnych jej elementów za pomocą indeksu, rozpoczynając od *0*:

```
>>> dna[0]
'C'
>>> rna[1]
'A'
```

Wstawienie znaku „-" przed indeksem spowoduje zwrócenie kolejnego znaku od końca tablicy:

```
>>> rna[-1]
'U'
>>> rna[-2]
'U'
>>> rna[-3]
'A'
```

Natomiast liczby zapisujemy w sposób następujący:

```
>>> liczba = 100
>>> liczba
100
>>> liczba2 = 1000.5
>>> liczba2
1000.5
>>> liczba3 = -50
>>> liczba3
-50
>>> liczba4 = 45.1E-2
>>> liczba4
0.451
```

Ostatni zapis odnosi się do notacji naukowej i oznacza 45.1 x 10^{-2}. Zmienne w Pythonie pozwalają na tymczasowe przechowywanie łańcuchów, liczb i innych wartości. Nazwy zmiennych są czułe na wielkość liter:

```
>>> x = 5
>>> x
5
>>> X = 6
>>> X
6
>>> x
```

```
5
```

Do przypisywania wartości zmiennym, jak już zauważyliśmy, służy operator przypisania, czyli znak „=” :

```
>>> y = 13
```

Przypisanie działa od strony prawej do lewej. Powyżej, zmiennej *y* przypisano liczbę *13*. Od momentu przypisania, zmienną można stosować zamiast liczby:

```
>>> print("Wartosc zmiennej y to ", y)
Wartosc zmiennej y to  13
```

Ta sama zmienna może zostać użyta wielokrotnie. Operator przypisania może służyć do zmiany jej wartości:

```
>>> print("Wartosc zmiennej y to nadal", y)
Wartosc zmiennej y to nadal 13
>>> y = 150
>>> print("Ale teraz wartosc zmiennej y to", y)
Ale teraz wartosc zmiennej y to 150
```

Zadania (A):

Dla sekwencji DNA: GTATTATCAATAGGAGCAGTATTTGCAATCATAGCCGGGTT-CGTACAGTGATTCCCCTTATTCACGGGAC wykonaj następujące zadania:

1. Przypisz powyższy łańcuch zmiennej o nazwie *dna.*
2. Oblicz długość tej sekwencji.
3. Oblicz, ile razy występuje każda z zasad.
4. Oblicz, ile razy występuje sekwencja „CC”, „GTA” oraz „CAG”.
5. Zamień sekwencję DNA na mRNA, (zamiana tyminy na uracyl) i przypisz ją nowej zmiennej „*mrna*”.
6. Sprawdź, ile kodonów alaniny (GCU, GCC, GCA, GCG) występuje w tej sekwencji.
7. Sprawdź, ile kodonów proliny (CCU, CCC, CCA, CCG) występuje w tej sekwencji.

Obliczenia arytmetyczne

Python potrafi wykonywać podstawowe operacje arytmetyczne. Odpowiedzialne są za nie symbole zwane operatorami: + (dodawanie), - (odejmowanie), * (mnożenie), / (dzielenie), ** (potęgowanie).

Poniższe przykłady przedstawiają wykonywanie obliczeń i wypisywanie ich wyniku:

```
>>> 2 + 2
4
>>> v = 20
>>> w = 2 * 3
>>> v * w
120
>>> a, b, c = 3, 56, 100
>>> a, b, c
( 3 , 56, 100)
>>> a
3
```

```
>>> b
56
>>> 7 / 3
2.3333333333333335
>>> a = b = c = 5
>>> a, b, c
(5, 5, 5)


>>> z = x * X + y
>>> z
180

>>> a = 3
>>> b = 5
>>> c = 2
>>> d = b ** c * a
>>> d
75
```

Zauważmy, że Python kieruje się naturalnymi priorytetami operatorów (kolejność wykonywania działań arytmetycznych). Chcąc zmienić kolejność stosujemy nawiasy:

```
>>> e = b ** (c * a)
>>> e
15625
```

Zadania (B):

Przypisz zmiennym *e*, *f*, *g* wartości odpowiednio: *6*, *7* i *10*, a następnie oblicz:

1. Pole powierzchni kwadratu o boku długości *e*,
2. Pole powierzchni prostokąta o bokach długości *e* i *f*,
3. Obwód i pole powierzchni koła o promieniu *g*,
4. Iloczyn *e* i *f*,
5. Iloraz *f* i *g*,
6. Różnicę *g* i *e*.

Listy

Listy są jednym z najważniejszych typów danych. Jest to uporządkowany zbiór stałych lub zmiennych ograniczonych nawiasami kwadratowymi. Przykłady list przypisywanych określonym zmiennym:

```
>>> lista1 = ["jeden", "dwa", "trzy"]
>>> lista1
['jeden', 'dwa', 'trzy']
>>> lista2 = [a, b, c, d]
>>> lista2
[3, 5, 2, 75]
>>> lista3 = ["cztery", "piec", a, c]
>>> lista3
['cztery', 'piec', 3, 2]
```

Lista może być używana tak jak tablica zaczynająca się od *0*. Pierwszym elementem niepustej listy o nazwie „*lista1*” jest zawsze *lista1[0]*:

```
>>> lista1[0]
'jeden'
```

Ostatnim elementem trzyelementowej listy jest *lista1[2]*, ponieważ indeksy są liczone zawsze od *0*.

```
>>> lista1[2]
'trzy'
```

Podobnie jak w przypadku łańcuchów, za pomocą ujemnych indeksów odnosimy się do elementów idących od końca do początku:

```
>>> lista3[-1]
2
>>> lista3[-2]
3
>>> lista3[-3]
'piec'
```

Można też pobrać podzbiór listy, który jest nazywany "wycinkiem" (ang. *slice*), poprzez określenie dwóch indeksów. Zwracaną wartością jest nowa lista zawierająca wszystkie elementy z listy rozpoczynające się od pierwszego wskazywanego indeksu i, idąc w górę, kończy na drugim wskazywanym indeksie, nie dołączając go. Kolejność elementów względem wcześniejszej listy jest także zachowana.

```
>>> lista4 = lista3[0:2]
>>> lista4
['cztery', 'piec']
```

Należy pamiętać, że listy są indeksowane od zera tzn. w tym przypadku *lista3[0:2]* zwraca pierwsze dwa elementy listy, rozpoczynając od *lista3[0]*, a kończąc na *lista3[1]*, ale nie dołączając *lista3[2]*.

W wycinaniu można stosować też skróty. Jeśli lewy indeks wynosi *0*, możemy go opuścić, wartość *0* jest domyślna:

```
>>> lista4 = lista3[:2]
>>> lista4
['cztery', 'piec']
```

Podobnie, jeśli prawy indeks jest długością listy, możemy go pominąć:

```
>>> lista4 = lista3[1:]
>>> lista4
['piec', 3, 2]
```

Jeśli obydwa indeksy zostaną pominięte, wszystkie elementy zostaną dołączone:

```
>>> lista4 = lista3[:]
>>> lista4
['cztery', 'piec', 3, 2]
```

Do listy możemy dodawać elementy za pomocą metody „*append*”. W ten sposób dodajemy pojedynczy element do końca listy:

```
>>> lista2.append(100)
>>> lista2
[3, 5, 2, 75, 100]
```

Za pomocą metody „*insert*” wstawiamy pojedynczy element do listy. Numeryczny argument jest indeksem, pod którym ma się znaleźć wstawiana wartość; reszta elementów, która znajdowała się pod tym indeksem lub miała większy indeks, zostanie przesunięta o jeden indeks dalej:

```
>>> lista2.insert(3, 100)
>>> lista2
[3, 5, 2, 100, 75, 100]
```

Zauważmy, że elementy w liście nie muszą być unikalne i mogą się powtarzać. Za pomocą metody „*extend*” łączymy listę z inną listą. Nie możemy wywołać „*extend*” z wieloma argumentami, trzeba ją wywoływać z pojedynczym argumentem - listą. W tym przypadku ta lista ma pięć elementów:

```
>>> lista2.extend([200, 300, 400, 500, 600])
>>> lista2
[3, 5, 2, 100, 75, 100, 200, 300, 400, 500, 600]
```

Listy możemy też przeszukiwać za pomocą metody „*index*”:

```
>>> lista2.index(100)
3
>>> lista2.index(200)
6
```

W powyższym przykładzie metoda „*index*” zwróciła indeks pierwszego wystąpienia wartości *100* i *200* z listy „*lista2*”. Za pomocą metody „*remove*” usuwamy pierwszy występujący element listy:

```
>>> lista2.remove(100)
>>> lista2
[3, 5, 2, 75, 100, 200, 300, 400, 500, 600]
```

Metoda „*remove*” usuwa tylko pierwszy występujący element. W naszym przypadku *100* wystąpiło dwa razy, ale usunięte zostało tylko pierwsze wystąpienie. Na listach można używać także operatorów. Np. aby połączyć dwie listy korzystamy z operatora „+”. Ten sam operator może być wykorzystany do połączenia większej liczby list:

```
>>> lista4 = lista1 + lista2 + lista3
>>> lista4
['jeden', 'dwa', 'trzy', 3, 5, 2, 75, 100, 200, 300, 400, 500, 600, 'cztery',
'piec', 3, 2]
```

Za pomocą plusa można także dodać nowe elementy do istniejącej listy:

```
>>> lista5 = lista4 + [700, 800]
>>> lista5
['jeden', 'dwa', 'trzy', 3, 5, 2, 75, 100, 200, 300, 400, 500, 600, 'cztery',
'piec', 3, 2, 700, 800].
```

Python posiada także operator „+=”. Operacja *lista5 += ['koniec']* jest równoważna operacji *lista5.extend(['koniec'])*:

```
>>> lista5 += ['koniec']
>>> lista5
['jeden', 'dwa', 'trzy', 3, 5, 2, 75, 100, 200, 300, 400, 500, 600, 'cztery',
'piec', 3, 2, 700, 800, 'koniec']
```

Wreszcie operator „*” zwielokrotnia elementy listy:

```
>>> lista6 = lista5 * 3
>>> lista6
['jeden', 'dwa', 'trzy', 3, 5, 2, 75, 100, 200, 300, 400, 500, 600, 'cztery',
'piec', 3, 2, 700, 800, 'koniec', 'jeden', 'dwa', 'trzy', 3, 5, 2, 75, 100,
200, 300, 400, 500, 600, 'cztery', 'piec', 3, 2, 700, 800, 'koniec', 'jeden',
'dwa', 'trzy', 3, 5, 2, 75, 100, 200, 300, 400, 500, 600, 'cztery', 'piec',
3, 2, 700, 800, 'koniec']
```

Jeżeli chcemy zamienić łańcuch znaków na listę, korzystamy z funkcji „*list*”:

```
>>> lancuch = "ABCDEFGHIJKLMNOPRSTUVWXYZ"
>>> lista = list(lancuch)
>>> lista
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
'P', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
```

Istnieje możliwość określenia długości listy za pomocą znanej już nam funkcji „*len*”, a także zliczenia wystąpienia kolejnych wartości (*count*), odwrócenia wartości listy (*reverse*) i ich posortowania (*sort*):

```
>>> len(lista)
25
>>> lista.count("E")
1
>>> lista.reverse()
>>> lista
['Z', 'Y', 'X', 'W', 'V', 'U', 'T', 'S', 'R', 'P', 'O', 'N', 'M', 'L', 'K',
'J', 'I', 'H', 'G', 'F', 'E', 'D', 'C', 'B', 'A']
>>> lista.sort()
>>> lista
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
'P', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
```

Oprócz list mamy tez tzw. krotki (ang. *tuple*), które różnią się od zwykłych list tym, że nie możemy ich zmieniać po ich utworzeniu. Krotki definiuje się tak samo jak listy, tylko że zamiast nawiasów kwadratowych stosujemy nawias okrągły:

```
>>> krotka = ("a", "b", "c", "d")
>>> krotka
('a', 'b', 'c', 'd')
>>> krotka[1]
'b'
>>> krotka[2]
'c'
>>> krotka[1:3]
('b', 'c')
```

W przypadku krotek, lista dostępnych metod jest ograniczona. Nie możemy np. dodawać nowych elementów do krotki, ani usuwać z niej elementów istniejących.

Zadania (C):

1. Zmiennej „*zasadydna*” przypisz listę czterech zasad występujących w DNA (A, T, C, G).
2. W liście tej zamień tyminę na uracyl.
3. Odwróć kolejność zasad.
4. Zmiennej „*dna*” przypisz łańcuch odpowiadający następującej sekwencji DNA: TAACAATAAACCCAAAATGACTAAAAACACAATTCATAGTGATATTCGCA-GGAGTGAACATAACTTTCTTCCCCCAACACTTCCTAGGACTAGCAGGCAT-ACCACGACGATACTCAGACTACCCAGACGCCTACACAACA.
5. Następnie zmiennej „*dnalista*” przypisz listę powstałą z powyższego łańcucha znaków.
6. Odwróć listę „*dnalista*”.
7. Określ długość tej listy.
8. Podaj 10., 20. i 75 element listy „*dnalista*”.
9. Utwórz wycinek listy „*dnalista*” od elementu 30. do 100., przypisując go zmiennej „*nowalista*”.
10. Do nowej listy dodaj listę powstałą z następującego łańcucha: AGGCCCCGTAATCGGAATGAGCACACTCTAAAGACTTTAACGAGTATCCA TTGGAGGGCAAGTCTGGTGC.
11. Określ długość tak powstałej listy.
12. Stwórz listę o nazwie „*powtorzenia*” zawierającą sekwencję „TACG” powtórzoną 10 razy.
13. Utwórz krotkę z czterech zasad występujących w RNA (stosując ich pełne nazwy, tj. „adenina”, „cytozyna” itd.) przypisując ją nowej zmiennej o nazwie „*rna*”.

Cz. II – funkcje i słowniki

Do tej pory pracowaliśmy z językiem Python w sposób interaktywny. Wadą takiego postępowania jest bezpowrotna utrata wszystkich danych i zmiennych po zamknięciu okna. Dlatego też lepiej jest umieścić cały kod tworzonego programu w pliku tekstowym. Plik taki można stworzyć w dowolnym edytorze tekstu (np. notatniku). My jednak skorzystamy z edytora udostępnianego przez *IDLE* (*Python GUI*). W menu *File* wybieramy *New File* i zapisujemy nowo otwarty plik w folderze grupowym pod nazwą *skrypt.py* (wybierając *File >> Save As*).

W pliku o nazwie „*skrypt.py*” umieścimy następujący kod (znak „#” oznacza komentarz):

```
#Przypisujemy zmiennej a wartosc
a = 7.0
# Przypisujemy zmiennej b wartosc
b = 3.0
# Podajemy sume, roznice, iloczyn i iloraz a i b
print("Suma a i b: ", a + b)
print("Roznica a i b: ", a - b)
print("Iloczyn a i b: ", a * b)
print("Iloraz a i b: ", a / b)
```

W powyższym skrypcie zmiennym *a* i *b* przypisaliśmy dwie liczby rzeczywiste (używając kropki dziesiętnej). Zapisujemy plik w folderze grupowym.

Spróbujmy wykonać teraz powyższy skrypt, za pomocą opcji *Run* >> *Run Module*.

Do zmiennych *a* i *b* przypisane zostały wartości w trakcie tworzenia programu. Ale możemy pozwolić użytkownikowi samemu wprowadzić wartości podczas działania programu (funkcja „*float*" zamienia łańcuch znaków na liczbę rzeczywistą):

```
#Uzytkownik wprowadza pierwsza wartosc:
a = float(input("Podaj pierwsza liczbe: "))
# Uzytkownik wprowadza druga wartosc:
b = float(input("Podaj druga liczbe: "))
print("Suma a i b: ", a + b)
print("Roznica a i b: ", a - b)
print("Iloczyn a i b: ", a * b)
print("Iloraz a i b: ", a / b)
```

Ponownie uruchamiamy skrypt. Jeżeli zależy nam na tym, aby podczas działania programu użytkownik podał łańcuch, korzystamy także z funkcji „*input*" (ale już bez „*float*"):

```
imie = input("Podaj swoje imie: ")
kierunek = input("Podaj kierunek studiow: ")
print("Witaj", imie, "!")
print("Jak Ci sie studiuje na kierunku ", kierunek, "?")
```

Zadania (D):

1. Sporządź skrypt o nazwie „*skrypt2.py*", który poprosi użytkownika o podanie dowolnej sekwencji DNA, wyświetli tę sekwencję, poda jej długość, policzy, ile jest zasad każdego rodzaju, zamieni ją na mRNA, powtórnie wyświetli i poda liczbę zasad poszczególnych rodzajów. Efekt powinien w przybliżeniu wyglądać tak:

   ```
   Podaj sekwencje DNA: TTACGCCGTAAGCGT
   Oto sekwencja DNA: TTACGCCGTAAGCGT
   Dlugosc tej sekwencji: 15
   Liczba A: 3
   Liczba T: 4
   Liczba C: 4
   Liczba G: 4
   Oto Twoje mRNA: UUACGCCGUAAGCGU
   Liczba A: 3
   Liczba U: 4
   Liczba C: 4
   Liczba G: 4
   ```

2. Sporządź skrypt o nazwie „*obliczenia.py*", który dla wprowadzonych przez użytkownika wartości wyliczy sumę oraz średnią arytmetyczną. Wynik powinien być zbliżony do następującego:

   ```
   Podaj 1. liczbe: 5.0
   Podaj 2. liczbe: 10.0
   Podaj 3. liczbe: 15.0
   Podaj 4. liczbe: 7.0
   Suma liczb: 5.0; 10.0; 15.0; 7.0 to: 37.0
   Srednia liczb: 5.0; 10.0; 15.0; 7.0 to: 9.25
   ```

3. Rozszerz powyższy skrypt tak, aby podawał wartość minimalną i maksymalną zadanego ciągu liczb:

```
Podaj 1. liczbe: 5.0
Podaj 2. liczbe: 10.0
Podaj 3. liczbe: 15.0
Podaj 4. liczbe: 7.0
Suma liczb: 5.0; 10.0; 15.0; 7.0 to: 37.0
Srednia liczb: 5.0; 10.0; 15.0; 7.0 to: 9.25
Wartosc minimalna to: 5.0
Wartosc maksymalna to: 15.0
```

4. Utwórz modyfikację poprzedniego skryptu (pod nazwą „*obliczenia2.py*”) tak, aby obliczenia były wykonywane dla następującego ciągu liczb: 5,0; 7,5; 2,8; 3,51; 4,25; 10,0; 5,5; 6,7; 3,9; 6,9; 11,2; 12,1; 6,3; 11,05; 12,03 (wskazówka skorzystaj z funkcji „*sum*”):

```
Suma liczb to: 108.74
Srednia liczb to: 7.24933333333
Wartosc minimalna to: 2.8
Wartosc maksymalna to: 12.1
```

Literatura

1. Baxevanis AD, Ouellette BFF. 2005. Bioinformatyka. Podręcznik do analizy genów i białek. Wydawnictwo Naukowe PWN, Warszawa
2. Pilgrim M. 2011. Zanurkuj w Pythonie. http:/ / en. wikipedia. org/ wiki/ %3Azanurkuj_w_pythonie
3. pblaszczyk.zut.edu.pl/cwiczenie_python_2/python_3.pdf
4. Walesiak M, Gatnar E. 2009. Statystyczna analiza danych z wykorzystaniem programu R. PWN, Warszawa
5. The Python Tutorial. 2011. http://docs.python.org/tutorial/