Tomasz Topoła 2021 – zrealizowano w ramach projektu TechTok Polska™

Python w pigułce

wersja niekompletna, przedpremierowa

WPROWADZENIE

Python jest obecnie jednym z najpopularniejszych języków programowania. Cieszy się on tak dużą popularnością ze względu na łatwość jego interpretacji oraz prostotę składni. W porównaniu do innych języków takich jak C++, C#, czy Java jest on zdecydowanie mniej wydajny, jednak sprawdza się świetnie do pisania małych programów czy skryptów. Jest on świetnym wyborem na maturę, ponieważ można w nim szybko i łatwo napisać działający program, a przecież na maturze jesteśmy proszeni o rozwiązanie problemu. Nie ważne jak szybko, ważne, żeby działało. Ten dokument to swego rodzaju kompendium wiedzy potrzebnej na maturze z informatyki do rozwiązywania zadań z programowania. Jest on kierowany w szczególności do uczniów, którzy mają już jakąś wiedzę o programowaniu i chcą ją sobie usystematyzować. Osoby, które z programowaniem nigdy styczności nie miały mogą napotkać pewne problemy ze zrozumieniem przedstawianych tutaj treści. Ponadto wszystkie nazwy zmiennych i funkcji będą zapisane tutaj w języku angielskim – tak jak powinno się robić. Dzięki zastosowania anglojęzycznego nazewnictwa kod będzie bardziej czytelny dla osób z całego świata. Jest to praktyka stosowana w większości dużych projektów komercyjnych i warto jest wyrabiać sobie nawyk używania języka angielskiego w programowaniu.

BUDOWA JĘZYKA PYTHON

Deklarowanie zmiennych

Zmienne to coś w rodzaju szufladek na dane. Każda zmienna ma swoją nazwę i przypisaną do niej wartość. W Pythonie nie da się zadeklarować zmiennej bez nadawania jej żadnej wartości, jednak w innych językach istnieje taka możliwość. Zmienną deklarujemy przez wpisanie jej nazwy, a następnie podanie wartości poprzedzonej znakiem równości.

```
1.  variable  = 5
2.  variable1 = ‘a’
3.  variable2 = „Jakies Zdanie”
```

Do zmiennej możemy przypisywać dane różnego typu i wykonywać różne operacje. Na zmiennych liczbowych możemy wykonywać obliczenia matematyczne:

```
1.  a = 5.5
2.  b = 10.0
3.  addition = a + b
4.  subtraction = a – b
5.  multiplication = a * b
6.  division = a / b
```

Na zmiennych przechowujących ciągi znaków również możemy pracować

```
1.  word = „Hello ”
2.  word1 = ”world!”
3.  sentence = word + word1        # ta operacja spowoduje połączenie dwóch ciągów znaków
```

W zmiennej *sentence* zapisane jest teraz zdanie „Hello world!”

Operacje I/O (wejścia/wyjścia) w konsoli

Na początku przygody z programowaniem najczęściej tworzymy tak zwane programy konsolowe. Są one proste, ponieważ skupiamy się na samym tworzeniu algorytmów i tworzenie rozbudowanych interfejsów graficznych nie rozprasza naszej uwagi.

```
1.  print('Hello World!')       # wypisuje ciąg znaków do konsoli
2.  inputData = input()         # pobiera dane z konsoli i zapisuje do zmiennej inputData
```

Pokazane powyżej linijki kodu pozwalają na wypisanie i pobranie informacji do konsoli.

```
1.  print('Hello {user}!'.format(user='Tomek'))
```

Dzięki funkcji *.format(nazwa_zmiennej=wartość)* możemy zamienić tagi *{nazwa_zmiennej}* występujące w ciągu znaków na wartości. Ułatawia to wypisywanie danych do konsoli.

Podstawowe typy danych

Dane przypisane do zmiennych mają swój określony typ. Podczas pracy z językiem Python często będą pojawiały nam się błędy wynikające z niepoprawnego typu danych. Aby je naprawić należy zastosować odpowiednie funkcje do konwersji typów.

Typy do obsługi znaków z klawiatury

```
1.  inputData = str(inputData)
2.  inputData = ord(inputData)
3.  inputData = chr(inputData)
```

W pythonie zamianę typów danych przeprowadzamy za pomocą funkcji, która zwraca nową wartość. Powyżej możemy wyróżnić kolejno trzy typy danych do pracy z tekstem:

1. str – *string* – do przechowywania ciągów znaków – całych zdań lub wyrazów.
2. ord – *ordinal* – zamienia pojedynczy znak z klawiatury na kod *Unicode*
3. chr – *char* – zamienia kod *Unicode* na znak z klawiatury

Typy do obsługi zmiennych liczbowych

```
1.  inputData = float(inputData)
2.  inputData = int(inputData)
```

Do większości programów w zupełności wystarczą nam dwa typy liczbowe:

1. int – integer (*ang. liczba całkowita*) – przechowuje liczby całkowite, np. 5, -100, 28
2. float – floating point number (*ang. liczba zmiennoprzecinkowa*) – przechowuje ułamki w zapisie dziesiętnym, np. *1.28, 2.15, 1.0, 167.12*

Instrukcje warunkowe

Instrukcje warunkowe pozwalają na wykonywanie części kodu dopiero, gdy spełnione zostanie jedno lub więcej założenie.

Instrukcje warunkowe tworzymy według wzoru:

```
1.  if warunek1 <spójnik logiczny> warunek2:
2.      # operacje do wykonania gdy warunek zostanie spełniony
```

Operatory porównania

```
1.  if inputData == 5:                  # jedno równe drugiemu
2.     print('warunek 1. spelniony')
3.  if inputData > 5:                   # jedno większe od drugiego
4.      print('warunek 2. spelniony')
5.  if inputData >= 5:                  # jedno większe lub równe drugiemu
6.      print('warunek 3. spelniony')
7.  if inputData < 5:                   # jedno mniejsze od drugiego
8.      print('warunek 4. spelniony')
9.  if inputData <= 5:                  # jedno mniejsze lub równe drugiemu
10.     print('warunek 5. spelniony')
11. if inputData != 5:                  # jedno nie jest równe drugiemu
12.     print('warunek 6. spelniony')
```

Powyżej zaprezentowano wszystkie możliwe w Pythonie operatory porównania. Porównywać można zarówno wartości liczbowe jak i ciągi znaków.

Spójniki logiczne

```
1.  if inputData != 5 and inputData > 2
2.      print('warunek 7. spelniony')
3.  if inputData != 5 or inputData < 2:
4.      print('warunek 8. spelniony')
```

Spójniki logiczne pozwalają na zawieranie wielu warunków w jednej instrukcji if.

- And – odpowiada matematycznemu „i” – wszystkie warunki muszą zostać spełnione aby instrukcja się wykonała
- Or – odpowiada matematycznemu „lub” – conajmniej jeden warunek musi zostać spełniony aby instrukcja się wykonała

Blok if/elif/else

Instrukcje warunkowe można łączyć w bloki. Dzięki temu, kiedy jedna z instrukcji z bloku się wykona pozostałe zostaną zignorowane. Instrukcja if to pierwsza w bloku, elif to Pythonowy odpowiednik else if – gdy poprzedni warunek nie został spełniony to sprawdź ten. Else wykonuje się gdy wszystkie inne warunki nie zostały spełnione.

```
1.  if inputData > 5:
2.      print('wieksze od 5')
3.  elif inputData < 5:
4.      print('mniejsze od 5')
5.  else:
6.      print('rowne 5')
```

Tablice

Tablica to szczególny typ zmiennej pozwalający na przechowywanie wielu wartości pod jedną nazwą. Można w nich przechowywać dowolne typy danych, nie należy jednak ich ze sobą mieszać – samo przypisanie różnych typów do jednej tablicy nie wygeneruje błędu, ale ten może pojawić się przy późniejszym użyciu danych. Ostatni przykład prezentuje tą złą praktykę.

```
1.  charArray = ['a', 'b', 'c']
2.  numberArray =  [1, 2, 3, 4]
3.  badArray = ['a', 2, 1.54]  # pomieszane typy int, float i string – zła praktyka
```

Aby wykonać operację na danej z tablicy należy wywołać jej nazwę a następnie w nawiasach kwadratowych podać indeks danej. Należy pamiętać, że dane w tablicach indeksowane są od zera.

```
1.  print(charArray[0])     # w tej linijce drukujemy do konsoli wartość zerowego indeksu
    tablicy charArray, czyli 'a'
2.  print(charArray[3])     # indeks 3 dla podanej tablicy nie istnieje, więc program
    wyrzuci błąd "array index out of range"
```

Wielkość tablicy można pobrać za pomocą funkcji *len()*

```
1.  len(numberArray)      # podana funkcja zwróci wartość 4
```

Zapisy matematyczne

W celu przyspieszenia pracy w programach często można spotkać skrócone zapisy matematyczne

```
1.  a = 5
2.
3.  a = a + 2   # zapis pełny - zwiększ zmienną a o 2
4.  a += 2      # zapis skrócony
5.
6.  a = a - 2   # zapis pełny - zmniejsz zmienną a o 2
7.  a -= 2
8.
9.  a = a * 2   # zapis pełny - pomnóż zmienną a przez 2
10. a *= 2
11.
12. a = a / 2   # zapis pełny - podziel zmienną a przez 2
13. a /= 2
```

Warto jest również znać funkcjonalność modulo *%* pozwalającą na obliczanie reszty z dzielenia.

```
1.  a = 15 % 4
```

Pętle

Pętle służą do wykonywania czynności powtarzalnych. Kolejne wykonania zawartości pętli będziemy nazywać iteracjami.

Pętla „for"

Pętla for będzie wykonywać się dla podanej zmiennej i z każdym wykonaniem pętli będzie zmieniać jej wartość. Poniżej zaprezentowana pętla wykona się 20 razy nadając zmiennej *i* wartości z przedziału <0; 20)

```
1.  for i in range(20):
2.      print(i)
```

Pętlę for bardzo często stosuje się do operacji na indeksach tablicy

```
1.  for i in range(len(numberArray)):
2.      numberArray[i] += 5  #każdy indeks tablicy numberArray zostanie zwiększony o 5.
```

Jednak w języku Python jest lepsze rozwiązanie do wykonywania pętli na każdym elemencie tablicy, listy czy inej struktury danych – przez syntaks *for zmienna in tablica* do zmiennej przypisywane będą kolejne wartości z danej struktury danych.

```
1.  for element in numberArray:
2.      element -= 5        #każdy indeks tablicy zmniejszamy o 5.
```

Pętla „while"

Pętlę while najłatwiej można określić mianem „zapętlonego if-a" – wykonuje się tak długo jak podany warunek jest prawdziwy. W języku Python nie ma pętli *do-while*

```
1.  while a > 2:
2.      print(a)
3.      a -= 2
```

Dodatkowe operatory dla pętli

Podczas wykonywania kolejnych iteracji pętli może wystąpić potrzeba przerwania pętli lub przeskoczenia do kolejnej iteracji. W tym celu korzystamy z funkcji *continue* i *break*:

```
1.  for i in tablica:
2.    if i == 5:
3.              break
4.    if i < 10:
5.              continue
6.    i *= 100
```

Proste struktury danych

Struktury danych ułatwiają nam zapisywanie informacji w trakcie pracy programu. Dzięki nim nie musimy wykonywać pewnych operacji ręcznie. Tutaj wymienione zostały jedynie dwie struktury – lista oraz mapa. Powinny one w zupełności wystarczyć do potrzeb maturalnych.

Lista

Lista jest trochę bardziej rozbudowaną tablicą – można wykonywać na niej różne operacje:

- .append(value) – dodaje wartość *value* na koniec listy
- .pop(index) – usuwa z listy wartość znajdującą się pod podanym indeksem
- .insert(index, value) – pod podany indeks (index) wstawia wartość (value) przesuwając pozostałe indeksy do przodu.

```
1.  myList.append(0)
2.  myList.pop(2)
3.  myList.insert(2, 10)
```

Wśród operacji dostępnych do wykonania na tablicy znajduje się również funkcja .sort() układająca wszystkie elementy rosnąco. Należy jednak pamiętać, że takiej funkcji na maturze można użyć jedynie wtedy, gdy sortowanie nie jest częścią polecenia. Jeśli chcemy sortować malejąco należy jako parametr podać reverse=True

```
1.  myList.sort() # [3, 2, 10, 0] -> [0, 2, 3, 10]
2.  myList.sort(reverse=True) # [3, 2, 10, 0] -> [10, 3, 2, 0]
3.
```

Mapa

Mapa to struktura danych wykorzystująca klucz oraz wartość

```
1.  myMap = {'apple': 10, 'pear': 11}    # deklaracja (hardcode) mapy
2.  myMap['apple'] = 15                  # zmiana wartośći pod kluczem ‘apple’ na 15
```

Własne funkcje

Aby uniknąć powtarzania pewnych elementów kodu możemy przenieść je do funkcji. Funkcja w programowaniu jest swego rodzaju podprogramem. Zmienne zadeklarowane wewnątrz funkcji nie są dostępne poza nią i są usuwane zaraz po zakończeniu pracy funkcji. Funkcję można wywoływać wielokrotnie w obrębie jednego programu. Do deklarowania funkcji w języku Python służy słowo kluczowe *def*. Funkcję deklarujemy za pomocą następującego wzoru:

```
1.  def nazwaFunkcji(argumenty, oddzielone, przecinkiem):
2.    # zawartość funkcji
3.    return wynik        # jeśli funkcja nie zwraca wyniku to ta linijka nie jest potrzebna
```

Pracę funkcji kończy słowo return. Oznacza ono zwrócenie jakiejś wartości. Warto pamiętać, że słowo return można umieścić również w środku funkcji tak, aby ta została przerwana na przykład przez instrukcję warunkową:

```
1.  def square(a):
2.    if a < 0:           # aby policzyć pole kwadratu bok nie może mieć ujemnej długości
3.      return “a must not be a negative value”
4.    return a*a
```

Dobrym sposobem na zobrazowanie sobie czym jest funkcja w programowaniu będzie porównanie jej do funkcji matematycznej.

$$f(x) = x^2$$

Taką funkcję w Pythonie zapiszemy w następujący sposób:

```
1.  def f(x):
2.      result = x*x
3.      return result
```

Aby skrócić zapis możemy pominąć zapisywanie wyniku do zmiennej:

```
1.  def f(x):
2.      return x * x
```

Aby wywołać funkcję w naszym programie po prostu wpisujemy jej nazwę, a w nawiasach podajemy argumenty. Jeśli funkcja nie przyjmuje żadnych argumentów, to zapisujemy puste nawiasy.

```
1.  for i in range(1, 10):
2.      print(square(i))
```

Pokazany powyżej fragment kodu wypisze do konsoli pole kwadratu o boku z zakresu od 1 do 10 korzystając przy tym z wcześniej opisanej tutaj funkcji *square(a).*

Przy pisaniu własnych funkcji należy pamiętać o kilku zasadach, dzięki którym nasz kod będzie lepszej jakości i stanie się bardziej czytelny:

- Funkcja powinna zajmować się tylko jednym zadaniem na raz – np. jeśli jej zadaniem jest obliczenie pola koła, to powinna zwrócić tą wartość i nie robić nic poza tym – nie wypisywać danych do konsoli, nie pobierać promienia koła od użytkownika itd. Tym powinny zająć się oddzielne funkcje.
- Nazwa funkcja powinna określać jej zastosowanie. Dla odpowiednich zadań stosujemy zunifikowane nazewnictwo:
    - Funkcje sprawdzające czy coś jest prawdą nazywamy zaczynając od słowa *is*. Na przykład – funkcja sprawdzająca czy liczba jest pierwsza będzie nazwana *isPrime()*
    - Funkcje pobierające wartość zaczną się od słówka *get.* Na przykład funkcja pobierająca promień koła nazwie się *getRadius()*
    - Analogicznie funkcję ustawiającą promień nazwiemy *setRadius()*

  Do pozostałych funkcji stosujemy dowolne inne nazewnictwo pamiętając o tym, żeby nazwa opisywała ich zadanie.

Operacje I/O na plikach tekstowych

W zadaniach maturalnych praktycznie zawsze dane do programu dostarczane są w postaci pliku *.txt*. Teoretycznie uczeń nie musi pobierać tych danych za pomocą programu i w ostateczności mógłby się ratować ręcznym wprowadzeniem danych do zmiennej.

Otwieranie plików:

W języku python do otwierania plików służy funkcja open(path, mode). Plik można otworzyć w czterech różnych trybach:

- r – read (odczyt). Służy do odczytywania wartości z pliku. Jest to wartość domyślna, wyrzuca błąd gdy plik nie istnieje
- a – append (dodawanie). Służy do dopisywania wartości do pliku. Gdy plik nie istnieje zostanie utworzony.
- w -write (zapis). Służy do zapisywania danych do pliku. Jeśli plik nie jest pusty, to jego wartość zostanie usunięta. Jeśli plik nie istnieje to zostanie utworzony.
- X – create (tworzenie). Służy do stworzenia nowego pliku. Jeśli plik o podanej nazwie już istnieje, to pojawi się błąd.

```
1.  file = open(„example.txt”, „r”)          # odczyt
2.  file = open(„example.txt”, „a”)          # dodawanie
3.  file = open(„example.txt”, „w”)          # zapis
4.  file = open(„example.txt”, „x”)          # tworzenie
```

Gdy nie podamy żadnego Python otworzy plik w trybie read.

Przy pracy z plikami bardzo ważne jest, że na koniec **zawsze należy pamiętać o zamknięciu pliku!** Służy do tego metoda *.close()*

```
1.  file.close()
```

Odczyt danych z pliku:

Nasza funkcja otwiera plik, następnie wszystkie linijki zapisuje do tablicy *data*, zamyka plik i zwraca tablicę data. Zmienna *path* przechowuje ścieżkę do pliku.

```
1.  def readDataFromFile(path):
2.      file = open(path)
3.      data = file.readlines()
4.      file.close()
5.      return data
```

Zapis danych do pliku

W zależności od potrzeb należy otworzyć plik w jednym z wymienionych wcześniej trybów. Najlepszym wyborem będzie *a* – append lub *w* – write. Do dopisania ciągu znaków korzystamy z metody .write(). Aby wstawić „enter” posługujemy się tagiem „\n”

```
1.  def saveResultsToFile(path):
2.      file = open(path, 'a')
3.      file.write('ciąg znaków\n')
4.      file.close()
```

WYBRANE ALGORYTMY

Algorytmy przeszukujące i sortujące

Praktycznie każde zadanie na maturze rozszerzonej z informatyki wymaga od nas przeszukania lub posortowania pewnego zbioru danych. Przedstawione tu metody znacznie ułatwiają to zadanie.

Znajdywanie największej lub najmniejszej liczby

Nasza funkcja jako argument będzie przyjmowała tablicę liczb. W celu znalezienia największej liczby tworzymy zmienną do przechowania wyniku i zapisujemy do niej zerowy indeks z tablicy, którą sprawdzamy. Następnie w pętli for nadpisujemy zmienną z wynikiem za każdym razem, gdy liczba pod obecnym indeksem jest większa od wcześniej znalezionego wyniku.

```
1.  def findGreatest(data):
2.      result = data[0]
3.      for i in data:
4.          if i > result:
5.              result = i
6.      return result
```

Analogicznie najmniejszą liczbę możemy znaleźć zmieniając operator porównania w instrukcji warunkowej wewnątrz pętli

```
1.  def findLowest(data):
2.      result = data[0]
3.      for i in data:
4.          if i < result:
5.              result = i
6.      return result
```

Sposób z wykorzystaniem instrukcji warunkowych został pominięty ze względu na jego niską wydajność.

Sortowanie tablicy metodą bubble sort

Bubble sort to najprostsza metoda sortowania do zaimplementowana. Porównuje ona kolejno wszystkie elementy tablicy i przesuwa większe w prawo, a mniejsze w lewo.

```
1.  def bubbleSort(array):
2.      for i in range(len(array)):
3.          for j in range(0, len(array) - i - 1):
4.              if array[j] > array[j+1]:
5.                  temp = array[j]
6.                  array[j] = array[j+1]
7.                  array[j+1] = temp
8.      return array
```

Aby lepiej zrozumieć sposób działania algorytmu bubble sort warto jest obejrzeć jego interpretację graficzną: https://www.youtube.com/watch?v=Cq7SMsQBEUw

Warto jest również prześledzić działanie algorytmu za pomocą debugera.

Algorytmy do pracy z liczbami

Mimo że w praktyce bardzo rzadko korzysta się z tego typu algorytmów i są one najczęściej wbudowane w funkcje języka programowania, to egzaminatorzy maturalni lubią nas o nie zapytać. Algorytmy te mogą również przydać się podczas pracy z mikrokontrolerami, które mają zbyt małą ilość pamięci żeby pomieścić całą bibliotekę funkcji matematycznych.

Sprawdzanie czy liczba jest pierwsza czy złożona

Liczby pierwsze bardzo często pojawiają się we wszelkich zadaniach z programowania. Najlepszym sposobem na sprawdzenie, czy liczba jest pierwsza czy złożona jest upewnienie się, że reszta z dzielenia liczby przez wszystkie cyfry z zakresu *<2; sprawdzana_liczba/2 + 1>*. Dodatkowo w pętli for do górnej granicy tego zakresu dodajemy 1, ponieważ pętla ta wykonuje się dla *i < range.*

```
1.  def isPrime(number):
2.      maxDivision = int(number/2) + 1
3.      for i in range(2, maxDivision + 1):
4.          if number % i == 0:
5.              return False
6.      return True
```

Aby określić górną granicę zakresu liczb przez które dzielimy wykonujemy działanie sprawdzanaLiczba/2 + 1. Pętla nie może pracować na liczbach zmiennoprzecinkowych, więc wynik dzielenia zamieniamy na typ int – liczbę całkowitą. Jeśli przy którymkolwiek dzielniku z tego przedziału reszta będzie równa zero oznacza to, że sprawdzana liczba jest złożona.

Sprawdzanie czy liczba jest doskonała

Liczba doskonała to taka, która jest równa sumie wszystkich swoich dzielników mniejszych od tej liczby. Przykładem takiej liczby jest 6. Jej dzielnikami są 1, 2, 3 oraz 6, którego nie uwzględniamy. $1 + 2 + 3 = 6$, więc 6 jest liczbą doskonałą.

Aby sprawdzić dowolną liczbę za pomocą programu tworzymy funkcję, która kolejno znajduje wszystkie dzielniki podanej liczby n. Robimy to obliczając resztę z dzielenia liczby n przez wszystkie liczby z zakresu od n do $n - 1$. Jeśli wartość modulo jest równa zero, to dodajemy liczbę, przez którą dzieliliśmy do listy dzielników. Sumujemy dzielniki i porównujemy sumę z podaną liczbą. Na podstawie tego określamy, czy liczba jest doskonała czy nie:

```
1.  def isPerfect(n):
2.      dividers = []
3.      for i in range(1, n):
4.          if n % i == 0:
5.              dividers.append(i)
6.      total = 0
7.      for i in dividers:
8.          total += i
9.      if total == n:
10.         return True
11.     return False
```

Wyznaczanie NWD i NWW

Do wyznaczania największego wspólnego dzielnika i najmniejszej wspólnej wielokrotności posłużymy się zmodyfikowanym Algorytmem Euklidesa.

Wyznaczanie NWD – Algorytm Euklidesa

Algorytm Euklidesa sukcesywnie dzieli liczbę A przez liczbę B. Za każdym razem zapisuje do zmiennej resztę z tego dzielenia. Następnie liczbę A zastępuje wynikiem z dzielenia, a liczbę B resztą z dzielenia. W momencie gdy reszta z dzielenia jest równa zero liczba A stanowi NWD

```
1.  def getNWD(a, b):
2.      if b == 0:
3.          return a
4.      while b != 0:
5.          remainder = a % b
6.          a = b
7.          b = remainder
8.      return a
```

Wyznaczanie NWW

NWW wyznaczamy ze wzoru $NWW(a,b) = \frac{a \cdot b}{NWD(a,b)}$

```
1.  def getNWW(a, b):
2.      return (a * b)/getNWD(a, b)
```

Obliczanie pierwiastka kwadratowego

Do wyliczenia pierwiastka kwadratowego z liczby dodatniej zastosujemy metodę Newtona Raphsona. Polega ona na obliczeniu przybliżonej wartości pierwiastka przez podzielene podanej liczby przez dwa. Następnie aby wynik był jeszcze bliższy do prawdziwego wykonujemy obliczenie według wzoru:

$$lepszy\ wynik = \frac{1}{2}(przybliżona\ wartość + \frac{liczba\ do\ spierwiastkowania}{przybliżona\ wartość})$$

Wartości do tego wzoru podstawiamy do momentu w którym ponowne wykonanie obliczenia nie zmienia wyniku.

```
1.  def newtonSqrt(n):
2.      approx = 0.5 * n
3.      better = 0.5 * (approx + n / approx)
4.      while approx != better:
5.          approx = better
6.          better = 0.5 * (approx + n / approx)
7.      return approx
```

approx (ang. approximate „wartość przybliżona”) – przybliżona wartość pierwiastka

Algorytmy do pracy z tekstem

Podczas zadań wymagających pracy z tekstem warto jest pamiętać o kilku właściwościach zmiennych typu string i char. Po pierwsze należy pamiętać, że znaki z klawiatury są

przechowywane w formie numerów *Unicode* lub *ASCII* (*Unicode* to zbiór *ASCII* rozszerzony o znaki spoza alfabetu łacińskiego). Dzięki temu możemy łatwo zmieniać poszczególne litery. Gdy polecenie mówi „podnieś każdą literę o jeden – z ‘a’ na ‘b’, z ‘c’ na ‘d’ itd.”, to skorzystamy właśnie z kodu ASCII który zwyczajnie zwiększymy o 1. Aby pobrać numer Unicode danej litery skorzystamy z typu *ord:*

```
1.  unicodeNumber = ord('A')
```

Typ str podczas odczytu danych zachowuje się jak tablica – można odczytać literę kryjącą się pod danym indeksem. Niestety typ ten nie pozwala na nadpisywanie poszczególnych indeksów, więc musimy sobie do tego celu nasze zdanie przedstawić w postaci tablicy znaków. Zdaniem nazwijmy tablicę znaków, a słowem ciąg znaków typu str. Do zamiany słowa na zdanie użyjemy poniższej funkcji:

```
1.  def wordToSentence(word):
2.      sentence = []
3.      for i in word:
4.          sentence.append(i.lower())
5.      return sentence
```

Natomiast aby odwrócić ten proces użyjemy takiego algorytmu:

```
1.  def sentenceToWord(sentence):
2.      word = ""
3.      for i in sentence:
4.          word += i
5.      return word
```

Sprawdzanie palindromów

Palindrom to słowo, które czytane od tyłu brzmi tak samo jak oryginalne, np. *abba*. Aby sprawdzić, czy podany ciąg znaków jest palindromem sprawdzamy kolejno, czy ostatni indeks tego tekstu jest równy pierwszemu, przedostatni drugiemu itd. Gdy funkcja natrafi na sytuację, w której jeden znak nie jest równy drugiemu zwróci wartość *False* – tekst nie jest palindromem. Przy okazji tego algorytmu korzystamy z właściwości, że w typie String można pracować na indeksach tak samo jak w tablicy.

```
1.  def isPalindrome(text):
2.      textLength = len(text)
3.      for i in range(textLength):
4.          if text[i] != text[textLength-1-i]:
5.              return False
6.      return True
```

Sprawdzanie anagramów

Anagramy to słowa składające się z tych samych liter. Przykładem zestawu anagramów są *Adam*, *dama*. Możemy sprawdzić czy dwa słowa są anagramami porównując ilość występujących w nich liter.

Nasza funkcja przyjmuje za parametry dwa słowa – *text1* oraz *text2.* Funkcja w swoim działaniu nie będzie brała pod uwagę, czy litera jest duża czy mała. Z tego powodu obie zmienne z tekstem zamieniamy na same duże litery przy użyciu funkcji *.upper().*

```
1.  def isAnagram(text1, text2):
2.      text1 = text1.upper()
3.      text2 = text2.upper()
4.      if len(text1) != len(text2):
5.          return False
6.      for i in range(65, 91):              # główna pętla
7.          letterCounter = 0
8.          for j in range(len(text1)):      # wewnętrzna pętla
9.              if text1[j] == chr(i):
10.                 letterCounter += 1
11.             if text2[j] == chr(i):
12.                 letterCounter -= 1
13.         if letterCounter != 0:
14.             return False
15.     return True
```

Główna pętla wykonuje się dla $i \in <65; 91)$. Numery te odpowiadają znakom od ‘A’ (65) do ‘Z’ (90). Oznacza to, że *i* będzie kolejno przyjmować za swoją wartość wszystkie znaki alfabetu łacińskiego. Wewnętrzna pętla sprawdza czy w obu słowach litera znajdująca się w zmiennej *i* wystąpiła tyle samo razy. Licznik *letterCounter* będzie zwiększany o 1 za każdym razem, gdy w pierwszym słowie zostanie napotkana dana litera oraz zmniejszony o 1, gdy litera ta wystąpi w drugim słowie. Jeśli litera w zmiennej *i* występuje w obu słowach tyle samo razy, to po zakończeniu wewnętrznej pętli licznik ma wartość zero.

Algorytmy szyfrowania

Szyfr to sposób zapisu danych uniemożliwiający odczytanie ich nieuprawnionej osobie. Każdy szyfr ma swój klucz – jakąś daną pozwalającą na zaszyfrowanie lub odszyfrowanie wiadomości. Ze względu na występujące klucze szyfry można podzielić na:

- **Symetryczne**, w których ten sam klucz służy zarówno do zapisania jak i odczytania danych
- **Asymetryczne,** w których do szyfrowania i odszyfrowywania służą dwa różne klucze.

Poza tym szyfry można podzielić ze względu na sposób szyfrowania:

- **Podstawieniowe** – polegają na zmienianiu poszczególnych na inne
- **Przestawieniowe** – polegają na zmienianiu kolejności występowania znaków

Szyfr Cezara

Szyfr cezara jest szyfrem symetrycznym, podstawieniowym. Kluczem w szyfrze Cezara jest liczba, o którą należy przesunąć każdą z liter. Na przykład słowo *abba* po zaszyfrowaniu z kluczem o wartości *3* będzie brzmiało *deed*.

Klucz może być liczbą całkowitą z zakresu od -25 do 25[1]. Jeśli będzie dodatni to przesuwamy litery „w prawo”, a jeśli ujemny to „w lewo”.

Literze *a* w tablicy ASCII odpowiada numer *97*, natomiast literze *b* numer *98*. Aby dokonać szyfrowania musimy dodać klucz do kodu ASCII. Należy również uwzględnić przypadki, w których nasz kod będzie mniejszy od *97,* lub większy od *122*[2]. W pierwszej sytuacji literę zwiększymy o $klucz + 26$, a w drugiej o $klucz - 26$

Na egzaminie możemy zostać poproszeni zarówno o zaszyfrowanie jak i odszyfrowanie wiadomości. Na szczęście do tego celu wystarczy nam jedna funkcja. W momencie szyfrowania podamy jej normalny klucz, a w momencie deszyfrowania podamy jego odwrotną wartość. Na przykład, jeśli naszym kluczem szyfrującym jest liczba 3, to aby odszyfrować zdanie za pomocą tego samego algorytmu wystarczy za klucz podać wartość -3.

W poniższym algorytmie zmieniamy wszystkie litery w ciągu, więc nie musimy zamieniać słowa na tablicę. Możemy każdą literę pobrać, zmienić, a następnie zapisać do nowo utworzonej tablicy. Na koniec należy pamiętać, aby tę tablicę z powrotem zamienić na słowo jedną z wymienionych wcześniej funkcji.

```
1.  def cesar(word, key):
2.      sentence = []
3.      for i in range(len(word)):
4.          letterCode = ord(word[i])
5.          if ord(word[i]) > 122 or ord(word[i]) < 97:
6.              continue
7.          if letterCode + key > 122:
8.              letterCode += key - 26
9.          elif letterCode + key < 97:
10.             letterCode += key + 26
11.         else:
12.             letterCode += key
13.         sentence.append(chr(letterCode))
14.     return sentenceToWord(sentence)
```

Metody rekursywne

Rekursja (rekurencja) to sposób rozwiązywania problemów algorytmicznych polegający na wywołaniu funkcji przez samą siebie. Rekursja jest sposobem znacznie mniej wydajnym od iteracji, jednak w niektórych przypadkach łatwiej jest zapisać pewne obliczenia w sposób rekursywny niż za pomocą pętli.

Ciąg Fibonacciego

Kolejne wyrazy ciągu Fibonacciego wyznaczamy z poniższego wzoru:

$$a_n = a_{n-1} + a_{n-2}$$

[1] Jeśli chcemy posługiwać się dowolnymi liczbami naturalnymi podany klucz musi zostać wewnątrz funkcji zamieniony na resztę z dzielenia przez 26 (lub -26 dla ujemnego klucza).

[2] Są to kody ASCII odpowiadające małym literom *a* oraz *z* alfabetu łacińskiego. Dla liter wielkich posłużymy się kodami 65 i 90.

Dla *n* o wartości 0, 1, *a* przyjmuje wartości 0, 1. Można ten wzór zaimplementować na dwa sposoby:

Sposób Rekursyjny

```
1.  def fibonacci_recursive(n):
2.      if n > 1:
3.          return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)
4.      elif n == 0 or n == 1:
5.          return n
```

Sposób Iteracyjny

```
1.  def fibonacci_iterative(n):
2.      if n == 0 or n == 1:
3.          return n
4.      last = 1
5.      previous = 0
6.      result = 1
7.      for i in range(1, n):
8.          result = last + previous
9.          previous = last
10.         last = result
11.     return result
```

Jak widać na powyższym przykładzie w niektórych sytuacjach łatwiej jest wykonać dane obliczenia sposobem iteracyjnym. Problem jednak w tym, że funkcja ta będzie wywoływać sama siebie bardzo wiele razy. O ile nie jest to problemem przy dziesiątym wyrazie ciągu, tak gdybyśmy mieli obliczyć dziesięciotysięczny, to okazałoby się, że zajęlibyśmy całkiem sporą część pamięci RAM komputera. W sposobie iteracyjnym natomiast, mimo że kodu jest więcej, to jest on znacznie bardziej wydajny.

Silnia

Do problemu silni również można podejść na dwa sposoby. Mimo że jest to jeden z częstych przykładów na zastosowanie podejścia rekursyjnego można powiedzieć, że to sposób iteracyjny jest nawet bardziej przejrzysty

Sposób rekursyjny

```
1.  def factorial_recursive(n):
2.      if n == 1:
3.          return n
4.      else:
5.          return (n * factorial_recursive(n-1))
```

Sposób iteracyjny

```
1.  def factorial_iterative(n):
2.      result = 1
3.      for i in range(1, n+1):
4.          result *= i
5.      return result
```