POLITECHNIKA ŚWIĘTOKRZYSKA

Wydział Elektrotechniki, Automatyki i Informatyki

Katedra Elektrotechniki Przemysłowej i Automatyki

Zakład Urządzeń i Systemów Automatyki

Metody Numeryczne w Technice

Wprowadzenie do języka Python.

Instrukcja laboratoryjna

Paweł Strączyński

2018

1 Podstawy języka Python

Język Python jest interpretowanym, obiektowym językiem wysokiego poziomu. Język cechuje rozbudowana biblioteka standardowa, dynamiczne typowanie oraz automatyczne przydzielanie pamięci. Charakterystyczny dla Pythona jest fakt, że wymaga on odpowiedniego formatowania kodu – wcięcia służą do wyróżniania poszczególnych bloków kodu.

1.1 Deklarowanie zmiennych, podstawowe operacje na liczbach i ciągach znakowych

Python jest językiem dynamicznie typowanym. Utworzenie zmiennej odbywa się podczas przypisania nazwie zmiennej wartości – 1.1. Tworzenie stringów odbywa się z wykorzystaniem jednego lub dwóch cudzysłowów.

Listing 1.1: Tworzenie zmiennych w języku Python

```
1 foo = 1
2 foo_real = 1.0
3 foobar = 10 + 2
4 bar = 'example␣text'
5 baz = "text2"
6 spam = 1.3 #Zmienna typu rzeczywistego
7 eggs = 1+2j #Liczba zespolona
```

Ponadto język Python cechuję silne typowanie – każde wyrażenie ma ustalony typ. Oznacza to, że konieczna jest jawna konwersja między poszczególnymi typami – 1.2. Funkcje do konwersji typów przedstawiono w tabeli 1.1 (język Python posiada 4 typy numeryczne). Aktualny typ

Tabela 1.1: Funkcje do konwersji typów

Funkcja	Opis
`int()`	Konwersja na liczbę całkowitą
`float()`	Konwersja na liczbę rzeczywistą
`long()`	Konwersja na liczbę typu long
`complex()`	Konwersja na liczbę zespoloną
`str()`	Konwersja na łańcuch znakowy

zmiennej można sprawdzić przy użyciu funkcji `type()`.

Listing 1.2: Silne typowanie w Pythonie

```
>> foo = 1
>> bar = '2'
>> foobar = foo + bar
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>foobar = foo + int(bar)
>>foobar
3
```

Zarówno na liczbach jak i stringach można wykonywać proste operacje. Podstawowe operatory przedstawiono w tabeli 1.2

Tabela 1.2: Podstawowe operacje

Operator	Opis
+	Dodawanie
-	Odejmowanie
*	Mnożenie
/	Dzielenie
%	Reszta z dzielenia (modulo)
**	Potęgowanie

Listing 1.3: Proste operacje w języku Python

```
>>foo = 1
>>bar = 2
>>foo + bar
3
>>qux = 'Lorem'
>>quux = 'ipsum'
>>qux + '␣' + quux
Lorem ipsum
>>spam = 1j
>>eggs = spam ** 2
>>eggs
(-1+0j)
>>corge = 'AB'
>>grault = corge * 4
>>grault
'ABABABAB'
```

Ciągi znakowe w Pythonie można formatować z wykorzystaniem operatora %. Stosując zapis:

```
format % wartość
```

Elementy formatujące pola format zostaną zastąpione elementami pola wartość. Jeżeli w polu format użyto jednego argumentu (zmiennej) to pole wartość jest pojedynczym elementem – tą zmienną. W przeciwnym razie jest krotką składającą się z odpowiednich elementów. Przykłady formatowania ciągów znakowych przedstawiono na listingu poniżej.

Listing 1.4: Formatowanie ciągów znakowych

```
>>foo = 1
>>bar = 'foo'
>>foobar = 'Wartosc␣zmiennej␣%s␣wynosi␣%d' % (bar,foo)
>>foobar
'Wartosc␣zmiennej␣foo␣wynosi␣1'
>>qux=124
>>quux = 'Liczba␣dziesietna␣%d␣to␣szestnastkowo␣%X' % (qux,qux)
>>quux
'Liczba␣dziesietna␣124␣to␣szestnastkowo␣7C'
>>spam = 'Numery:'
>>ham = 1.34
>>eggs = 2
>>spam = '%s␣%5.4f␣%02d' % (spam,ham,eggs)
>>spam
'Numery:␣1.3400␣02'
```

1.2 Listy, słowniki, tuple

Listy w Pythonie to nic innego jak zbiory rożnych elementów. Elementami listy mogą być wszystkie typy danych. Ponadto listy mogą być dowolnie zagnieżdżane. Na listingu 1.5 przedstawiono sposób definiowania list oraz dostępu do ich poszczególnych elementów. Warto zwrócić uwagę że użycie ujemnego indeksowania pozwala na odniesienie do elementu względem końca listy. Jak widać listy mogą zawierać również elementy rożnych typów.

Listing 1.5: Tworzenie list i dostęp do elementów

```
>>baz  =  ['Poniedzialek', 'Wtorek', 'Sroda', 'Czwartek', 'Piatek']
>>baz
['Poniedzialek', 'Wtorek', 'Sroda', 'Czwartek', 'Piatek']
>>baz[0]
'Poniedzialek'
```

```
>>baz[1]
'Wtorek'
>>baz[-2]
'Czwartek'
>>qux = ['Jeden', [1, 2], 3+0j]
>>qux[1][1] #Odczyt zagniezdzonego elementu listy
2
>>qux[1]
[1,2]
```

Do utworzenia listy w postaci ciągu arytmetycznego możemy użyć polecenia `range()`. Zakresu możemy użyć dwojako. Używając formy:

`range(n)`

Wtedy zostanie utworzona n-elementowa lista kolejnych liczb całkowitych rozpoczynając od zera. Polecenia `range()` możemy również użyć jako:

`range(x1,x2,n)`

gdzie odpowiednio:

- x1 – początkowy element listy,
- x2 – końcowy element listy,
- dx – krok (różnica między kolejnymi elementami).

Na listingu 1.6 przedstawiono przykłady tworzenia list będących ciągami arytmetycznymi oraz zawierających wiele tych samych elementów.

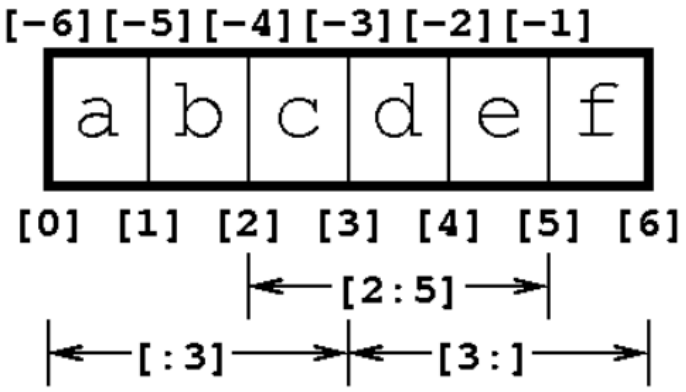
Listing 1.6: Tworzenie list cd.

```
>>list(range(5)) #range(5) Pythonie2
[0, 1, 2, 3, 4]
>>list(range(2,7))
[2, 3, 4, 5, 6]
>>list(range(2,7,2))
[2, 4, 6]
>>[1]*3
[1, 1, 1]
>>foo = [1, 2, 3]
>>foobar = foo * 2
>>foobar
[1, 2, 3, 1, 2, 3]
```

W Pythonie w łatwy sposób możemy używać „wycinków” list. Możemy dokonywać operacji bezpośrednio tylko na określonym zakresie listy ale również przepisywać fragmenty list i w ten sposób tworzyć nowe. Sposób dostępu do poszczególnych elementów list przedstawia rysunek 1.2.

Listing 1.7: Wyciniki list

```
>>foo=[10, 11, 12, 13, 14]
>>foo[0:2]
[10, 11]
>>foo[:2] #skrocony zapis j/w
[10, 11]
>>foo[-1]
14
>>bar = foo[1:3]
>>bar
[11, 12]
```



Rysunek 1.1: Dostęp do poszczególnych elementów list

Przydatnym narzędziem w języku Python jest tzw. wytwornik list. Wytworniki list tworzy się według następującego schematu:

```
[wyrażenie for zmienna in sekwencja]
```

Poniżej przedstawiono przykłady użycia wytworników list.

Listing 1.8: Przykłady użycia wytworników list

```
>>[(2*x+3) for x in range(5)]
[3, 5, 7, 9, 11]
>>[(x,x**3) for x in range(2,5)]
```

```
[(2, 8), (3, 27), (4, 64)]
>>foo=list(range(7))
>> [x for x in foo if x > 4] #Wytwornik listy z warunkiem
[5, 6]
```

Listy będące w Pythonie obiektami posiadają szereg metod umożliwiających operacje na nich. Metody umożliwiające operacje na listach przedstawiono w tabeli 1.3. Przykłady ich wykorzystania zostały zawarte na listingu 1.9.

Tabela 1.3: Metody umożliwiające operacje na listach

Metoda	Opis
`Append(x)`	Dodanie elementu `x` na końcu listy
`Count(x)`	Zliczenie wystąpienia wartości x w liście
`Index(x)`	Indeks pierwszego wystąpienia elementu x na liście
`Remove(x)`	Usunięcie pierwszego elementu o wartości x z listy
`Sort(x)`	Sortowanie listy
`Reverse(x)`	Odwrócenie kolejności elementów listy

Listing 1.9: Operacje na listach

```
>>spam = [10, 21, 23, 11, 24]
>>spam.append(11)
>>print(spam)
[10, 21, 23, 11, 24, 11]
>>spam.count(11)
2
>>spam.index(11)
3
>>spam.remove(11)
>>print(spam)
[10, 21, 23, 24, 11]
>>spam.sort()
>>print(spam)
[10, 11, 21, 23, 24]
>>spam.reverse()
>> print(spam)
[24, 23, 21, 11, 10]
```

Kolejnym typem danych w języku Python są słowniki. Dostępu do elementów słownika nie dokonujemy jak w przypadku list z użyciem indeksu lecz przez podanie klucza. Słownik zatem

to typ sekwencji który składa się z kluczy i przyporządkowanych do nich wartości. Tworzenie słowników i operacje na nich przedstawiono na listingu poniżej.

Listing 1.10: Tworzenie słowników i operacje na nich

```
>>foo = {'imie':'Nikola', 'nazwisko':'Tesla'}
>>foo['imie']
'Nikola'
>>foo['rok_ur'] = 1856 #Dodanie elementu do slownika
>> print(foo)
{'imie':'Nikola', 'nazwisko':'Tesla', 'rok_ur':1856}
>>bar ={} #Utworzenie pustego slownika
```

Przedstawione wyżej sekwencje danych (listy, słowniki) umożliwiają modyfikacje ich zawartości. Niemodyfikowalną sekwencją obiektów w Pythonie są krotki (tuple). Przykład użycia tupli przedstawiono na listingu 1.11

Listing 1.11: Użycie tupli w Pythonie

```
>>foobar = (1, 'dwa')
>>print(foobar)
(1, 'dwa')
>>foobar[0]
1
```

1.3 Instrukcje warunkowe, pętle

W Pythonie warunkowego wykonywania operacji dokonuję się z wykorzystaniem konstrukcji `if-(elif)-(else)`:

```
if wyrazenie_warunkowe:
        instrukcje
elif wyrazenie_warunkowe:
        instrukcje
else:
        instrukcje
```

Do wykonania porównania w języku Python wykorzystuje się dwa znaki równości (==). Wyrażenie „nie równe” w Pythonie oznacza się jako `!=`. Do łaczenia wyrażeń warunkowych używa się operatorów or oraz and które odpowiadają logicznym operacją boolowskim. Na listingu 1.12 przedstawiono przykładowe wyrażenie warunkowe.

Listing 1.12: Przykładowe wyrażenie warunkowe w Pythonie

```
spam = 1.4
if spam > 2:
        ham = 1
elif spam > 1 and spam < 2:
        ham = 2
else:
        ham = 3
```

W języku Python istnieje również trójargumentowe wyrażenie warunkowe:

```
wyrażenie1 if wyrażenie_warunkowe else wyrażenie2
```

Jest ono odpowiednikiem konstrukcji warunkowej z operatorem `?` z języka C.
Podobnie jak w innych językach programowania wysokiego poziomu, również w Pythonie możliwe jest iteracyjne (cykliczne) wykonywanie operacji z wykorzystaniem pętli. Pętla `for` wykorzystywana do iteracyjnego wykonywania instrukcji ma następującą postać:

```
for pozycja in sekwencja:
        instrukcje
```

Przy stosowaniu pętli for pomocna jest funkcja `range()` tworząca sekwencje ciągu liczb całkowitych. Przykładowe użycie pętli for przedstawiono na listingu poniżej.

Listing 1.13: Przykładowe użycie pętli for

```
for foo in 'PYTHON':
        print('%s.' % (foo))
foobar = []
for bar in range(5):
        foobar.append(bar)
```

Pętlą `while` wykonująca się do czasu gdy spełnione jest wyrażenie warunkowe ma postać:

```
while wyrażenie_warunkowe:
        instrukcje
```

Przykładowe użycie pętli `while` przedstawiono poniżej.

Listing 1.14: Przykładowe użycie pętli while

```
while(True): #Petla nieskonczona
        pass

```

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

Warto zwrócić uwagę że powyżej została użyta instrukcja pass. Nie robi ona nic konkretnego – używa się jej tam gdzie składnia wymaga obecności instrukcji.

1.4 Funkcje, klasy i obiekty

Język Python umożliwia tworzenie funkcji i korzystanie tym samym z funkcyjnego paradygmatu programowania. Funkcje tworzy się według następującego wzoru:

```
def nazwa_funkcji(argumenty):
        instrukcje
        return wyrazenie_zwracane_przez_funkcję
```

Ponadto możliwe jest również definiowanie funkcji anonimowych z wykorzystaniem operatora lambda:

```
Dodaj2 = lambda  argumenty: wyrażenie
```

Przykładowe funkcje przedstawiono na listingu 1.15

Listing 1.15: Tworzenie funkcji w języku Python

```
def Pierwiastek(foo,bar):
        return foo ** (1 / bar)

def Dodaj(spam,ham):
        eggs = spam + ham
        return eggs

def KodASCII(foobar):
        return ord(foobar)

Dodaj2 = lambda foo,bar : foo+bar
```

Python jest językiem zorientowanym obiektowo – umożliwia tworzenie klas, dziedziczenie ze stworzonych i wbudowanych klas oraz tworzenie obiektów. Klasy w języku Python tworzymy według następującego schematu:

```
class  nazwa_klasy(klasy z których dziedziczy):
        atrybuty
        metody
```

W Pythonie istnieją takie metody które maja specjalne znaczenie. Jedną z nich jest metodą o nazwie `__init__()` która jest wywoływana automatycznie po utworzeniu obiektu. Metoda ta często błędnie nazywana jest konstruktorem – w momencie jej wywoływania obiekt już istnieje. Przykład tworzenia klasy oraz jej instancji (obiektu) przedstawiono poniżej.

Listing 1.16: Tworzenie klasy i obiektu

```
class Szescian():
        def __init__(self, foo):
                self.bar = foo
        def PolePow(self):
                return 6 * (self.bar ** 2)
        def Objetosc(self):
                return self.bar ** 3

...

>>szescian1 = Szescian(2)
>>print(szescian1.PolePow())
24
```

W Pythonie pierwszy argument metody należącej do danej klasy (referencja do instancji tej klasy) nazwany jest `self`. Argument ten pełni tą samą rolę co np. słowo `this` w języku C++ jednak nie jest to zastrzeżone słowo a jedynie umowną koncepcją nazewnictwa.

2 Moduł NumPy

NumPy to podstawowy moduł umożliwiający wykonywanie zaawansowanych obliczeń matematycznych w języku Python. Umożliwia dokonywanie operacji na wektorach i macierzach.

2.1 Importowanie modułów w Pythonie

Aby móc korzystać z bibliotek należy je zaimportować. W Pythonie istnieją dwa sposoby importowania modułów. Pierwszy sposób to zaimportowanie modułu z wykorzystaniem instrukcji:

```
import moduł as skrócona_nazwa
```

Drugi sposób korzysta z konstrukcji:

```
from moduł import atrybuty, metody
```

Może się wydawać, że użycie polecenia import numpy oraz from numpy import * (gwiazdka oznacza że importujemy wszystko) da taki sam efekt. Istnieje jednak zasadnicza różnica która widoczna jest podczas dostępu do atrybutów i metod danego modułu. W przypadku użycia konstrukcji from moduł import atrybuty,metody importowanie zostaje wykonane do lokalnej przestrzeni nazw. W tym przypadku dostęp do atrybutów i metod jest możliwy bezpośrednio bez podawania modułu z którego pochodzą - 2.17.

Listing 2.17: Importowanie bibliotek w języku Python

```
>>import numpy as np
>>pi
Traceback (most recent call last):

  File "<ipython-input-2-68f7b1e53523>", line 1, in <module>
    pi

NameError: name 'pi' is not defined
>>np.pi
3.141592653589793
>>from numpy import *
>>pi
3.141592653589793
```

2.2 Tworzenie macierzy

W module Numpy wyróżnić można dwa podstawowe typy macierzowe:

- matrix,
- array.

Najistotniejsza różnica między typami macierzowymi w module NumPy to sposób wykonywania operacji mnożenia.

1. 1. W przypadku tablic (array) użycie operatora * powoduje mnożenie elementowe macierzy – odpowiednik operacji .* w MATLAB-ie. Aby wykonać mnożenie macierzy (a nie ich elementów) należy użyć metody dot().

2. Używając obiektów klasy matrix operator `*` powoduje mnożenie macierzy. Mnożenie elementów macierzy możliwe jest z wykorzystaniem metody `multiply()`

Przykłady tworzenia tablic i macierzy oraz wspomniane wyżej różnice w wykonywaniu operacji mnożenia przedstawiono na listingu 2.18. Należy zwrócić uwagę, że indeksowanie elementów zaczyna się od zera (nie jak w MATLAB-ie od 1).

Listing 2.18: Rożnicę między typami macierzowymi w NumPy

```
>>A = np.array([[1, 2], [3, 4]])
>>B = np.array([[1, 1], [2, 2]])
>>A * B
array([[1, 2],
       [6, 8]])
>>np.dot(A,B)
array([[ 5,  5],
       [11, 11]])
>>A= np.matrix([[1, 2], [3, 4]])
>>B = np.matrix([[ 1, 1], [2, 2]])
>>A * B
matrix([[ 5,  5],
        [11, 11]])
>>np.multiply(A,B)
matrix([[1, 2],
        [6, 8]])
>>A[0,0]
1
```

Aby utworzyć macierz w postaci wektora stanowiącego ciąg arytmetyczny można wykorzystać następujące metody:

`arange(n)`

`arange(x1,x2,k)`

`linspace(x1,x2,n)`

gdzie:

`x1` - początkowy element tablicy,

`x2` – końcowy element tablicy,

`k` – krok – różnica między kolejnymi elementami ciągu,

`n` – liczba elementów wektorach.

Aby uzyskać wektor elemntów o rozkładzie w skali logarytmicznej należy użyć metody `logspace()`.

Jej użycie jest analogiczne do metody `linspace()`. Ponadto moduł NumPy zawiera szereg metod pozwalających szybkie tworzenie „typowych macierzy" takich jak:

- ones - macierz jedynek,
- zeros - macierz zer,
- eye - macierz jednostkowa.

Ich użycie przedstawiono na listingu poniżej.

Listing 2.19: Tworzenie macierzy - NumPy

```
>>Z = zeros((2,4),uint8)
>>print(Z)
array([[0, 0, 0, 0],
       [0, 0, 0, 0]], dtype=uint8)
>>I = eye(3)
>>print(I)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>A = np.arange(9)
>>A
array([0, 1, 2, 3, 4, 5, 6, 7, 8])
>>A.shape = (3,3) #Formowanie ksztaltu macierzy
>>A
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>A.T   #Transponowanie tablicy
array([[0, 3, 6],
       [1, 4, 7],
       [2, 5, 8]])
>>A.min() #Wartosc minimalna w tablicy
0
>>A.size #Ilosc elementow macierzy
9
```

Powyżej przedstawiono przykładowe atrybuty i metody obiektu typu array. Szczegółową dokumentację klasy ndarray znaleźć można w SciPy.org.

2.3 Podstawowe operacje matematyczne, operacje porównania

Operacje matematyczne na tablicach możliwe są z wykorzystaniem operatorów lub specjalnych funkcji. Przykładowe użycie przedstawiono na listingu poniżej. Funkcje matematyczne przedstawiono w 2.1.

Listing 2.20: Operacje matematyczne na macierzach - NumPy

```
>>A = np.array((1, 2, 3))
>>B = np.array((2, 3, 5))
>>A+B
array([3, 5, 8])
>>np.add(A,B)
array([3, 5, 8])
>>np.power(A,2)
array([1, 4, 9])
```

Biblioteka NumPy umożliwia również porównywanie zawartości tablic oraz dokonywania operacji logicznych. Przykładowe operacje przedstawiono na listingu poniżej. Podobnie jak w przypadku operacji matematycznych mozliwe jest wykorzystanie operatorów jaki i specjalnych funkcji.

Listing 2.21: Operacje porównania i logiczne - przykłady

```
>>A = np.array((1, 2, 3, 4))
>>B = np.array((1, 3, 5, 4))
>>A == B
array([ True, False, False,  True], dtype=bool)
>>np.equal(A,B)
array([ True, False, False,  True], dtype=bool)
>>np.any(A > 6) #Prawda, jezeli ktorys z elementow spelnia warunek
False
>>np.any(A > 3)
True
>>np.all( A > 0 ) #Prawda, jezeli wszystkie z elementow spelniaja warunek
True
>> np.all( A > 0 ) and np.all( A > 3 )
False
```

Tabela 2.1: Operatory porównania i logiczne

Operator	Funkcja
`A == B`	`equal(A,B)`
`A >= B`	`greater_equal(A,B)`
`A <= B`	`less_equal(A,B)`
`A > B`	`greater(A,B)`
`A < B`	`less(A,B)`
`A != B`	`not_equal(A,B)`
`wyrażenie1 and wyrażenie2`	`logical_and(wyrażenie1,wyrażenie2)`
`wyrażenie1 or wyrażenie2`	`logical_or(wyrażenie1,wyrażenie2)`
`not wyrażenie`	`logical_not(wyrażenie)`
	`logical_xor(wyrażenie)`

2.4 Funkcje trygonometryczne, statystyczne oraz inne funkcje matematyczne

Biblioteka NumPy zawiera szereg funkcji matematycznych. Dostępne są m.in. funkcje trygonometryczne, funkcje umożliwiające proste działania statystyczne oraz szereg innych funkcji. Ważniejsze funkcje przedstawiono w tabeli 2.2. Przykładowe wykorzystanie wyżej przedstawionych funkcji pokazano na listingu poniżej.

Listing 2.22: Funkcje matematyczne – biblioteka NumPy

```
>>foo = sin(2) + exp(3)
>>foo
20.994834350013349
>>bar = ceil(foo)
>>bar
21.0
>>negative(bar)
-21.0
>>fmod(2.3, 1.4)
0.89999999999999991
>>foobar = 3+2j
>>conjugate(foobar)
(3-2j)
```

Tabela 2.2: Funkcje matematyczne - NumPy

Operator	Funkcja
`sin(x)`	sinus kąta `x`
`sinh(x)`	sinus hiperboliczny kąta `x`
`cos(x)`	cosinus kąta `x`
`cosh(x)`	cosinus hiperboliczny kąta `x`
`arcsin(x)`	arcus sinus `x`
`arcsinh(x)`	arcus sinus hiperboliczny `x`
`arccos(x)`	arcus cosinus `x`
`arccosh(x)`	arcus cosinus hiperboliczny `x`
`arctan(x)`	arcus tangens `x`
`arctanh(x)`	arcus tengens hiperboliczny `x`
`arctan2(x,y)`	zwraca kąt którego tangens jest ilorazem `x` oraz `y`
`exp(x)`	funkcja ekspotencjalna e^x
`log(x)`	logarytm naturalny liczby `x`
`log10(x)`	logarytm dziesiętny liczby `x`
`sqrt(x)`	pierwiastek kwadratowy z `x`
`absolute(x)`	wartość bezwzględna z `x`
`conjugate(x)`	sprzężenie liczby zespolonej `x`
`negative(x)`	liczba odwrotna do `x`
`ceil(x)`	Zaokrąglenie liczby `x` w górę
`floor(x)`	Zaokrąglenie liczby `x` w dół
`fabs(x)`	Wartość bezwzględna liczby zmiennoprzecinkowej `x`
`hypot(x)`	Funkcja odległości euklidesowej `x` i `y` $\sqrt{x^2 + y^2}$
`fmod(x)`	Reszta z dzielenia liczby rzeczywistych `x` i `y`
`maximum(x,y)`	Wartość maksymalna z liczb `x` i `y`
`minimum(x,y)`	Wartość minimalna z liczb `x` i `y`

3 Biblioteka Matplotlib

Matplotlib to biblioteka do tworzenia wykresów stworzona dla języka Python i modułu NumPy. API biblioteki zostało zaprojektowane w ten sposób aby jak najbardziej przypominało to z MATLAB-a. Szczegółowa dokumentacja biblioteki dostępna jest na stronie projektu. W tabeli 3.1 przedstawiono podstawowe funkcje umożliwiające tworzenie wykresów 2D. Przy-

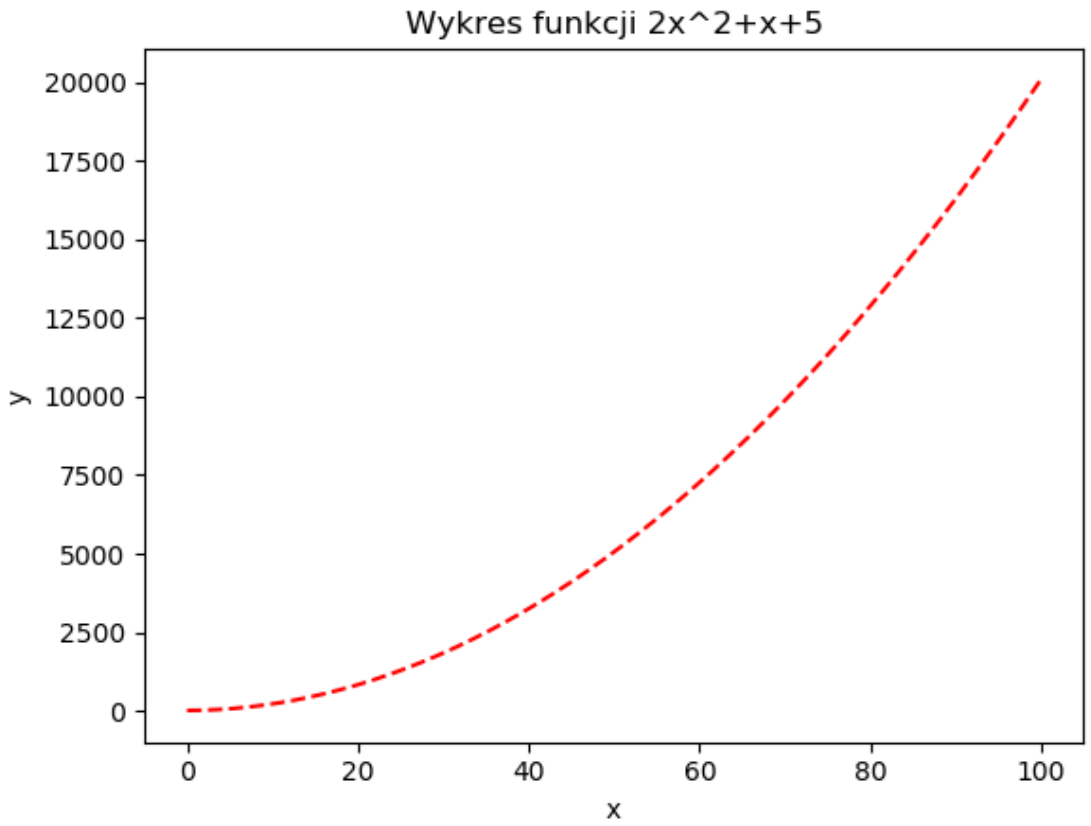
Tabela 3.1: Podstawowe funkcje biblioteki Matplotlib do rysowania wykresów

Funkcja	Opis
`figure(x)`	otwarcie okna rysunkowego x
`plot(x,y,styl_linii)`	stworzenie wykresu 2D funkcji y=f(x)
`show()`	wyświetlenie obrazu
`title('tekst')`	dodanie tytułu do wykresu
`xlabel('tekst')`	dodanie opisu osi x
`ylabel('tekst')`	dodanie opisu osi y
`axis([xmin, xmax, ymin, ymax])`	ograniczenie osi wykresu w podanym zakresie
`grid(True/False)`	włączenie/wyłączenie siatki
`savefig('nazwa_pliku')`	zapis rysunku do pliku
`legend('tekst')`	dodanie legendy do wykresu
`hold(True/False)`	zablokowanie okna rysunkowego
`subplot(l.w,l.k,nr_okna)`	utworzenie podwykresu
`[X,Y] =meshgrid(x,y)`	przekształca parę wektorów x,y w parę macierzy X,Y
`contour(x,y,z)`	stworzenie wykresy konturowego funkcji y=f(x,y)
`contourf(x,y,z)`	wypełniony wykres konturowy funkcji y=f(x,y)
`surf(x,y,z)`	stworzenie wykresu 3D funkcji y=f(x,y)
`specgram(x)`	stworzenie spektogram x
`imshow(A)`	wyświetlenie obrazu z macierzy A

kład prostych wykresów stworzonych z wykorzystaniem biblioteki przedstawiono poniżej.

Listing 3.23: Wykres funkcji $2x^2 + x + 5$ - Matplotlib

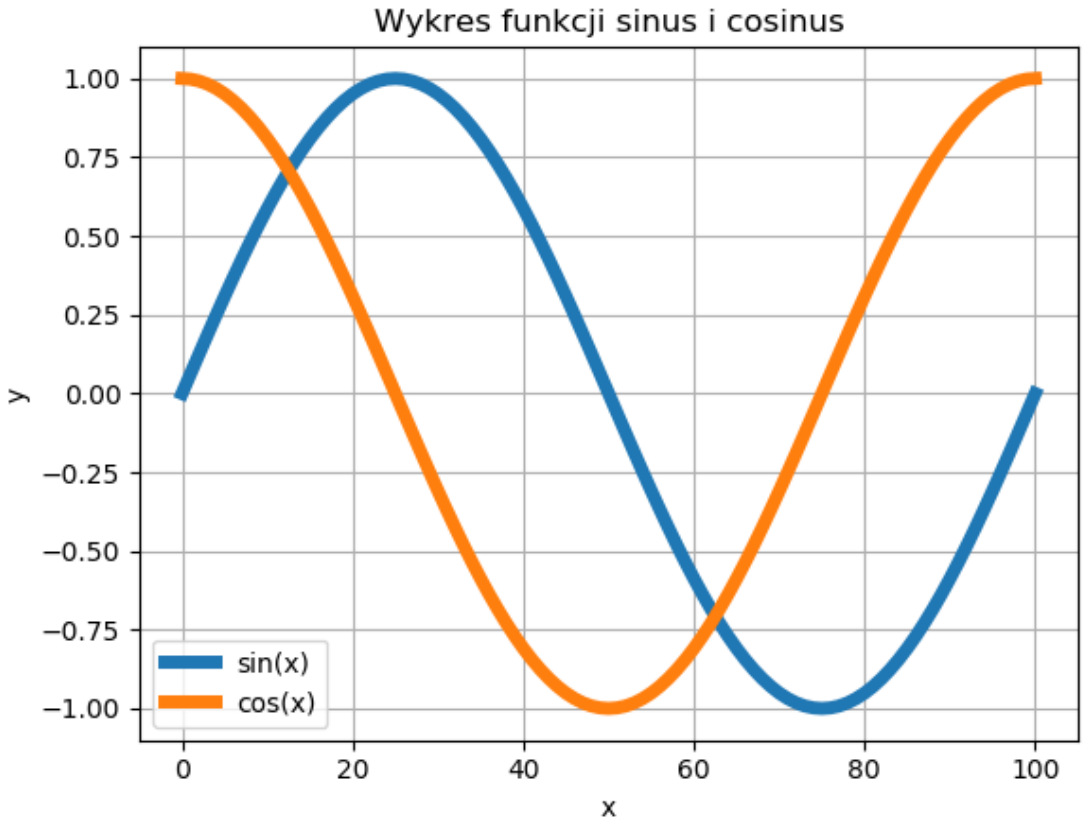
```
1 import matplotlib.pylab as plt
2 x=plt.arange(0, 100, 0.1) #Stworzenie wektora wartosci x
3 y=2 * (x ** 2) + x + 5 #Wyznaczenie wartosci y
4 plt.plot(x,y,'r--') #Narysowanie wykresu y=f(x)
5 plt.title('Wykres funkcji 2x^2+x+5')
6 plt.xlabel('x')
7 plt.ylabel('y')
8 plt.savefig('wykres.png') #Zapis wykresu do pliku
9 plt.show()
```



Rysunek 3.1: Wykres funkcji $2x^2 + x + 5$ - Matplotlib

Listing 3.24: Wykres funkcji sinus i cosinus

```
from scipy import *
from pylab import *
x = r_[0:101]
y01 = sin( 2 * pi * x  /100)
y02 = cos( 2 * pi * x / 100)
plot(x, y01, linewidth = 5.0)
hold(True)
plot(x, y02, linewidth = 5.0)
xlabel('x'); ylabel('y')
title('Wykres funkcji sinus i cosinus')
legend(('sin(x)','cos(x)'))
grid(True)
show()
```



Rysunek 3.2: Wykres funkcji sinus i cosinus

4 Zadania do wykonania

Zadanie 1 Zapoznać się z podstawami języka Python. Uruchomić i zmodyfikować przykłady z instrukcji

Zadanie 2 Zapoznać się z modułem NumPy. Uruchomić i zmodyfikować przykłady z instrukcji.

Zadanie 3 Zapoznać się z biblioteką Matplotlib. Uruchomić przykłady z instrukcji. Utworzyć własne wykresy.

Zadanie 4 Utworzyć funkcję wyznaczającą wartość liczby π według zależności:

$$\sum_{k=0}^{\infty} \frac{-1^k}{2k+1} = \frac{\pi}{4}$$

Zadanie 5 Utworzyć skrypt który:

- tworzy wektor wierszowy `v0` o 10 elementach którego liczba początkowa to 0 a końcowa to 5,
- tworzy wektor `v1` powstały przez dodanie liczby 5 do wektora `v0`,
- tworzy macierz `A` (2x10) przez konkatenacje wektorów `v0` oraz `v1`,

Zadanie 6 Wygenerować wykres funkcji:

$$y_1(t) = \sin(t)e^{-0,08t}$$

gdzie $t \in < 0, 12 >$ z krokiem równym $0, 01$.
Wykres powinien posiadać tutuł, opisy osi, oraz legendę.