

Przygotowanie do zajęć z laboratorium "Metody numeryczne"

1. Zainstaluj najnowszą wersję Python

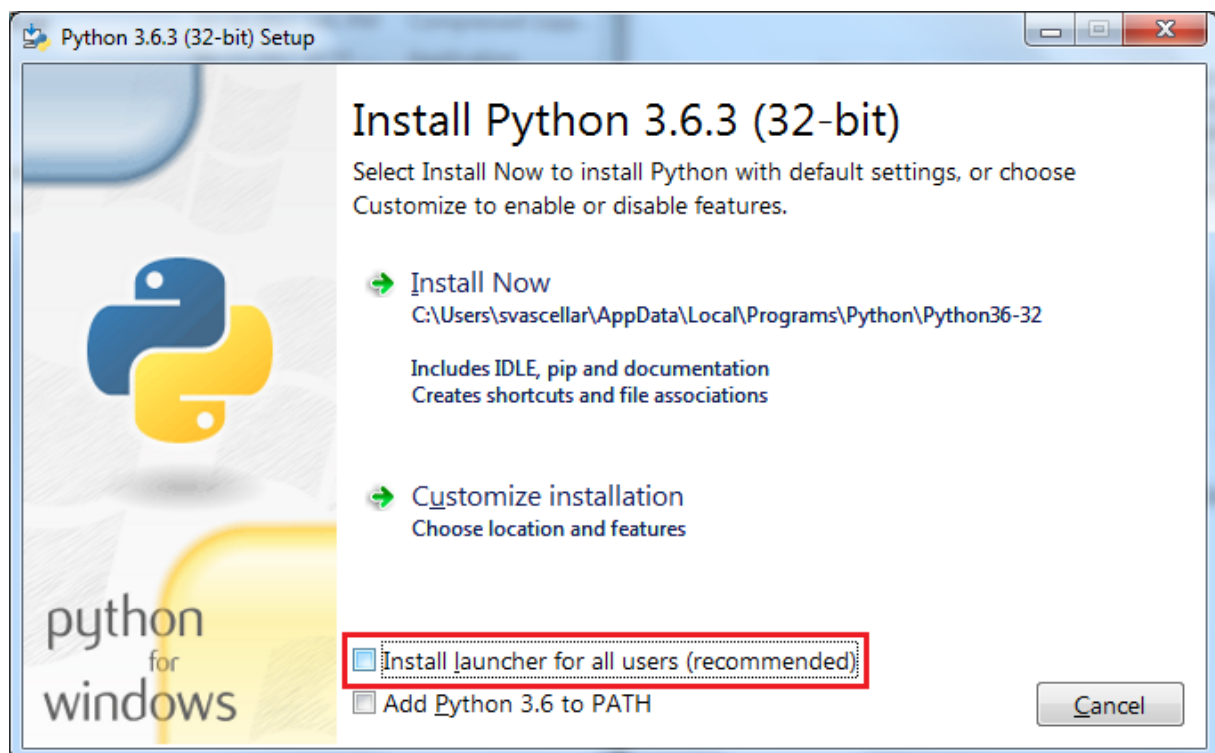
Python jest niezwykle popularnym językiem programowania, który może służyć do tworzenia stron internetowych, gier, oprogramowania naukowego, grafiki i wielu, wielu innych rzeczy.

Początki Pythona sięgają późnych lat 80., a jego głównym założeniem jest czytelność, dzięki której kod jest łatwy do zrozumienia przez ludzi (a nie tylko przez komputery!).

Na początku sprawdź, czy twój komputer działa na 32-bitowej czy 64-bitowej wersji Windowsa, poprzez wciśnięcie kombinacji przycisków Windows + Pause/Break, które otworzą Właściwości systemu i sprawdzenie linii "Typ systemu". Możesz ściągnąć Pythona dla Windowsa ze strony <https://www.python.org/downloads/windows/>.

Kliknij w link "Latest Python 3 Release - Python x.x.x". Jeżeli twój komputer pracuje na 64-bitowej wersji Windowsa, ściągnij Windows x86-64 executable installer. W innym wypadku ściągnij Windows x86 executable installer. Po ściągnięciu instalatora, powinnaś go uruchomić (klikając dwukrotnie w niego) i postępować według wyświetlanych instrukcji.

Podczas instalacji zauważysz ekran oznaczony jako "Setup". Upewnij się, że zaznaczyłaś checkbox "Add Python 3.6 to PATH" oraz kliknęłaś "Install Now",



2. Zainstaluj środowisko programistyczne Spyder

Spyder to klasyczne środowisko programistyczne, swoją budową i działaniem mocno wzorowane na RStudio, flagowym narzędziu programistów R. Posiada sporo opcji konfiguracyjnych (np. predefiniowany wygląd przypominający Matlaba itp.), dzięki czemu odnajdą się w nim osoby posiadające doświadczenie z innymi językami programowania. Uzupełnia on dystrybucję Anacondy o narzędzie dla osób wolących tradycyjny sposób programowania.

The screenshot shows the Spyder Python IDE interface. The editor window displays a script that imports the `rosen` function and the `differential_evolution` algorithm from `scipy.optimize`. It defines a set of bounds for a 5-dimensional problem and runs the optimization with `disp=True`. The console window on the right shows the iterative output of the differential evolution algorithm, with the function value `f(x)` decreasing from approximately $1.04771e-30$ to 0 over 56 steps. At the bottom of the console, the final result is displayed as an array of ones.

```
1 from scipy.optimize import rosen, differential_evolution
2
3 bounds = [(0,2), (0, 2), (0, 2), (0, 2), (0, 2)]
4
5 result = differential_evolution(rosen, bounds, disp=True)
6
7 result.x, result.fun
```

```
differential_evolution step 520: f(x)= 1.04771e-30
differential_evolution step 521: f(x)= 1.04771e-30
differential_evolution step 522: f(x)= 1.04771e-30
differential_evolution step 523: f(x)= 1.04771e-30
differential_evolution step 524: f(x)= 1.04771e-30
differential_evolution step 525: f(x)= 1.04771e-30
differential_evolution step 526: f(x)= 1.04771e-30
differential_evolution step 527: f(x)= 0
differential_evolution step 528: f(x)= 0
differential_evolution step 529: f(x)= 0
differential_evolution step 530: f(x)= 0
differential_evolution step 531: f(x)= 0
differential_evolution step 532: f(x)= 0
differential_evolution step 533: f(x)= 0
differential_evolution step 534: f(x)= 0
differential_evolution step 535: f(x)= 0
differential_evolution step 536: f(x)= 0
differential_evolution step 537: f(x)= 0
differential_evolution step 538: f(x)= 0
differential_evolution step 539: f(x)= 0
differential_evolution step 540: f(x)= 0
differential_evolution step 541: f(x)= 0
differential_evolution step 542: f(x)= 0
differential_evolution step 543: f(x)= 0
differential_evolution step 544: f(x)= 0
differential_evolution step 545: f(x)= 0
differential_evolution step 546: f(x)= 0
differential_evolution step 547: f(x)= 0
differential_evolution step 548: f(x)= 0
differential_evolution step 549: f(x)= 0
differential_evolution step 550: f(x)= 0
differential_evolution step 551: f(x)= 0
differential_evolution step 552: f(x)= 0
differential_evolution step 553: f(x)= 0
differential_evolution step 554: f(x)= 0
differential_evolution step 555: f(x)= 0
differential_evolution step 556: f(x)= 0

In [5]: result.x
Out[5]: array([ 1.,  1.,  1.,  1.,  1.])
```

3. Zaktualizuj biblioteki NumPy i SciPy

Pakiet NumPy

Moduł NumPy jest podstawowym zestawem narzędzi dla języka Python umożliwiającym zaawansowane obliczenia matematyczne, w szczególności do zastosowań naukowych (tzw. obliczenia numeryczne, jak mnożenie i dodawanie macierzy, diagonalizacja czy odwrócenie, całkowanie, rozwiązywanie równań, itd.). Daje on nam do dyspozycji specjalizowane typy danych, operacje i funkcje, których nie ma w typowej instalacji Pythona.

Pakiet SciPy

Pakiet SciPy to zestaw algorytmów numerycznych i przydatnych funkcji, dzięki którym Python może zastąpić dedykowane środowiska do obliczeń numerycznych (Matlab, GNU Octave, Scilab). Pakiet wykorzystuje NumPy do tworzenia wielowymiarowych struktur danych i operacji na nich. Pakiet podzielony jest na podmoduły, z których większość odpowiada tematycznie wybranej dziedzinie metod numerycznych.

Zainstaluj niezbędne dodatkowe biblioteki.

Operacje arytmetyczne

Podstawowe operacje (+, -, *, /) działają tak jak można się spodziewać.

```
In[1]:2+2
```

```
Out[1]: 4
```

Potęgowanie oznaczane jest znakiem ** , zatem 2 do potęgi trzeciej (2^3) wygląda tak:

```
In[2]:2**3
```

```
Out[2]: 8
```

Inne operatory arytmetyczne to % — reszta z dzielenia albo modulo: 3 dzielone przez dwa to 1 (bo dwójka mieści się w trójce całkowicie jeden raz) i reszta 1:

```
In[3]: 3%2
```

```
Out[3]: 1
```

W bardziej skomplikowanych obliczeniach można posługiwać się podstawowym zestawem funkcji (tak zwane funkcje wbudowane):

- abs wartość bezwzględna
- divmod(a,b) zwraca wynik dzielenia całkowitoliczbowego i resztę z dzielenia jako parę liczb

```
In[4]:divmod(3,2)
```

```
Out[4]: (1, 1)
```

- float konwertuje liczbę do postaci zmiennoprzecinkowej

```
In[5]: float(3)
```

```
Out[5]:3.0
```

- round(x[,n]) zaokrągla liczbę do n miejsc po przecinku:

```
In[6]: round(3.1415)
```

```
Out[6]:3
```

```
In[7]: round(3.1415,1)
```

```
Out[7]: 3.1
```

Użycie innych funkcji matematycznych wymaga specjalnych zabiegów: podpowiedzenia programowi, że będziemy chcieli z nich korzystać:

```
import math
```

mówi, że zechcemy używać modułu math zawierającego różne funkcje i stałe matematyczne. Po wydaniu tego polecenia możemy napisać:

```
In[8]: math.cos(math.pi / 3)
Out[8]: 0.5000000000000001
In[9]: math.log(1024, 2)
Out[9]: 10.0 10.0
In[10]: math.pi
Out[10]: 3.1415926535897931
```

Dostępne stają się funkcje trygonometryczne, hiperboliczne, logarytmiczne i wykładnicze, funkcje specjalne oraz stałe: π i e

```
In[10]: math.e
Out[10]: 2.7182818284590451
```

Z całej biblioteki można zaimportować tylko pojedynczą funkcję

```
from decimal import Decimal
```

Obliczenia

W prostych obliczeniach można korzystać z pythona jak z podręcznego kalkulator: wyniki obliczeń podawane będą na bieżąco. Można też wyniki zapamiętywać (coś jak pamięci kalkulatora). Z tym, że poszczególne pamięci możemy nazywać według naszego uznania:

```
In[1]: r=5
In[2]: obwod=2*math.pi*r
In[3]: pole=math.pi*r**2
```

Podanie w linii nazwy pamięci (zmiennej) powoduje wydrukowanie jej wartości:

```
In[4]: r
Out[4]: 5
In[5]: r, pole
Out[5]: (5, 78.539816339744831)
In[6]: r, pole, obwod
Out[6]: (5, 78.539816339744831, 31.415926535897931)
```

Bloki instrukcji

Cechą odróżniającą język python od innych języków (Na przykład C) jest sposób wyróżniania bloków instrukcji. W języku C używa się do tego nawiasów klamrowych {}.

W pythonie używa się „wcięć”.

Potrzeba korzystania z bloków instrukcji pojawia się, między innymi:

1. podczas wykonywania instrukcji warunkowych,
2. przy tworzeniu pętli,
3. podczas definiowania funkcji.

Na przykład:

```
In[7]:for i in range(1,10):  
    print(i)
```

```
In[8]:if a > b :  
    print( 'A wieksze od B')  
else :  
    print('B wieksze od A')
```

Poniżej przykład programu rozwiązującego równanie kwadratowe:

```
from math import sqrt  
a = input('a=')  
b = input('b=')  
c = input('c=')  
a = float(a)  
b = float(b)  
c = float(c)  
delta = b*b - 4*a*c  
if delta > 0:  
    x1 = (-b - sqrt(delta))/(2*a)  
    x2 = (-b + sqrt(delta))/(2*a)  
    print('x1=',x1)  
    print('x2=',x2)  
elif delta == 0:  
    x = -b/2/a  
    print('x=', x)  
else :  
    print('Nie ma pierwiastkow rzeczywistych!')
```

The screenshot shows the Spyder Python IDE interface. The editor window displays a Python script named `trojmian.py` that solves a quadratic equation $ax^2 + bx + c = 0$. The script uses `input()` to get coefficients `a`, `b`, and `c`, converts them to floats, calculates the discriminant $\Delta = b^2 - 4ac$, and uses conditional logic to find the roots or determine if there are no real roots. The console window shows the execution of the script with three different sets of inputs: `a=1, b=2, c=1` (resulting in two real roots), `a=1, b=1, c=1` (resulting in no real roots), and `a=-1, b=-2, c=1` (resulting in two real roots).

```
1 from math import sqrt
2 a = input('a=')
3 b = input('b=')
4 c = input('c=')
5 a = float(a)
6 b = float(b)
7 c = float(c)
8 delta = b*b - 4*a*c
9 if delta > 0:
10     x1 = (-b - sqrt(delta))/(2*a)
11     x2 = (-b + sqrt(delta))/(2*a)
12     print('x1=', x1)
13     print('x2=', x2)
14 elif delta == 0:
15     x = -b/2/a
16     print('x=', x)
17 else:
18     print('Nie ma pierwiastkow rzeczywistych!')
19
```

Console output:

```
In [33]: runfile('C:/Users/u46/Desktop/dif evol/spline/
trojmian.py', wdir='C:/Users/u46/Desktop/dif evol/spline')
a=1
b=2
c=1
x= -1.0

In [34]: runfile('C:/Users/u46/Desktop/dif evol/spline/
trojmian.py', wdir='C:/Users/u46/Desktop/dif evol/spline')
a=1
b=1
c=1
Nie ma pierwiastkow rzeczywistych!

In [35]: runfile('C:/Users/u46/Desktop/dif evol/spline/
trojmian.py', wdir='C:/Users/u46/Desktop/dif evol/spline')
a=1
b=2
c=-1
x1= -2.414213562373095
x2= 0.41421356237309515

In [36]:
```

Uwagi:

1. Program można zapisać np. pod nazwą: `trojmian.py`, a później uruchomić w terminalu wypisując `python trojmian.py` albo uruchamiamy spyder, otwieramy plik źródłowy File → Open i uruchamiamy Run → Run module albo naciskając klawisz F5.
2. Funkcja `float` została użyta aby skonwertować tekst (ciąg znaków) do liczby. Informacje wczytywane z terminala funkcją `input` są typu tekstowego!

Instrukcje warunkowe

Instrukcja warunkowa używana jest podczas programowania do wprowadzenia rozgałęzienia w zależności od wartości jakiegoś parametru czy spełnienia jakiegoś warunku. Typowy przykład konieczności wprowadzenia instrukcji warunkowej to program wyliczania rzeczywistych miejsc zerowych trójkątnego kwadratowego. W zależności od wartości parametru Δ albo wyliczamy jeden pierwiastek rzeczywisty, albo dwa, albo musimy stwierdzić, że pierwiastków rzeczywistych nie ma.

Fragment programu wyliczającego pierwiastki przedstawiono w poprzednim rozdziale. (Tam gdzie się kończy wcięcie — kończy się zakres działania odpowiedniego fragmentu warunku.)

Pętle

Pętla to taka konstrukcja programistyczna, która nakazuje wykonanie jakiejś czynności:

- kilka razy

```
for i in range(3,12) :  
    print( i)
```

- tak długo aż jakiś warunek zostanie spełniony.

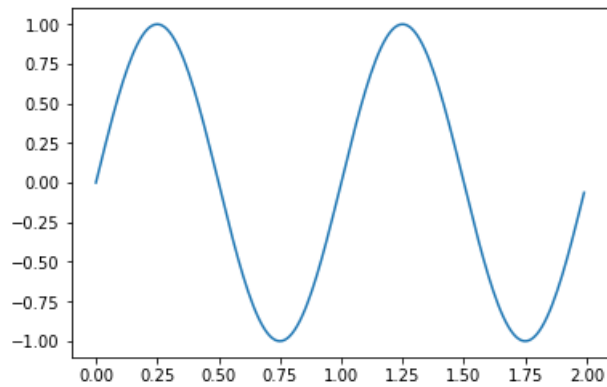
```
i=3  
while i < 12 :  
    print( i)  
    i = i+1
```

W obu powyższych przypadkach efekt powinien być taki:

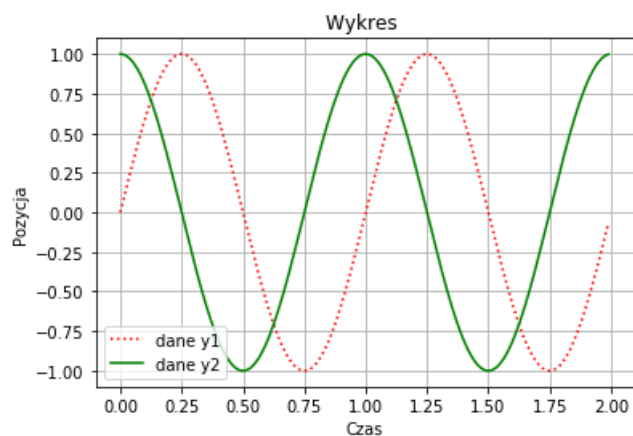
```
3  
4  
5  
6  
7  
8  
9  
10  
11
```

Rysowanie funkcji

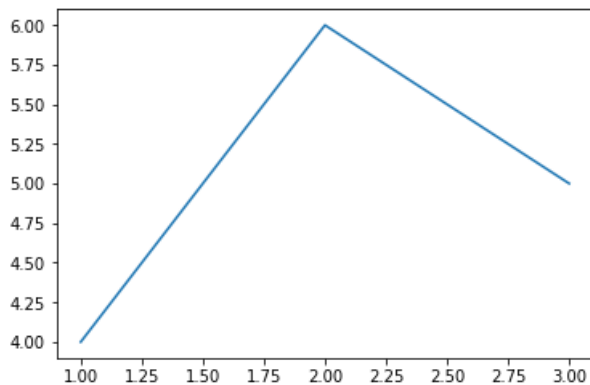
```
In [36]: import numpy as np
...: import matplotlib.pyplot as plt
...: x = np.arange(0.0, 2.0, 0.01)
...: y = np.sin(2.0*np.pi*x)
...: plt.plot(x,y)
...: plt.show()
...:
...:
```



```
In [3]: import numpy as np
...: import matplotlib.pyplot as plt
...: x = np.arange(0.0, 2.0, 0.01)
...: y1 = np.sin(2.0*np.pi*x)
...: y2 = np.cos(2.0*np.pi*x)
...: plt.plot(x,y1,'r:',x,y2,'g')
...: plt.legend(('dane y1','dane y2'))
...: plt.xlabel('Czas')
...: plt.ylabel('Pozycja')
...: plt.title('Wykres ')
...: plt.grid(True)
...: plt.show()
...:
...:
```




```
In [4]: import matplotlib.pyplot as plt
...: x = [1,2,3]
...: y = [4,6,5]
...: plt.plot(x,y)
...: plt.show()
...:
...:
```



Definiowanie własnych funkcji

```
In[1]:def dodaj(z1, z2):
...:     return z1 + z2
```

tworzy funkcję sumującą oba argumenty (ważne jest wcięcie wszystkich instrukcji)

```
In[2]:dodaj(1,2)
Out[2]:3
```

wywołuje odpowiednią funkcję

```
In [1]: def dodaj(z1, z2):
...:         return z1 + z2
...:
...:
```

```
In [2]: dodaj(1,2)
Out[2]: 3
```

Opracowanie sprawozdania z laboratorium zawiera opis:

1. Instalacji Pythona 3.X
2. Instalacji środowiska programistycznego Spyder
3. Instalacji bibliotek numerycznych
4. Test działania Pythona w oparciu o omówione konstrukcje programistyczne (należy zaprezentować podobne, ale nie identyczne).