Programowanie Python 1

(CP1S02005)

Politechnika Białostocka - Wydział Elektryczny

Cyfryzacja przemysłu, sem. II, studia stacjonarne I stopnia

Rok akademicki 2024/2025

Wykład nr 2 (18.03.2025)

dr inż. Jarosław Forenc

Plan wykładu nr 2

- Pętle
  - pętla for, funkcja range()
  - instrukcje break i continue
  - pętla while
- Ciągi tekstowe
  - implementacja, sposób zapisu, odwołania do elementów, metody
  - porównywanie ciągów tekstowych, zastosowanie operatorów + i *
- Listy
  - implementacja, metody tworzenia, lista składana
  - indeksy elementów, wycinki, funkcje i metody
- Krotki
  - Implementacja, metody tworzenia
  - indeksy elementów, wycinki

Python - pętla for

- Pętla for jest używana do iteracji przez elementy kolekcji lub innego iterowanego obiektu
- Składnia pętli for:

```
for element in kolekcja:
    # kod wykonywany dla każdego elementu
```

  - element - zmienna przyjmująca wartości elementów z kolekcji podczas każdej iteracji
  - kolekcja - lista, krotka, słownik, zbiór lub inna kolekcja iterowalna
- Podczas każdej iteracji element przyjmuje wartości kolejnego elementu z kolekcji i dla tej wartości wykonywane są instrukcje

Python - funkcja range()

- Funkcja range() służy do generowania serii liczb całkowitych, używana jest powszechnie w iteracjach, zwłaszcza w pętli for

```
range(start, stop, step)
```

  - start - liczba całkowita, od której zaczyna się seria, domyślnie ma wartość 0 (parametr opcjonalny)
  - stop - liczba całkowita, na której kończy się seria (nie jest wliczana w serię)
  - step - krok, o który seria się zwiększa, domyślnie ma wartość 1 (parametr opcjonalny)

- Zwraca obiekt typu „range”, który reprezentuje serię liczb całkowitych
- Nie generuje listy od razu, ale tworzy obiekt, który generuje liczby na żądanie

Python - pętla for i funkcja range()

- generowanie liczb całkowitych z zakresu [0, 4]

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

```
for i in range(5):
    print(i, end = " ")
print()
```

```
0 1 2 3 4
```

Python - pętla for i funkcja range()

- generowanie liczb całkowitych z zakresu [2, 4]

```
for i in range(2,5):
    print(i, end = " ")
```

```
2 3 4
```

- generowanie liczb całkowitych z zakresu [1, 10] z krokiem 2

```
for i in range(1,11,2):
    print(i, end = " ")
```

```
1 3 5 7 9
```

Python - pętla for i funkcja range()

- błędne wartości start i stop spowodują wygenerowanie pustego ciągu

```
for i in range(9,0):
    print(i, end = " ")
```

```

```

- wartość step może być ujemna

```
for i in range(9,0,-1):
    print(i, end = " ")
```

```
9 8 7 6 5 4 3 2 1
```

Python - pętla for i funkcja range()

- wygenerowanie n+1 liczb rzeczywistych z przedziału [start, stop]

```
start = 0
stop = 1
n = 10
step = (stop - start) / n
for i in range(n + 1):
    value = start + i * step
    print(value)
```

```
0.0
0.1
0.2
0.30000000000000004
0.4
0.5
0.6000000000000001
0.7000000000000001
0.8
0.9
1.0
```

- w pętli for może być wykonanych kilka instrukcji (wymagają wcięcia o takiej samej wielkości)

Python - pętla for i instrukcja break

- instrukcja break jest używana wewnątrz pętli for, aby przerwać jej działanie, niezależnie od tego, ile iteracji zostało wykonanych
- Przykład: poszukiwanie pierwszej liczby podzielnej zarówno przez 3, jak i przez 7 w przedziale [1, 100]

```
for liczba in range(1, 101):
    if liczba % 3 == 0 and liczba % 7 == 0:
        print("Znaleziona liczba to:", liczba)
        break
```

```
Znaleziona liczba to: 21
```

Python - pętla for i instrukcja break

- instrukcja break jest używana wewnątrz pętli for, aby przerwać jej działanie, niezależnie od tego, ile iteracji zostało wykonanych
- Przykład: poszukiwanie pierwszej liczby podzielnej zarówno przez 3, jak i przez 7 w przedziale [1, 100] (wynik bez break)

```
for liczba in range(1, 101):
    if liczba % 3 == 0 and liczba % 7 == 0:
        print("Znaleziona liczba to:", liczba)
```

```
Znaleziona liczba to: 21
Znaleziona liczba to: 42
Znaleziona liczba to: 63
Znaleziona liczba to: 84
```

Python - pętla for i instrukcja continue

- instrukcja continue służy do przerwania aktualnej iteracji pętli for i przejścia do następnej iteracji
- Przykład: wyświetlenie tylko liczb nieparzystych z przedziału [1, 10]

```
for liczba in range(1, 11):
    if liczba % 2 == 0:
        continue
    print(liczba, end = " ")
```

```
1 3 5 7 9
```

Python - pętla for, najczęstsze błędy

- brak dwukropka (:)

```
for i in range(5)
    print(i)
```

```
PS C:\Users\jaros> python -u "d:\MyApp.py"
  File "d:\MyApp.py", line 1
    for i in range(5)
                     ^
SyntaxError: expected ':'
```

Python - pętla for, najczęstsze błędy

- brak wcięcia w instrukcji po pętli for

```
for i in range(5):
print(i)
```

```
PS C:\Users\jaros> python -u "d:\MyApp.py"
  File "d:\MyApp.py", line 2
    print(i)
    ^
IndentationError: expected an indented block
                after 'for' statement on line 1
```

Python - pętla for, najczęstsze błędy

- niepotrzebne wcięcie

```
for i in range(5):
    print(i)
    print("Koniec")
```

```
0
Koniec
1
Koniec
2
Koniec
3
Koniec
4
Koniec
```

Python - pętla for, najczęstsze błędy

- próba iterowania obiektu, który nie jest iterowalny

```
value = 10
for i in value:
    print(i)
```

```
PS C:\Users\jaros> python -u "d:\MyApp.py"
Traceback (most recent call last):
  File "d:\MyApp.py", line 2, in <module>
    for i in value:
TypeError: 'int' object is not iterable
```

Python - zagnieżdżanie pętli for

- Przykład: tabliczka mnożenie

```
for i in range(1, 11):
    for j in range(1, 11):
        wynik = i * j
        print(f"{wynik:4}", end=" ")
    print()
```

```
   1    2    3    4    5    6    7    8    9   10
   2    4    6    8   10   12   14   16   18   20
   3    6    9   12   15   18   21   24   27   30
   4    8   12   16   20   24   28   32   36   40
   5   10   15   20   25   30   35   40   45   50
   6   12   18   24   30   36   42   48   54   60
   7   14   21   28   35   42   49   56   63   70
   8   16   24   32   40   48   56   64   72   80
   9   18   27   36   45   54   63   72   81   90
  10   20   30   40   50   60   70   80   90  100
```

Python - pętla while

- Pętla while służy do wykonywania bloku kodu dopóki warunek jest spełniony (przyjmuje wartość True)
- Składnia pętli while:

```
while warunek:
    # kod wykonywany dopóki warunek jest prawdziwy
```

  - Przykład:

```
numer = 1
while numer <= 5:
    print(numer)
    numer = numer + 1
```

```
1
2
3
4
5
```

Python - pętla while i instrukcja break

- instrukcja break jest używana wewnątrz pętli while, aby przerwać jej działanie
- Przykład: poszukiwanie pierwszej liczby podzielnej zarówno przez 3, jak i przez 7 w przedziale [1, 100]

```
liczba = 1
while liczba <= 100:
    if liczba % 3 == 0 and liczba % 7 == 0:
        print("Znaleziona liczba to:", liczba)
        break
    liczba += 1
```

```
Znaleziona liczba to: 21
```

Python - pętla while i instrukcja continue

- instrukcja continue służy do przerwania aktualnej iteracji pętli while i powrót na początek pętli
- Przykład: wyświetlenie tylko liczb nieparzystych z przedziału [1, 10]

```
liczba = 0
while liczba < 10:
    liczba = liczba + 1
    if liczba % 2 == 0:
        continue
    print(liczba, end = " ")
```

```
1 3 5 7 9
```

Python - ciągi tekstowe

- Ciąg tekstowy to seria znaków, które służą do przechowywania informacji (danych) tekstowych
- Ciąg tekstowy może być ujęty w cudzysłów lub apostrofy

```
napis1 = "Witaj świecie!"
napis2 = 'Witaj swiecie!'

napis3 = """Witaj świecie!"""
napis4 = '''Witaj świecie!'''
```

  - w przypadku ograniczenia trzema znakami, ciąg tekstowy może znajdować się w więcej niż jednym wierszu kodu programu

- Ciągi tekstowe są obiektami klasy str
  - https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str

Python - ciągi tekstowe

- nie można jednocześnie stosować cudzysłowu i apostrofu do ograniczenia tekstu

```
napis1 = "Witaj świecie!'
```

```
File "d:\MyApp.py", line 1
    napis1 = "Witaj świecie!'
             ^
SyntaxError: unterminated string literal (detected at line 1)
```

```
napis1 = 'Witaj świecie!"
```

```
File "d:\MyApp.py", line 1
    napis1 = 'Witaj świecie!"
             ^
SyntaxError: unterminated string literal (detected at line 1)
```

Python - ciągi tekstowe

- można stosować inne znaki (cudzysłów / apostrof) jako cytaty

```
kod1 = "Przedmiot 'Programowanie C' (CP1S01005)"
kod2 = 'Przedmiot "Programowanie Python 1" (CP1S02005)'
```

- umieszczenie znaku cudzysłowu (") w tekście ograniczonym tymi samymi znakami wymaga dodania znaku \ (ang. backslash)

```
kod1 = "Przedmiot \"Programowanie C\" (CP1S01005)"
```

- umieszczenie znaku apostrofu (') w tekście ograniczonym tymi samymi znakami wymaga dodania znaku \ (ang. backslash)

```
kod2 = 'Przedmiot \'Programowanie Python 1\' (CP1S02005)'
```

Python - ciągi tekstowe

- przykład występowania znaku apostrofu w tekście

```
tekst1 = "Prawo Ampere'a"
tekst2 = "Kupiłem nowego iPhone'a"

print(tekst1)
print(tekst2)
```

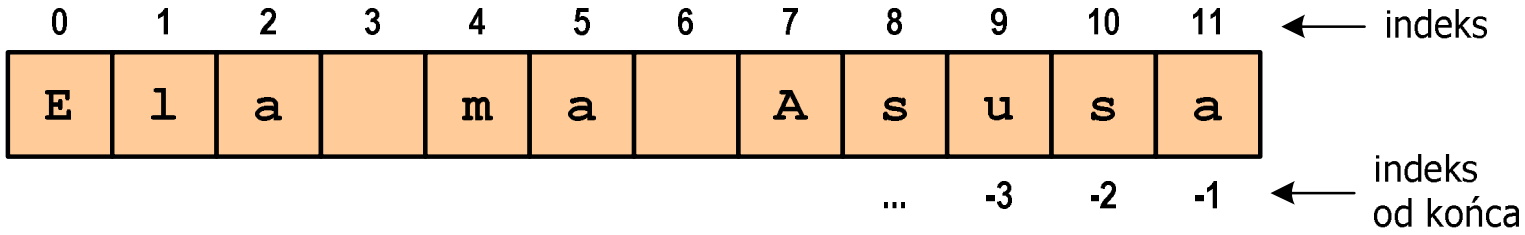
```
Prawo Ampere'a
Kupiłem nowego iPhone'a
```

- uwaga: apostrof stawiamy wtedy, gdy ostatnia litera wyrazu obcego nie jest wymawiana w języku polskim

Python - ciągi tekstowe (odwołania do elementów)

- ciąg tekstowy jest typem sekwencyjnym - możliwy jest zatem dostęp do dowolnego elementu poprzez podanie jego indeksu

```
tekst = "Ela ma Asusa"
```



- odwołanie ma postać: nazwa_ciągu[indeks]

```
tekst = "Ela ma Asusa"

print(f"Pierwszy znak od początku: {tekst[0]}")
print(f"Drugi znak od początku:    {tekst[1]}")
print(f"Drugi znak od końca:       {tekst[-2]}")
```

```
Pierwszy znak od początku: E
Drugi znak od początku:    l
Drugi znak od końca:       s
```

Python - ciągi tekstowe (odwołania do elementów)

- zastosowanie dwukropka (:) do tworzenia indeksów elementów

```
lista[indeks_początkowy : indeks_końcowy : krok]
```

- każdy z indeksów może być pominięty, wtedy przyjmowane są wartości domyślne:
  - indeks_początkowy: 0 (indeks pierwszego elementu)
  - indeks_końcowy: len(lista) (indeks ostatniego elementu)
  - krok: 1

```
tekst = "Ela ma Asusa"
print(f"Trzy pierwsze znaki: {tekst[0:3]}")    # 0,1,2
```

```
Trzy pierwsze znaki: Ela
```

Python - ciągi tekstowe (odwołania do elementów)

```
tekst = "Ela ma Asusa"
print(f"Trzy pierwsze znaki: {tekst[:3]}")
print(f"Tekst bez trzech pierwszych znaków: {tekst[3:]}")
```

```
Trzy pierwsze znaki: Ela
Tekst bez trzech pierwszych znaków:  ma Asusa
```

```
tekst = "Ela ma Asusa"
print(f"Ostatni znak: {tekst[-1]}")
print(f"Dwa ostatnie znaki: {tekst[-2:]}")
print(f"Bez dwóch ostatnich znaków: {tekst[0:-2]}")
```

```
Ostatni znak: a
Dwa ostatnie znaki: sa
Bez dwóch ostatnich znaków: Ela ma Asu
```

Python - ciągi tekstowe (odwołania do elementów)

```
tekst = "Ela ma Asusa"
print(f"Co drugi element od pierwszego: {tekst[::2]}")
print(f"Co drugi element od drugiego: {tekst[1::2]}")
```

```
Co drugi element od pierwszego: Eam ss
Co drugi element od drugiego: l aAua
```

```
tekst = "Ela ma Asusa"
print(f"Od końca: {tekst[::-1]}")
```

```
Od końca: asusA am alE
```

Python - ciągi tekstowe (wybrane metody)

- funkcja len() - zwraca długość ciągu tekstowego (liczbę znaków)

```
tekst = input("Podaj tekst: ")
dlugosc = len(tekst)
print(f"Liczba wprowadzonych znaków: {dlugosc}")
```

```
Podaj tekst: Programowanie Python 1
Liczba wprowadzonych znaków: 22
```

Python - ciągi tekstowe (wybrane metody)

- metoda title() - zamienia pierwsze litery każdego wyrazu na wielkie (nie modyfikuje oryginalnego tekstu)

```
napis = "ela ma asusa"
print(napis.title())
print(napis)
```

```
Ela Ma Asusa
ela ma asusa
```

- trwała zmiana wielkości liter w tekście

```
napis = "ela ma asusa"
napis = napis.title()
print(napis)
```

```
Ela Ma Asusa
```

Python - ciągi tekstowe (wybrane metody)

- metoda upper() - zamienia małe litery na wielkie

```
napis = "Ela ma Asusa"
print(napis.upper())
```

```
ELA MA ASUSA
```

- metoda lower() - zamienia wielkie litery na małe

```
napis = "Ela ma Asusa"
print(napis.lower())
```

```
ela ma asusa
```

Python - ciągi tekstowe (wybrane metody)

- metoda removeprefix() - usuwa prefiks z łańcucha znaków (jeśli istnieje)

```
url = "https://we.pb.edu.pl"
url = url.removeprefix("https://")
print(url)
```

```
we.pb.edu.pl
```

- metoda removesuffix() - usuwa sufiks z łańcucha znaków (jeśli istnieje)

```
fname = "ocena_python.txt"
print(f"Nazwa pliku: {fname.removesuffix(".txt")}")
```

```
Nazwa pliku: ocena_python
```

Python - ciągi tekstowe (wybrane metody)

- metoda lstrip() - usuwa białe znaki z lewej strony ciągu tekstowego (na początku tekstu)
- metoda rstrip() - usuwa białe znaki z prawej strony ciągu tekstowego (na końcu tekstu)
- metoda strip() - usuwa białe znaki z lewej i z prawej strony ciągu tekstowego (na początku i na końcu tekstu)
- białe znaki: spacje (" "), tabulatory ("\t"), znaki nowej linii ("\n")

```
tekst = "   Jan Kowalski  "

print(f"[{tekst}]")
print(f"[{tekst.lstrip()}]")
print(f"[{tekst.rstrip()}]")
print(f"[{tekst.strip()}]")
```

```
[   Jan Kowalski  ]
[Jan Kowalski  ]
[   Jan Kowalski]
[Jan Kowalski]
```

Python - ciągi tekstowe (wybrane metody)

- metody lstrip(), rstrip() i strip() mogą mieć argument w postaci zestawu innych znaków do usunięcia

```
tekst = "###Jan Kowalski###"

print(f"[{tekst}]")
print(f"[{tekst.lstrip("#")}]")
print(f"[{tekst.rstrip("#")}]")
print(f"[{tekst.strip("#")}]")
```

```
[###Jan Kowalski###]
[Jan Kowalski###]
[###Jan Kowalski]
[Jan Kowalski]
```

Python - ciągi tekstowe (wybrane metody)

- metoda startswith(prefix) - zwraca wartość True jeśli dany ciąg tekstowy zaczyna się od podanego prefiksu

```
tekst = "Witaj świecie"
if tekst.startswith("Witaj"):
    print("Ciąg zaczyna się od: 'Witaj'")
else:
    print("Ciąg nie zaczyna się od: 'Witaj'")
```

- metoda endswith(suffix) - zwraca wartość True jeśli dany ciąg tekstowy kończy się podanym sufiksem

```
tekst = "Witaj świecie"
if tekst.endswith("świecie"):
    print("Ciąg kończy się na: 'świecie'")
else:
    print("Ciąg nie kończy się na: 'świecie'")
```

Python - ciągi tekstowe (wybrane metody)

- metoda count(substring) - zwraca liczbę wystąpień określonego podciągu w danym ciągu

```
tekst = "Ela ma laptopa"
ile_a = tekst.count("a")
print(f"Liczba wystąpień litery 'a': {ile_a}")
```

```
Liczba wystąpień litery 'a': 4
```

Python - ciągi tekstowe (wybrane metody)

- metoda find(substring) - wyszukuje określony podciąg w danym ciągu
- zwraca indeks pierwszego wystąpienia tego podciągu lub -1 jeśli podciąg nie został znaleziony

```
tekst = "Ela ma laptopa"
indeks = tekst.find("laptop")
if indeks == -1:
    print("Brak podciągu")
else:
    print(f"Początek podciągu to elementu nr: {indeks}")
```

```
Początek podciągu to elementu nr: 7
```

- metoda rfind(substring) - zwraca indeks ostatniego wystąpienia podciągu w ciągu lub zwraca -1 jeśli podciąg nie został znaleziony

Python - porównywanie ciągów tekstowych

- do ciągów tekstowych można zastosować operatory porównania

Operator	Znaczenie	Operator	Znaczenie
==	równa się	!=	nie równa się
>	większe niż	>=	mniejsze lub równe
<	mniejsze niż	<=	większe lub równe

- w wyniku porównania otrzymujemy wartość True lub False

```
pass = input("Podaj hasło: ")
if pass == "123456":
    print("To jest najpopularniejsza hasło na świecie")
else:
    print("Hasło jest OK")
```

- dwa ciągi tekstowe są sobie równe, gdy składają się z takich samych znaków umieszczonych na identycznych pozycjach

Python - porównywanie ciągów tekstowych

- przy porównywaniu tekstów należy zwrócić uwagę na wielkość liter

```
imie = input("Jak na imię miał Kopernik? ")
if imie.lower() == "mikołaj":
    print("Poprawna odpowiedź!")
else:
    print("Błędna odpowiedź!")
```

```
Jak na imię miał Kopernik? Mikołaj
Poprawna odpowiedź!
```

```
Jak na imię miał Kopernik? mikołaj
Poprawna odpowiedź!
```

```
Jak na imię miał Kopernik? mikolaj
Błędna odpowiedź!
```

Python - ciągi tekstowe (przykład)

- określenie liczby cyfr w wierszu tekstu

```
tekst = (input("Podaj tekst: "))
ile = 0
for znak in tekst:
    if ord(znak) >= 48 and ord(znak) <= 57:
        ile = ile + 1
print(f"W tekście jest {ile} cyfr")
```

```
Podaj tekst: asd58Dr4Hik2189
W tekście jest 7 cyfr
```

- funkcja ord() - argumentem jest pojedynczy znak (typu str), zwraca odpowiadającą mu wartość liczbową Unicode tego znaku

Python - ciągi tekstowe i operatory: +, *

- operator + jest używany do łączenia dwóch ciągów tekstowych w jeden

```
imie = "Jan"
nazwisko = "Kowalski"
osoba = imie + " " + nazwisko
print(osoba)
```

```
Jan Kowalski
```

- operator * jest używany do powielania ciągu określoną liczbę razy

```
tekst = "<->"
nowy = tekst * 10
print(nowy)
```

```
<-><-><-><-><-><-><-><-><-><->
```

Python - listy

- Lista (ang. list) - modyfikowalna struktura danych przechowująca kolekcję elementów ułożonych w określonej kolejności

```
liczby = [5, 2, 8, 3, 0, 2, 1]
funkcje = ["print", "input", "sort", "sqrt"]
```

  - elementy listy umieszczane są wewnątrz nawiasów kwadratowych i rozdzielane przecinkami
  - listy mogą zawierać elementy różnych typów danych (liczby, łańcuchy znaków, inne listy, krotki, słowniki)
  - elementy listy nie muszą być ze sobą powiązane (ale zazwyczaj są)
  - listy są dynamiczne - w trakcie działania programu można dodawać do nich elementy i usuwać je

Python - krotki

- Krotka (ang. tuple) - niemodyfikowalna struktura danych przechowująca kolekcję elementów ułożonych w określonej kolejności (lista elementów, której nie można zmienić)

```
liczby = (5, 2, 8, 3, 0, 2, 1)
funkcje = ("print", "input", "sort", "sqrt")
```

  - elementy krotki umieszczane są wewnątrz nawiasów zwykłych i rozdzielane przecinkami
  - krotki mogą zawierać elementy różnych typów danych (liczby, łańcuchy znaków, listy, inne krotki, słowniki)
  - zajmują mniej miejsca w pamięci niż listy, co czyni je bardziej efektywnymi, gdy dane będą tylko odczytywane

Python - listy

- do wyświetlenia listy można zastosować funkcję print()

```
bierne = ["rezystor", "cewka", "kondensator"]
print(bierne)
```

```
['rezystor', 'cewka', 'kondensator']
```

```
moja_lista = [1, 2, 3, "a", "b", "c", 2.5]
czynne = []
print(moja_lista)
print(czynne)
```

```
[1, 2, 3, 'a', 'b', 'c', 2.5]
[]
```

Python - lista, metody tworzenia

- użycie pary nawiasów kwadratowych do oznaczenia pustej listy

```
moja_lista = []
print(moja_lista)
```

```
[]
```

- użycie nawiasów kwadratowych, oddzielenie elementów przecinkami

```
imiona1 = ["Ala"]
imiona2 = ["Piotr", "Kuba", "Kacper"]
print(imiona1)
print(imiona2)
```

```
['Ala']
['Piotr', 'Kuba', 'Kacper']
```

Python - lista, metody tworzenia

- zastosowanie konstruktora bezargumentowego: list()

```
moja_lista = list()
print(moja_lista)
```

```
[]
```

- zastosowanie konstruktora z argumentem w postaci obiektu iterowalnego: list(iterable)
- obiekt iterowalny może być sekwencją, kontenerem obsługującym iterację lub obiektem iteratora
- konstruktor buduje listę, której elementy są takie same i w tej samej kolejności, co elementy obiektu iterowalnego

Python - lista, metody tworzenia

- inna lista jako argument konstruktora: list(iterable)

```
imiona = ["Piotr", "Kuba", "Kacper"]
names = list(imiona)
print(names)
```

```
['Piotr', 'Kuba', 'Kacper']
```

- inne obiekty iterowalne jako argumenty konstruktora: list(iterable)

```
litery = list("ABCDE")
liczby = list((1, 2, 3, 4, 5))
print(litery)
print(liczby)
```

```
['A', 'B', 'C', 'D', 'E']
[1, 2, 3, 4, 5]
```

Python - lista, metody tworzenia

- zastosowanie listy składanej (ang. list comprehension)
- lista składana łączy w pojedynczym wierszu kodu pętlę for, utworzenie nowego elementu i jego automatyczne dołączenie do listy

```
nazwa_listy = [wyrażenie for element in sekwencja]
```

- przykład: lista kwadratów liczb z przedziału od 1 do 10

```
kwadraty = [x**2 for x in range(1,11)]
print(kwadraty)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- pętla for generuje liczby x dla wyrażenia, wartości wyrażenia dodawane są do listy kwadraty (uwaga: po pętli for nie ma dwukropka)

Python - lista, metody tworzenia

- zastosowanie listy składanej (ang. list comprehension)
- możliwe jest również dodanie warunku do listy składanej w celu filtrowania elementów

```
nazwa = [wyrażenie for element in sekwencja if warunek]
```

- przykład: lista kwadratów liczb parzystych z przedziału od 1 do 10

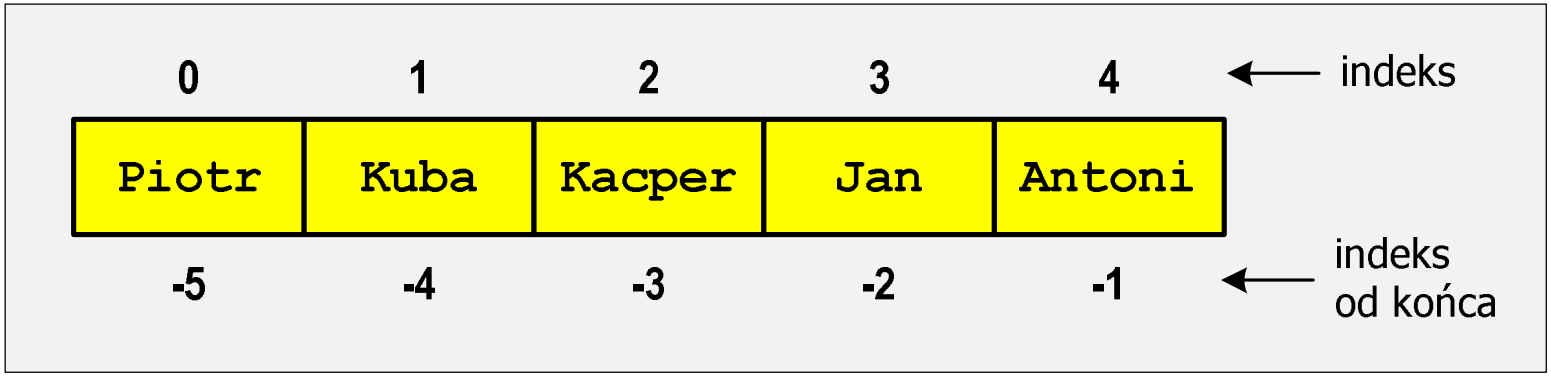
```
kwadraty = [x**2 for x in range(1,11) if x % 2 == 0]
print(kwadraty)
```

```
[4, 16, 36, 64, 100]
```

Python - lista, indeksy elementów

- lista jest uporządkowaną kolekcją, dostęp do jej dowolnego elementu można uzyskać przez podanie jego położenia (indeksu)
- pierwszy element listy ma indeks równy 0, drugi element listy ma indeks równy 1, itd.
- ostatni element listy ma dodatkowy indeks wynoszący -1, drugi element od końca listy ma indeks -2, itd.

```
imiona = ["Piotr", "Kuba", "Kacper", "Jan", "Antoni"]
```



Python - lista, indeksy elementów

- odwołując się do elementów listy podajemy nazwę listy i w nawiasach kwadratowych numer elementu: nazwa_listy[indeks]

```
imiona = ["Piotr", "Kuba", "Kacper", "Jan", "Antoni"]
print(f"Witaj {imiona[1]}!")
print(f"Witaj {imiona[-1]}!")
```

```
Witaj Kuba!
Witaj Antoni!
```

- użycie nieprawidłowego indeksu spowoduje błąd kompilacji

```
print(f"Witaj {imiona[5]}!")
```

```
Traceback (most recent call last):
  File "d:\MyApp.py", line 2, in <module>
    print(f"Witaj {imiona[5]}!")
                   ~~~~~~^^^
IndexError: list index out of range
```

Python - lista, indeksy elementów

- stosując dwukropek (:) można pracować na fragmentach listy nazywanych wycinkami (ang. slices)

```
nazwa_listy[indeks_początkowy : indeks_końcowy : krok]
```

- wycinek rozpoczyna się od elementu o podanym indeksie początkowym i kończy na elemencie o indeksie o jeden mniejszym niż podany indeks końcowy
- każdy z elementów w nawiasach kwadratowych może być pominięty, wtedy przyjmowane są wartości domyślne
- indeks początkowy - wartość domyślna to 0 (indeks pierwszego elementu)
- indeks końcowy - wartość domyślna to len(lista) (liczba elementów listy)
- krok - wartość domyślna to 1

Python - lista, indeksy elementów

- przykłady tworzenia wycinków

```
imiona = ["Piotr", "Kuba", "Kacper", "Jan", "Antoni"]

print(imiona[0:3])
print(imiona[:4])
print(imiona[-3:])

print(imiona[::2])

print(imiona[:])
```

```
['Piotr', 'Kuba', 'Kacper']
['Piotr', 'Kuba', 'Kacper', 'Jan']
['Kacper', 'Jan', 'Antoni']

['Piotr', 'Kacper', 'Antoni']

['Piotr', 'Kuba', 'Kacper', 'Jan', 'Antoni']
```

Python - listy, funkcje i metody

Funkcja	Wynik
`len(lista)`	zwraca liczbę elementów znajdujących się na liście
`min(lista)`	zwraca wartość najmniejszego elementu listy
`max(lista)`	zwraca wartość największego elementu listy
`sum(lista)`	zwraca sumę elementów znajdujących się na liście
`sum(lista,start=0)`	zwraca sumę elementów znajdujących się na liście, start to opcjonalny parametr określający wartość początkową sumy (domyślnie 0)
`sorted(lista)`	zwraca posortowaną listę elementów, nie modyfikuje oryginalnej listy

Python - listy, funkcje i metody

Funkcja	Wynik
`del lista[i:j]`	usuwa elementy listy o indeksach od i do j-1, ten sam wynik daje: lista[i:j] = []
`del lista[i:j:k]`	usuwa elementy listy o indeksach od i do j-1 z krokiem k
`lista.append(x)`	dodaje element x na końcu listy
`lista.clear()`	usuwa wszystkie elementy z listy, ten sam wynik daje: del lista[:]
`lista.copy(x)`	tworzy kopię listy, ten sam wynik daje: lista[:]
`lista.extend(t)`	dodaje na końcu listy zawartość obiektu t

Python - listy, funkcje i metody

Funkcja	Wynik
`lista.insert(i,x)`	wstawia do listy wartość x na pozycji o indeksie i
`lista.pop()`	usuwa ostatni element z listy i zwraca jego wartość
`lista.pop(i)`	usuwa i-ty element z listy i zwraca jego wartość
`lista.remove(x)`	usuwa pierwszy element z listy równy x
`lista.reverse()`	odwraca kolejność elementów znajdujących się na liście
`lista.sort()`	sortuje elementy znajdujące się na liście

Python - listy

- wstawienie elementu w dowolnym miejscu listy - metoda insert()

```
bierne = ["rezystor", "cewka", "kondensator"]
bierne.insert(0,"dioda")
print(bierne)
```

```
['dioda', 'rezystor', 'cewka', 'capacitor']
```

- usunięcie elementu listy o znanym indeksie - funkcja del

```
bierne = ["rezystor", "cewka", "kondensator"]
del bierne[2]
print(bierne)
```

```
['rezystor', 'cewka']
```

Python - listy

- metoda pop() usuwa ostatni element z listy i zwraca jego wartość

```
bierne = ["rezystor", "cewka", "kondensator"]
print(bierne)
element = bierne.pop()
print(f"Usunięty element to: {element}")
print(bierne)
```

```
['rezystor', 'cewka', 'kondensator']
Usunięty element to: kondensator
['rezystor', 'cewka']
```

Python - listy

- argumentem metody pop() może być indeks usuwanego elementu

```
bierne = ["rezystor", "cewka", "kondensator"]
print(bierne)
element = bierne.pop(1)
print(f"Usunięty element to: {element}")
print(bierne)
```

```
['rezystor', 'cewka', 'kondensator']
Usunięty element to: cewka
['rezystor', 'kondensator']
```

Python - listy

- usunięcie elementu na podstawie wartości - metoda remove()

```
bierne = ["rezystor", "cewka", "kondensator"]
print(bierne)
bierne.remove("rezystor")
print(bierne)
```

```
['rezystor', 'cewka', 'kondensator']
['cewka', 'kondensator']
```

- metoda remove() usuwa tylko pierwsze wystąpienie elementu

Python - listy

- usunięcie wszystkich elementów z listy - metoda clear()

```
bierne = ["rezystor", "cewka", "kondensator"]
bierne.clear()
print(bierne)
```

```
[]
```

- określenie wielkości listy - funkcja len()

```
bierne = ["rezystor", "cewka", "kondensator"]
ile = len(bierne)
print(f"Liczba elementów listy to: {ile}")
```

```
Liczba elementów listy to: 3
```

Python - pętla for i listy

- wyświetlenie elementów listy - w zmiennej element umieszczane są kolejne elementy listy bierne

```
bierne = ["rezystor", "cewka", "kondensator"]
for element in bierne:
    print(element)
```

```
rezystor
cewka
kondensator
```

- w powyższej metodzie nie musimy znać liczby elementów na liście

Python - pętla for i listy

- stosując funkcję enumerate() możemy otrzymać indeksy elementów listy

```
bierne = ["rezystor", "cewka", "kondensator"]
for indeks, element in enumerate(bierne):
    print(f"Indeks: {indeks}, Wartość: {element}")
```

```
Indeks: 0, Wartość: rezystor
Indeks: 1, Wartość: cewka
Indeks: 2, Wartość: kondensator
```

Python - ciągi tekstowe i listy

- metoda sort() - sortuje elementy na liście w kolejności alfabetycznej (elementy są sortowane trwale)

```
bierne = ["rezystor", "cewka", "kondensator"]
bierne.sort()
print(bierne)
```

```
['cewka', 'kondensator', 'rezystor']
```

- elementy można posortować w odwrotnej kolejności alfabetycznej

```
bierne = ["rezystor", "cewka", "kondensator"]
bierne.sort(reverse = True)
print(bierne)
```

```
['rezystor', 'kondensator', 'cewka']
```

Python - ciągi tekstowe i listy

- funkcja sorted() - sortuje elementy na liście w kolejności alfabetycznej (bez trwałego zapisywania)

```
bierne = ["rezystor", "cewka", "kondensator"]
print(bierne)
print(sorted(bierne))
print(bierne)
```

```
['rezystor', 'cewka', 'kondensator']
['cewka', 'kondensator', 'rezystor']
['rezystor', 'cewka', 'kondensator']
```

- funkcja sorted() może także mieć argument reverse = True (sortowanie w odwrotnej kolejności alfabetycznej)

Python - ciągi tekstowe i listy

- metoda split(separator, maxsplit) - dzieli ciąg tekstowy na fragmenty, zwane tokenami, na podstawie określonego separatora
- separator (opcjonalny) - znak lub ciąg znaków, który służy jako separator podziału tekstu, domyślnie separator to białe znaki (spacje, tabulatory, znaki nowej linii)
- maxsplit (opcjonalny) - maksymalna liczba podziałów, które zostaną wykonane

```
tekst = input("Podaj tekst: ")
lista = tekst.split()
print(lista)
```

```
Podaj tekst: Ela ma laptopa
['Ela', 'ma', 'laptopa']
```

Python - lista, indeksy elementów

- wyświetlenie listy z wykorzystaniem pętli for

```
imiona = ["Piotr", "Kuba", "Kacper"]
for imię in imiona:
    print(f"Witaj {imię}!")
```

```
Witaj Piotr!
Witaj Kuba!
Witaj Kacper!
```

- wyświetlenie wycinka listy z wykorzystaniem pętli for

```
imiona = ["Piotr", "Kuba", "Kacper"]
for imię in imiona[0:2]:
    print(f"Witaj {imię}!")
```

```
Witaj Piotr!
Witaj Kuba!
```

Python - lista, operacje (in)

- sprawdzenie, czy wartość znajduje się na liście - słowo kluczowe in

```
import random

liczby = [random.randint(0,9) for _ in range(10)]
print(liczby)

nr = int(input("Podaj liczbę: "))

if nr in liczby:
    print(f"Liczba {nr} znajduje się na liście")
else:
    print(f"Liczba {nr} nie znajduje się na liście")
```

```
[1, 4, 1, 8, 5, 4, 0, 9, 0, 1]
Podaj liczbę: 1
Liczba 1 znajduje się na liście
```

- podkreślenie po słowie kluczowym for jest często stosowane, gdy zmienna iteracyjna nie jest używana wewnątrz pętli

Python - lista, operacje (not in)

- sprawdzenie, czy wartość nie znajduje się na liście - słowo kluczowe not in

```
import random

liczby = [random.randint(0,9) for _ in range(10)]
print(liczby)

nr = int(input("Podaj liczbę: "))

if nr not in liczby:
    print(f"Liczba {nr} nie znajduje się na liście")
else:
    print(f"Liczba {nr} znajduje się na liście")
```

```
[2, 5, 8, 2, 0, 3, 9, 1, 7, 4]
Podaj liczbę: 6
Liczba 6 nie znajduje się na liście
```

Python - lista, operacje (czy lista jest pusta?)

- sprawdzenie, czy lista nie jest pusta

```
bierne = ["rezystor", "cewka", "kondensator"]

if bierne:
    print("Lista 'bierne' nie jest pusta")
else:
    print("Lista 'bierne' jest pusta")
```

```
Lista 'bierne' nie jest pusta
```

- jeśli nazwa listy pojawi się w poleceniu if, to zwracana jest wartość True, gdy lista zawiera przynajmniej jeden element
- w przypadku pustej listy zwracaną wartością będzie False

Python - lista, operacje (kopia listy)

- przypisanie nowej nazwy do istniejącej listy nie utworzy jej kopii

```
bierne = ["rezystor", "cewka"]

new_bierne = bierne
bierne.append("kondensator")

print(f"bierne =     {bierne}")
print(f"new_bierne = {new_bierne}")
```

```
bierne =     ['rezystor', 'cewka', 'kondensator']
new_bierne = ['rezystor', 'cewka', 'kondensator']
```

- obie zmienne prowadzą do tego samego obiektu w pamięci (listy)

Python - lista, operacje (kopia listy)

- utworzenie kopii listy wymaga zastosowania wycinka zawierającego całą listę początkową

```
bierne = ["rezystor", "cewka"]

new_bierne = bierne[:]
bierne.append("kondensator")

print(f"bierne =     {bierne}")
print(f"new_bierne = {new_bierne}")
```

```
bierne =     ['rezystor', 'cewka', 'kondensator']
new_bierne = ['rezystor', 'cewka']
```

- otrzymujemy dwa oddzielne obiekty w pamięci (dwie listy)

Python - lista, operacje (przenoszenie elementów)

- do przenoszenia elementów z jednej listy na drugą listę lepiej jest stosować pętlę while niż pętlę for (wewnątrz pętli for nie należy modyfikować listy, która jest przeglądana)

```
bierne = ["rezystor", "cewka", "kondensator"]
new_bierne = []

while bierne:
    element = bierne.pop()
    print(f"Przenoszony element: {element}")
    new_bierne.append(element)

print(new_bierne)
```

```
Przenoszony element: kondensator
Przenoszony element: cewka
Przenoszony element: rezystor
['kondensator', 'cewka', 'rezystor']
```

Python - lista, operacje (usunięcie elementów)

- usunięcie z listy wartości, które występują wiele razy

```
names = ["Jan", "Ola", "Jan", "Ela", "Ela", "Ula"]
print(names)

element = input("Podaj element do usunięcia: ")
while element in names:
    names.remove(element)

print(names)
```

```
['Jan', 'Ola', 'Jan', 'Ela', 'Ela', 'Ula']
Podaj element do usunięcia: Ela
['Jan', 'Ola', 'Jan', 'Ula']
```

Python - krotka, metody tworzenia

- użycie pary nawiasów zwykłych do oznaczenia pustej krotki

```
moja_krotka = ()
print(moja_krotka)
```

```
()
```

- użycie nawiasów zwykłych, oddzielenie elementów przecinkami

```
imiona = ("Piotr", "Kuba", "Kacper")
print(imiona)
```

```
('Piotr', 'Kuba', 'Kacper')
```

Python - krotka, metody tworzenia

- definicja krotki jednoelementowej wymaga dodania na końcu przecinka

```
imie1 = ("Ala",)
imie2 = "Ola",
print(imie1)
print(imie2)
```

```
('Ala',)
('Ola',)
```

- brak przecinka na końcu spowoduje utworzenie zwykłej zmiennej, a nie krotki

```
imie = ("Ala")
print(imie)
```

```
Ala
```

Python - krotka, metody tworzenia

- zastosowanie konstruktora bezargumentowego: tuple()

```
moja_krotka = tuple()
print(moja_krotka)
```

```
()
```

- zastosowanie konstruktora z argumentem w postaci obiektu iterowalnego: tuple(iterable)
- obiekt iterowalny może być sekwencją, kontenerem obsługującym iterację lub obiektem iteratora
- konstruktor buduje krotkę, której elementy są takie same i w tej samej kolejności, co elementy obiektu iterowalnego

Python - krotka, metody tworzenia

- inna krotka (lub lista) jako argument konstruktora: tuple(iterable)

```
imiona = ("Piotr", "Kuba", "Kacper")
names = tuple(imiona)
print(names)
```

```
('Piotr', 'Kuba', 'Kacper')
```

- inne obiekty iterowalne jako argumenty konstruktora: tuple(iterable)

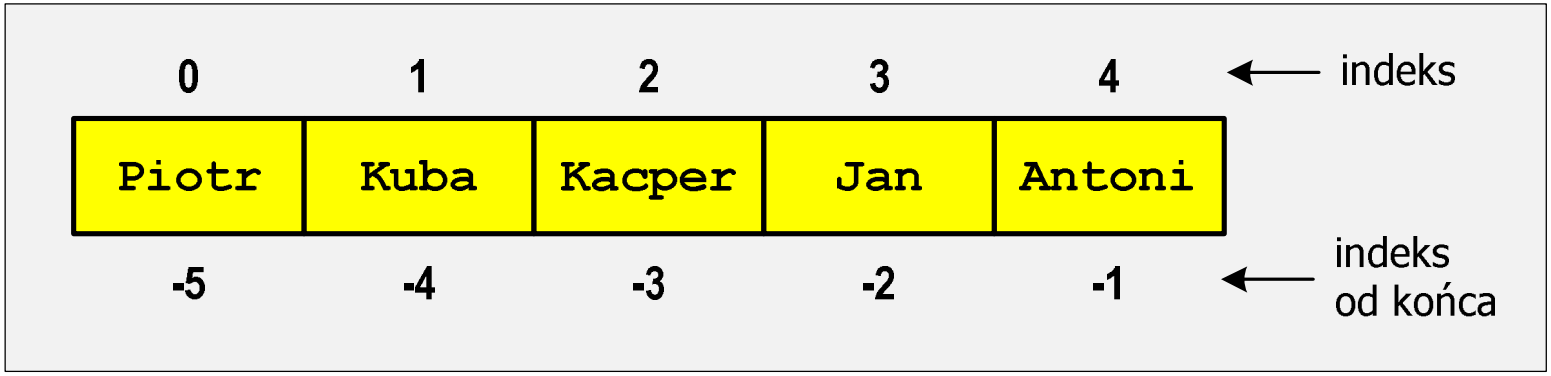
```
litery = tuple("ABCDE")
liczby = tuple((1, 2, 3, 4, 5))
print(litery)
print(liczby)
```

```
('A', 'B', 'C', 'D', 'E')
(1, 2, 3, 4, 5)
```

Python - krotka, indeksy elementów

- krotka, podobnie jak lista, jest uporządkowaną kolekcją, dostęp do jej dowolnego elementu można uzyskać przez podanie jego indeksu

```
imiona = ("Piotr", "Kuba", "Kacper", "Jan", "Antoni")
```



- odwołując się do elementów krotki podajemy nazwę krotki i w nawiasach kwadratowych numer elementu: nazwa_krotki[indeks]
- stosując dwukropek (:) można pracować na fragmentach krotki (wycinkach)

Python - lista, indeksy elementów

- przykłady odwołań do elementów krotki

```
imiona = ("Piotr", "Kuba", "Kacper", "Jan", "Antoni")

print(imiona[0])
print(imiona[1])
print(imiona[-1])

print(imiona[2:])
print(imiona[::2])
```

```
Piotr
Kuba
Antoni

('Kacper', 'Jan', 'Antoni')
('Piotr', 'Kacper', 'Antoni')
```

Python - krotka

- próba zmiany wartości elementu krotki zakończy się błędem

```
imiona = ("Piotr", "Kuba", "Kacper", "Jan", "Antoni")
imiona[0] = "Paweł"
```

```
Traceback (most recent call last):
  File "d:\MyApp.py", line 2, in <module>
    imiona[0] = "Paweł"
    ~~~~~~^^^
TypeError: 'tuple' object does not support item assignment
```

- można nadpisać krotkę nowymi wartościami

```
imiona = ("Piotr", "Kuba", "Kacper", "Jan", "Antoni")
print(imiona)

imiona = ("Marta", "Anna", "Grażyna", "Barbara")
print(imiona)
```

Koniec wykładu nr 2

Dziękuję za uwagę!