Introduction to Python

Xuemao Zhang
East Stroudsburg University

January 22, 2024

Outline

- Introduction to Python
- Python variables
- Python data types
  - Number
  - Boolean
  - list
  - Array in `numpy`
- Python functions
- `if ... else`
- `for` loops
- `while` loops

Introduction to Python

- Python is a high-level, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant **indentation**.

```python
# define a function called factorial
def factorial(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result


# Test the function
result = factorial(5)
print(f"The factorial of 5 is: {result}") # formatted string
```

```
## The factorial of 5 is: 120
```

Introduction to Python

- Download: https://www.python.org/downloads/
    - You **do not** need to **download** and install python on your computer for this course.
        - We'll use cloud computing Google Colaboratory https://colab.research.google.com/notebooks/

Introduction to Python

- Why you should consider Python?
  - Python is free, open-source.
  - Simplicity and readability: Python's syntax is clear and concise, so we can focus more on the mathematical concepts.
  - Python allows you to execute code interactively using Jupyter Notebooks or or visual stuido code etc.
  - Python packages **numpy**, **sympy** and **scipy** are powerful tools for differential calculus and integral calculus.
  - Community and Resources: Countless online resources, forums, and tutorials are available to help you navigate the world of Python programming for mathematics.
  - It is a leading programming language in data science.

Introduction to Python

What Is a Python Package?
https://www.udacity.com/blog/2021/01/what-is-a-python-package.html

- A python module is a Python program that you import, either in interactive mode or into your other programs. 'Module' is really an umbrella term for reusable code.
- A python **package** or **library** is a collection of modules. Modules that are related to each other are mainly put in the same package/library.
- Python Package Index (PyPI) is the repository of software for Python at http://pypi.python.org/pypi. As of a day in December 2023, there are over 200,000 python packages to ease developers' regular programming experience. Once a package/library is successfully installed, then you can import the package/library within your script.

Introduction to Python

- There are several important Python packages for mathematicians are `numPy`, `sympy`, `scipy` and the built-in module `math`
- **numpy**: A fundamental package for scientific computing with Python. It provides support for large, multi-dimensional arrays and matrices, along with mathematical functions to operate on these arrays.
- **sympy**: A **symbolic** mathematics library that allows for symbolic computation, including algebraic manipulation, equation solving, calculus, and more.
- **scipy**: `SciPy` builds on NumPy and provides additional functionality for scientific computing. It includes modules for optimization, integration, ODEs (Ordinary Differential Equation), statistics, and more.
- **math**: The `math` module is part of the Python standard library and provides basic mathematical operations. It includes functions for arithmetic, logarithms, trigonometry, and more.

Python variables

- Python has no command for declaring a variable. A variable is created the moment you first assign a value to it.
- Variable names are case-sensitive.

```
a = 4
A = 40
print(a)
```

```
## 4
```

```
print(A)
```

```
## 40
```

- A variable can have a short name (like x and y) or a more descriptive name (age, length, total_volume).

Python variables

- Rules for Python variables:
  - A variable name must start with a letter or the underscore character
  - A variable name cannot start with a number
  - A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
  - Variable names are case-sensitive (age, Age and AGE are three different variables)
- Legal variable names:

```python
myvar = 9
my_var = 9
_my_var = 9
myVar = 9
MYVAR = 9
myvar2 = 9
```

- Illegal variable names:

```
2myvar = 9
my-var = 9
my var = 9
```

Python variables

- Many Values to Multiple Variables: Python allows you to assign values to multiple variables in one line:

```
x, y, z = 1,2,3
print(x)
```

```
## 1
```

```
print(y)
```

```
## 2
```

```
print(z)
```

```
## 3
```

- And you can assign the same value to multiple variables in one line

```
x= y= z = 4
print(x)
```

```
## 4
```

```
print(y)
```

Python variables

- The Python `print()` function is often used to output variables.
- In the `print()` function, you output multiple variables, separated by a comma:

```
print(x, y, z)
```

```
## 4 4 4
```

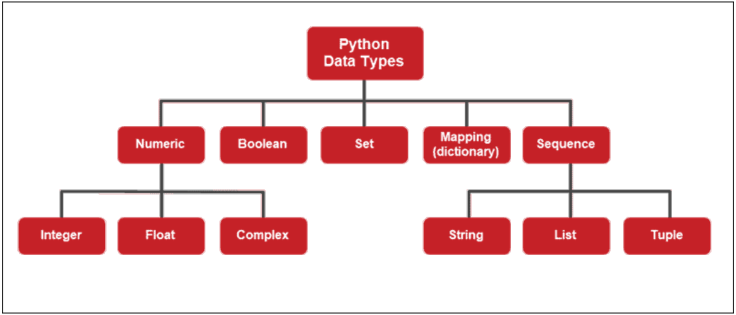
- Formatted string

```
print(f"The values of x, y and z are: {x, y, z}.")
```

```
## The values of x, y and z are: (4, 4, 4).
```

Python data types: Number

- Python has a lot of data types, we consider `Number`, `Boolean` and `List` only.
- A **data type** is a characteristic that tells the compiler (or interpreter) how a programmer intends to use the data. There are two general categories of data types, differing whether the data is changeable after definition:
  - **Mutable**. Data types that are changeable after assignment.
  - **Immutable**. Data types that are not changeable after assignment.



Python data types: Number

- Python numbers are created by the standard way

```
var = 382
print(var)
```

```
## 382
```

- We use function `type()` to check data type

```
type(var)
```

```
## <class 'int'>
```

Python data types: Number

Type	Format	Description
int	a = 10	Signed Integer
float	a = 45.67	Floating point real values, numbers defined with a decimal point
complex	a = 1+2j	<real part> + <complex part>j

- Most of the time using the standard Python number type is fine. Python will automatically convert a number from one type to another if it needs. But, under certain circumstances that a specific number type is needed (ie. complex, hexidecimal), the format can be forced into a format by using a function.

Python data types: Number

- For example, let's use the function `float()` and `int()`
  - Note: Python3.x Version deleted the long integer data type and the `long()` function.

```
print(type(float(34)))
```

```
## <class 'float'>
```

```
print(type(int(34.5)))
```

```
## <class 'int'>
```

```
print(type(30))
```

```
## <class 'int'>
```

```
print(type(30/3))
```

```
## <class 'float'>
```

```
print( type(int(30/3)) )
```

```
## <class 'int'>
```

Python data types: Boolean

- Booleans represent one of two values: `True` or `False`.

```
print(type(True))
```

```
## <class 'bool'>
```

```
print(type(False))
```

```
## <class 'bool'>
```

```
print(True == 1)
```

```
## True
```

```
print(False == 0)
```

```
## True
```

```
print(2 == 2)
```

```
## True
```

```
print(2 == 3)
```

```
## False
```

Python data types: List

- A Python list is an **ordered mutable/changeable** array. Lists **allow duplicate elements** regardless of their type. Adding or removing members from a list allows changes after creation.
- Create a list in Python by using square brackets, separating individual elements with a comma.
  - Each element can be of any data type.
- All lists in Python are *zero-based indexed*. You can access individual list elements. When referencing a member or the length of a list the number of list elements is always the number shown plus one.

```
A = [1, 2, 3, 3.4]
print(A, "is", type(A))
```

```
## [1, 2, 3, 3.4] is <class 'list'>
```

Python data types: List

```
B = len(A)
# `len` will return the length of the list which is 3.
# The index is 0, 1, 2, 3.
print(A[0],A[1],A[2],A[3])
```

```
## 1 2 3 3.4
```

```
print(A[0:2]) #stop=2; the end element is not included
```

```
## [1, 2]
```

- The **colon operator** : is used for slicing, indexing a specific range and displaying the output using colon operator.
- By leaving out the start value, the range will start at the first item:

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

```
## [1, 2]
```

Python data types: List

- Negative indexes are also possible
  - -1 refers to the last item, -2 refers to the second last item etc.
  - Negative indexing is especially useful for navigating to the end of a long list of members.

```
print(A[-1])
```

```
## 3.4
```

Python data types: List

- The elements of a list can be list

```
A=[[1,2],[3,4]]
print(A)
```

```
## [[1, 2], [3, 4]]
```

- **Append** Items: To add an item to the end of the list, use the `append()` method

```
A=[[1,2],[3,4]]
B=[5,6]
A.append(B)
print(A)
```

```
## [[1, 2], [3, 4], [5, 6]]
```

Python data types: List

- **Extend** Items: To add all elements of an item to the end of the list, use the `extend()` method
- The following is to **append elements** from another list to the current list by the `extend()` method.

```
A=[[1,2],[3,4]]
B=[5,6]
A.extend(B)
print(A)
```

```
## [[1, 2], [3, 4], 5, 6]
```

Python data types: Array

- Defined in library `numpy` (http://www.scipy-lectures.org/), vectors and matrices are for numerical data manipulation.

```
import numpy as np
data1 = [1, 2, 3, 4, 5] # list
arr1 = np.array(data1)  # 1d array
print(type(arr1))
```

```
## <class 'numpy.ndarray'>
```

```
print(arr1)
```

```
## [1 2 3 4 5]
```

```
print(arr1[1])  #python index starts from 0
```

```
## 2
```

Python data types: Array

- Arrays can be two-dimensional

```
data2 = [range(1, 5), range(5, 9)] # list of lists
# range() generates a range of numbers
arr2 = np.array(data2) # 2d array or matrix.
# arr2 = np.array([range(1, 5), range(5, 9)])

print(arr2)
```

```
## [[1 2 3 4]
##  [5 6 7 8]]
```

```
print(np.shape(arr2)) #dimensions of the matrix
```

```
## (2, 4)
```

```
print(type(arr2))
```

```
## <class 'numpy.ndarray'>
```

```
print(arr2[0,1])
```

```
## 2
```

```
print(arr2[1,1])
```

```
## 6
```

Python data types: Array

```
print(arr2[0, :]) # row 0
```

```
## [1 2 3 4]
```

```
print(arr2[:, 0]) # column 0
```

```
## [1 5]
```

```
print(arr2[:, :2])
```

```
## [[1 2]
##  [5 6]]
```

```
#columns strictly before index 2 (the first 2 columns)
```

Python data types: Array

```
print(arr2[:, 2:])
```

```
## [[3 4]
##  [7 8]]
```

```
# columns after index 2 ( column 2 included)
```

```
print(arr2[:, 1:4])
```

```
## [[2 3 4]
##  [6 7 8]]
```

```
# columns between index 1 (included) and 4 (excluded)
```

- The double colon `::` operator in python are used for jumping of elements in multiple axes. It is also a slice operator. Every item of the sequence gets sliced using double colon.

```
print(arr2[:, 2::])  #it is equivalent to print(arr2[:, 2:])
```

```
## [[3 4]
##  [7 8]]
```

```
print(arr2[:, 2:])
```

```
## [[3 4]
##  [7 8]]
```

Python data types: Array

- The syntax of a Slice operator using double colon is [**Start** : **Stop** : **Steps**].
    - **Start** (Indicates the number from where the slicing will start),
    - **Stop**(Indicates the number where the slicing will stop) and
    - **Steps**(Indicates the number of jumps interpreter will take to slice the string) are the three flags and all these flags are integer values.

```
print(arr2[:, 0:3:2]) # all rows, every other column
```

```
## [[1 3]
##  [5 7]]
```

```
#The above code can be reduced to a short cut by using double colon ::
print(arr2[:, ::2])
```

```
## [[1 3]
##  [5 7]]
```

```
print(arr2[:, ::-1]) # reverse order of columns
```

```
## [[4 3 2 1]
##  [8 7 6 5]]
```

Python data types: Array

```
y=arr2[:, [0,1,2,2]] #column 0,1,2,2 of arr2
print(y)
```

```
## [[1 2 3 3]
##  [5 6 7 7]]
```

- vector and matrix of ones

```
x1=np.ones(3)
print(x1)
```

```
## [1. 1. 1.]
```

```
x2=np.ones((3,3))
print(x2)
```

```
## [[1. 1. 1.]
##  [1. 1. 1.]
##  [1. 1. 1.]]
```

Python data types: Array

- vector and matrix of zeros

```
x1=np.zeros(3)
print(x1)
```

```
## [0. 0. 0.]
```

```
x2=np.zeros((3,3))
print(x2)
```

```
## [[0. 0. 0.]
##  [0. 0. 0.]
##  [0. 0. 0.]]
```

```
x2[0,0]=999  #change element
print(x2)
```

```
## [[999.   0.   0.]
##  [  0.   0.   0.]
##  [  0.   0.   0.]]
```

Python functions

- A function takes a list of argument values, performs a computation with those values, and returns a single result. Python gives you many built-in functions.
  - see Python Built-in Functions https://docs.python.org/3/library/functions.html

```
print(abs(-2))
```

```
## 2
```

```
print(round(3.1415926,3))
```

```
## 3.142
```

```
print(any([True,False]))
```

```
## True
```

Python functions

- We can also create our own functions. These functions are called user-defined functions.
- Functions are declared using the **def** keyword, and the value produced is returned using the **return** keyword. Consider a simple function which returns the square of the input, $y = x^2$.
  - colon : is used to represent an indented block. It is not for slicing.
  - Python uses **indentation** to indicate a block of code.
  - The number of spaces is up to you as a programmer, but it has to be at least one.

```
def square(x):
  return x**2

x = 2
y = square(x) # Call the function
print(x,y)
```

```
## 2 4
```

Python `if ... else`

- Comparison Operators (like `==, >, <, >=, <=`) are used to evaluate to True or False depending on input condition.
- An `if statement` is written by using the **if** keyword.
  - Python relies on indentation to define scope in the code. Other programming languages often use curly-brackets for this purpose.
- `if` keyword

```
if logical:
   Code to run if logical True
```

```python
a = 33
b = 200
if b > a:
  print("b is greater than a")
```

```
## b is greater than a
```

Python `if ... else`

- else keyword

```
if logical:
    Code to run if logical True
else:
    Code to run if logical False
```

```python
a = 33
b = 33
if b > a:
  print("b is greater than a")
else:
  print("b is not greater than a")
```

```
## b is not greater than a
```

Python `if ... else`

- The **elif** keyword is pythons way of saying "if the previous conditions were not true, then try this condition" or `else if`.

```
a = 33
b = 33
if b > a:
  print("b is greater than a")
elif a == b:  #no indentation
  print("a and b are equal")
```

```
## a and b are equal
```

Python `if ... else`

- The **else** keyword catches anything which isn't caught by the preceding conditions.

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

```
## a is greater than b
```

Python `if ... else`

- One more example

```
x = 5
if x<5:
    x+=1
elif x>5:
    x-=1
else:
    x=x**2
print(x)
```

```
## 25
```

Python `if ... else`

- The `and` keyword is used to combine conditional statements

```
a = 200
b = 33
c = 500
if a > b and c > a:
  print("Both conditions are True")
```

```
## Both conditions are True
```

- The `or` keyword is used to combine conditional statements

```
a = 200
b = 33
c = 500
if a > b or a > c:
  print("At least one of the conditions is True")
```

```
## At least one of the conditions is True
```

Python `for` loops

- A `for` loop is used for iterating over a sequence (that is either a range, list, a tuple, a dictionary, a set, or a string).

```
for item in iterable:
  Code to run
```

- The `range()` function allows you to iterate over a sequence of numbers. It starts from 0, increments by 1, and stops before a specified number

```python
for i in range(4):
  print(i)
```

```
## 0
## 1
## 2
## 3
```

Python `for` loops

```python
for i in range(1, 5, 1):
 print(i)
```

```
## 1
## 2
## 3
## 4
```

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

```
## 1
## 3
## 5
## 7
## 9
```

Python `for` loops

- The `break` statement: with the `break` statement we can stop the loop before it has looped through all the items

```
for i in [1, 2, 3, 4, 5]:
  if i == 3:
    break
  else:
    print(i)
```

```
## 1
## 2
```

Python `for` loops

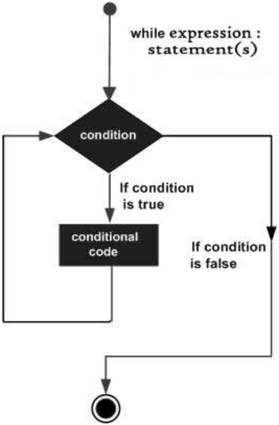
- The `continue` statement: with the `continue` statement we can stop the current iteration of the loop, and continue with the next

```
for i in [1, 2, 3, 4, 5]:
  if i == 3:
    continue
  else:
    print(i)
```

```
## 1
## 2
## 4
## 5
```

Python `while` loops

- With the `while` loop we can execute a set of statements as long as a condition is true.



Python `while` loops

```
a=5
while a<10:
  print(a)
  a+=1
```

```
## 5
## 6
## 7
## 8
## 9
```

```
print("Out of Loop")
```

```
## Out of Loop
```

- `while` loops should generally be avoided when `for` loops are sufficient. However, there are situations, for example the number of iterations required is not known in advance, where no `for` loop equivalent exists.

Python `while` loops

- Example: use Newton's method find the roots of $f(x) = x^3 - 5x + 1$.
- First, we define the function and its first derivative

```
def f(x):
    return x**3 - 5*x + 1

def df(x):
    return 3*x**2 - 5
```

Python `while` loops

```python
x_0 = 4 #initial guess: -3, 1, 4
x_n = x_0+0.1  #updated x
tolerance=1e-6
max_iterations=100

iteration = 0
while iteration < max_iterations and abs(x_n-x_0)>tolerance:
    x_0=x_n
    f_x_n = f(x_n)
    df_x_n = df(x_n)
    x_n = x_n - f_x_n / df_x_n #updated x_n
    iteration += 1

if abs(x_n - x_0) <= tolerance:
    print(f"Root found: {x_n} after {iteration} iterations")
else:
    print("Maximum iterations reached. No root found.")
```

```
## Root found: 2.1284190638445772 after 7 iterations
```

License

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

Numerical Integration

In this handout, we consider three numerical integration methods: Midpoint Rule, Trapezoidal Rule, and Simpson's Rule. We have discussed the Midpoint Rule in the beginning of the course, but we did not consider Errors of Approximation. We use the three numerical methods to approximate the following integral in this handout.

$$\int_0^1 e^{x^2}\,dx$$

Midpoint Rule

Let $a = x_0 < x_1 = a + \frac{b-a}{n} < \cdots < x_i = a + i\frac{b-a}{n} < \cdots < x_n = a + n\frac{b-a}{n} = b$ be a partition of the interval $[a, b]$ with n subintervals of equal length. The approximation to $\int_a^b f(x)\,dx$ using the Midpoint Rule is

$$M_n = \int_a^b f(x)\,dx \approx \Delta x\big[f(\bar{x}_1) + \cdots + f(\bar{x}_{n-1}) + f(\bar{x}_n)\big],$$

where $\bar{x}_i = \frac{x_{i-1}+x_i}{2}$ is the midpoint of the subinterval. And the the error of the Midpoint Approximation E_M is bounded by

$$|E_M| = \left|\int_a^b f(x)\,dx - M_n\right| \le \frac{K(b-a)^3}{24n^2},$$

where $|f''(x)| \le K$ on $[a, b]$.

We write a Python function to do the above calculations.

In [ ]:
```
import numpy as np

# enclose the comments in triple quotes
def midpoint_rule(func, a, b, n):
    """
    Compute the definite integral of a function using the midpoint rule.

    Parameters:
    - func: The function to be integrated.
    - a, b: The interval of integration [a, b].
    - n: The number of subintervals.

    Returns:
    - result: The approximate integral.
    - error: An estimate of the absolute error.
    """

    dx = (b - a) / n  # Width of each subinterval
```

```
    x = np.linspace(a, b, n+1)
    x_mid = (x[:-1] + x[1:]) / 2
    result = np.sum(func(x_mid) * dx)

    # Error estimation using the second derivative
    x_values = np.linspace(a, b, num=1000)
    second_derivative = np.max(np.abs(np.gradient(np.gradient(func(x_values), x_val
    error = (b - a) * dx**2 * second_derivative / 24

    return result, error
```

In [ ]:
```
# Example usage:
def f(x):
    return np.exp(x**2)

# Integration interval and number of subintervals
a, b = 0, 1
n = 1000

# Calculate the midpoint rule result and error
result, error = midpoint_rule(f, a, b, n)

# Display the result and error
print("Midpoint Rule Result:", result)
print("Error Estimate:", error)
```

```
Midpoint Rule Result: 1.462651519383762
Error Estimate: 6.757312382793164e-10
```

Compare our results with the *quad()* function in *Scipy* library.

In [ ]:
```
from scipy.integrate import quad
result, error = quad(f, a, b)
print("Quad Result:", result)
print("Error Estimate:", error)
```

```
Quad Result: 1.4626517459071815
Error Estimate: 1.623869645314337e-14
```

Trapezoidal Rule

Let $a = x_0 < x_1 = a + \frac{b-a}{n} < \cdots < x_i = a + i\frac{b-a}{n} < \cdots < x_n = a + n\frac{b-a}{n} = b$ be a partition of the interval $[a, b]$ with n subintervals of equal length. The approximation to $\int_a^b f(x)\,dx$ using the Trapezoidal Rule is

$$T_n = \int_a^b f(x)\,dx \approx \frac{\Delta x}{2}\big[f(x_0) + 2f(x_1) + \cdots + 2f(x_{n-1}) + f(x_n)\big].$$

Moreover, if $|f''(x)| \leq K$ on $[a, b]$ and E_T is the error of the approximation, then

$$|E_T| = \left|\int_a^b f(x)\,dx - T_n\right| \leq \frac{K(b-a)^3}{12n^2}.$$

Calculations using the Trapezoidal Rule are similar to the Midpoint Rulewhich is left as a homework.

Simpson's Rule

Partition $[a, b]$ into $n = 2k$ subintervals using the end points

$$x_0 = a, x_1 = a + \Delta x, x_2 = a + 2\Delta x, \cdots, x_n = a + 2k\Delta x = b.$$

Then

$$\begin{aligned} S_n = \int_a^b f(x)\, dx \approx \frac{\Delta x}{3} \Big[f(x_0) &+ 4f(x_1) + 2f(x_2) + 4f(x_3) + \cdots \\ &+ 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n) \Big]. \end{aligned}$$

Moreover, if $|f^{(4)}(x)| \leq K$ for all x in $[a, b]$ and E_S is the error of the Simpson's Rule approximation, then

$$|E_S| = \left| \int_a^b f(x)\, dx - S_n \right| \leq \frac{K(b-a)^5}{180n^4}.$$

In [ ]:

```
import numpy as np

# enclose the comments in triple quotes
def simpsons_rule(func, a, b, n):
    """
    Compute the definite integral of a function using Simpson's Rule.

    Parameters:
    - func: The function to be integrated.
    - a, b: The interval of integration [a, b].
    - n: The number of subintervals (must be even).

    Returns:
    - result: The approximate integral.
    - error: An estimate of the absolute error.
    """

    if n % 2 != 0:
        raise ValueError("The number of subintervals (n) must be even for Simpson's

    h = (b - a) / n  # Width of each subinterval
    x = np.linspace(a, b, n+1)

    result = h/3 * (func(x[0]) + 4*np.sum(func(x[1:-1:2])) + 2*np.sum(func(x[2:-2:2

    # Error estimation using the fourth derivative
    x_values = np.linspace(a, b, num=1000)
    fourth_derivative = np.max(np.abs(np.gradient(np.gradient(np.gradient(np.gradie
```

```python
    error = (b - a) * h**4 * fourth_derivative / 180

    return result, error
```

In [ ]:
```python
# Example usage:
result, error = simpsons_rule(f, a, b, n)

# Display the result and error
print("Simpson's Rule Result:", result)
print("Error Estimate:", error)
```

```
Simpson's Rule Result: 1.4626517459074835
Error Estimate: 1.1351474659300735e-11
```

Now, we compare our result with the Simpson's Rule in library *Scipy*.

In [ ]:
```python
from scipy.integrate import simps
# Sample points
x = np.linspace(a, b, 1000)
y = f(x)
result_simpson = simps(y, x)
print("SciPy Simpson's Rule Result:", result_simpson)
#it does not provide estimating error
```

```
SciPy Simpson's Rule Result: 1.4626517459097506
```

Calculations of Riemann Sum

In this handout, We use the python package `numpy` to calculate Riemann sums. NumPy provides a powerful array object that can represent arrays of any dimension. We need 1-dimensional vectors only

1. Getting started

To use numpy, we need to import it to Python first.

In [ ]:
```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
print(arr[0])
print(arr[0:3]) #The first 3 element
print(arr[4])  #last element
print(arr[-1])  #negative index  to access the last element of the array
print(arr[-2])
```

```
[1 2 3 4 5]
1
[1 2 3]
5
5
4
```

In [ ]:
```
#some operations
print(2*arr)
arr2= np.array([6, 7, 8, 9, 10])
print(arr+arr2)
print(arr*arr2) #element-wise multiplication
```

```
[ 2  4  6  8 10]
[ 7  9 11 13 15]
[ 6 14 24 36 50]
```

In [ ]:
```
np.dot(arr, arr2) #dot product
```

Out[ ]: 130

In [ ]:
```
arr@arr2 #dot product
```

Out[ ]: 130

In [ ]:
```
np.sum(arr)  #sum of all elements
```

Out[ ]: 15

2. Calculation of Riemann sums

Consider the function f(x)=x^2-4*x on the closed interval [0, 4]. Lambda functions in Python are anonymous functions created using the lambda keyword in Python.

In [ ]:
```
f = lambda x: x**2-4*x    # input parameter is x
a = 0
b = 4
n = 4  # you may update it to a large number later
```

In [ ]:
```
f(0)
```

Out[ ]:
```
0
```

In [ ]:
```
dx=(b-a)/n
```

In [ ]:
```
import numpy as np
```

In [ ]:
```
#an array of n+1 evenly spaced values starting from a to b
x = np.linspace(a,b,n+1)
print(x)
```

```
[0. 1. 2. 3. 4.]
```

In [ ]:
```
x_left=x[:-1]
x_left
```

Out[ ]:
```
array([0., 1., 2., 3.])
```

In [ ]:
```
left_riemann_sum = np.sum(f(x_left) * dx)
print("Left Riemann Sum:", left_riemann_sum)
```

```
Left Riemann Sum: -10.0
```

In [ ]:
```
x_right = x[1:]
x_right
```

Out[ ]:
```
array([1., 2., 3., 4.])
```

In [ ]:
```
right_riemann_sum = np.sum(f(x_right) * dx)
print("Right Riemann Sum:",right_riemann_sum)
```

```
Right Riemann Sum: -10.0
```

In [ ]:
```
x_mid = (x[:-1] + x[1:])/2
x_mid
```

Out[ ]:
```
array([0.5, 1.5, 2.5, 3.5])
```

In [ ]:
```
midpoint_riemann_sum = np.sum(f(x_mid) * dx)
print("Midpoint Riemann Sum:",midpoint_riemann_sum)
```

```
Midpoint Riemann Sum: -11.0
```

using for loop (it is a worse option)

In [ ]:
```
for i in range(4):
  print(i)
```

```
0
1
2
3
```

In [ ]:
```
rightsum=0
for i in range(n):
  #rightsum=rightsum+f(x_right[i])*dx
  #print(x_right[i])
  rightsum+=f(x_right[i])*dx
```

In [ ]:
```
rightsum
```

Out[ ]:
```
-10.0
```

Sequences

A sequence is list of numbers written in a definite order which follows a pattern. Therefore, we can use Python to generate, manipulate, and analyze mathematical sequences.

Printing first n terms of a sequence

In [ ]:
```
def f(x):
    return x/(x+1)

n=5
for i in range(1,n+1):
    print(f(i))
```

```
0.5
0.6666666666666666
0.75
0.8
0.8333333333333334
```

In [ ]:
```
import numpy as np
def g(x):
    return np.sqrt(x-3)

n=5
for i in range(3,n+3):
    print(g(i))
```

```
0.0
1.0
1.4142135623730951
1.7320508075688772
2.0
```

In [ ]:
```
import numpy as np
def h(x):
    return np.cos(x*np.pi/6)

n=5
for i in range(n):
    print(h(i))
```

```
1.0
0.8660254037844387
0.5000000000000001
6.123233995736766e-17
-0.4999999999999998
```

In [ ]:
```
# Fibonacci sequence is defined recursively
def fibonacci_sequence(n):
    fib_seq = [1, 1] #the first two terms
    while len(fib_seq) < n:
```

```
        fib_seq.append(fib_seq[-1] + fib_seq[-2])
    return fib_seq

# Generate Fibonacci sequence with 10 terms
fibonacci_seq = fibonacci_sequence(10)
print("Fibonacci Sequence:", fibonacci_seq)
```

```
Fibonacci Sequence: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Limit of a sequence

If one can associate a function $f(x)$ with the sequence $\{a_n\}$, we find the limit of the sequence using the Function Value Theorem.

In [ ]:
```
# Find the limit of the sequence lnn/n
from sympy import symbols, limit, log, oo
n = symbols('n')
sequence = log(n) / n
result = limit(sequence, n, oo)
print("Limit of the sequence ln(n)/n as n approaches infinity:", result)
```

```
Limit of the sequence ln(n)/n as n approaches infinity: 0
```

In [ ]:
```
# Find the limit of the sequence (1+5/n)**n
from sympy import symbols, limit, log, oo
n = symbols('n')
sequence = (1+5/n)**n
result = limit(sequence, n, oo)
print("Limit of the sequence (1+5/n)**n as n approaches infinity:", result)
```

```
Limit of the sequence (1+5/n)**n as n approaches infinity: exp(5)
```

Computation of limit of a sequence can be done with function *sympy.series.limitseq.limit_seq* in library *Sympy*. See Limits of Sequences https://docs.sympy.org/latest/modules/series/limitseq.html

In [ ]:
```
from sympy import limit_seq
from sympy.abc import n
limit_seq((1+5/n)**n, n)
```

Out[ ]: e^5

Series

A series is the mathematical expression obtained by adding together **all the terms** of a sequence, capturing the **cumulative sum** of its elements. If the sum of the terms converges to a finite value, the series is said to converge. If the sum of the terms diverges (does not approach a finite value), then the series is said to diverge. Python can be used for determing the convergence of a sereies.

1. Convergence or Divergence of a Series

In [ ]:
```
# partial sum of the sequence 1/n
def partial_sum(func,n):
    series_partialsum = 0.0
    for i in range(1, n + 1):
        series_partialsum += func(i)
    return series_partialsum

# Example: Calculate the sum of the first 5 terms
def f(x):
    return 1/x

n=10
result = partial_sum(f,n)

print(f"The sum of the first {n} terms of the sequence is: {result}")
```

```
The sum of the first 10 terms of the sequence is: 2.9289682539682538
```

Or we can use function *summation* in library *sympy*.

In [ ]:
```
from sympy import symbols,summation
n=10
x = symbols('x', integer=True)
partial_sum_result = summation(f(x), (x, 1, n))
print(type(partial_sum_result))
print(partial_sum_result.evalf())
```

```
<class 'sympy.core.numbers.Rational'>
2.92896825396825
```

Now let's calculate the limit of the partial sum to check convergence/divergence of the Harmonic Series $\sum 1/n$.

In [ ]:
```
from sympy import symbols, summation, oo

x = symbols('x', integer=True)
f = 1/x

# Calculate the partial sum for n=10
```

```
limit_partial_sum = summation(f, (x, 1, oo))
print("Limit of the Partial Sum as n approaches infinity:", limit_partial_sum)
```

```
Limit of the Partial Sum as n approaches infinity: oo
```

Let's consider the convergence/divergence of the Alternating Harmonic Series $\sum(-1)^n/n$.

In [ ]:
```
g = (-1)**x / x
limit_partial_sum = summation(g, (x, 1, oo))
print("Limit of the Partial Sum as n approaches infinity:", limit_partial_sum)
```

```
Limit of the Partial Sum as n approaches infinity: -log(2)
```

Another example: convergence/divergence of $\sum \frac{n^2}{e^{2n}}$.

In [ ]:
```
from sympy import exp
h = x**2 / exp(2*x)
limit_partial_sum = summation(h, (x, 1, oo))
print("Limit of the Partial Sum as n approaches infinity:", limit_partial_sum)
print(type(limit_partial_sum))
print("Limit of the Partial Sum as n approaches infinity:", limit_partial_sum.evalf
```

```
Limit of the Partial Sum as n approaches infinity: (exp(-2) + 1)*exp(-2)/(1 - exp
(-2))**3
<class 'sympy.core.mul.Mul'>
Limit of the Partial Sum as n approaches infinity: 0.237679627431505
```

2. Series Expansion

For more information, see Series Expansions
https://docs.sympy.org/latest/modules/series/series.html

In [ ]:
```
from sympy import symbols, cos, series
x,p = symbols('x,p')
series((1+x)**p,x,x0=0, n=4)  #binomial series
```

Out[ ]:

$$1 + px + \frac{px^2\left(p-1\right)}{2} + \frac{px^3\left(p-2\right)\left(p-1\right)}{6} + O\left(x^4\right)$$

In [ ]:
```
series(x*cos(x),x,x0=0, n=14)
```

Out[ ]:

$$x - \frac{x^3}{2} + \frac{x^5}{24} - \frac{x^7}{720} + \frac{x^9}{40320} - \frac{x^{11}}{3628800} + \frac{x^{13}}{479001600} + O\left(x^{14}\right)$$

In [ ]:
```
from sympy import tan,atan
series(tan(x), x, x0=0, n=14, dir="+")
```

Out[ ]:

$$x + \frac{x^3}{3} + \frac{2x^5}{15} + \frac{17x^7}{315} + \frac{62x^9}{2835} + \frac{1382x^{11}}{155925} + \frac{21844x^{13}}{6081075} + O\left(x^{14}\right)$$

In [ ]:
```
series(atan(x), x, x0=0, n=14, dir="+")
```

Out[ ]: $x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \frac{x^{11}}{11} + \frac{x^{13}}{13} + O\left(x^{14}\right)$

3. Formal power series

A formal power series is of the form

$$\sum a_n x^n$$

that is considered independently from any notion of convergence, and can be manipulated with the usual algebraic operations on series (addition, subtraction, multiplication, division, partial sums, etc.).

For more information, see Formal Power Series https://docs.sympy.org/latest/modules/series/formal.html#

3.1 Formal Power Series Expansion

In [ ]:
```
from sympy import fps, sin, exp
from sympy.abc import x
f1 = fps(exp(x))
f2 = fps(sin(x))
f1
```

Out[ ]:
$$\left(\sum_{k=1}^{\infty} \begin{cases} \frac{x^k}{k!} & \text{for}\ k \bmod 1 = 0 \\ 0 & \text{otherwise} \end{cases}\right) + 1$$

In [ ]:
```
f2
```

Out[ ]:
$$x + \left(\sum_{k=2}^{\infty} \begin{cases} \frac{\left(-\frac{1}{4}\right)^{\frac{k}{2}-\frac{1}{2}} x^k}{\left(\frac{3}{2}\right)^{\left(\frac{k}{2}-\frac{1}{2}\right)} \left(\frac{k}{2}-\frac{1}{2}\right)!} & \text{for}\ k \bmod 2 = 1 \\ 0 & \text{otherwise} \end{cases}\right)$$

3.2 Limit of formal power series

In [ ]:
```
from sympy import limit
# Calculate the limit as x approaches 0
limit_result=limit(f1, x, 0, dir='+')
limit_result # does not yield numerical result
```

Out[ ]:
$$\lim_{x \to 0^+} \left(\sum_{k=1}^{\infty} \begin{cases} \frac{x^k}{k!} & \text{for}\ k \bmod 1 = 0 \\ 0 & \text{otherwise} \end{cases}\right) + 1$$

3.3 Addition of formal power series

In [ ]:

```
from sympy import log
g = fps(log(1+x))
h = fps(log(1-x))
g+h
```

Out[ ]:

$$\sum_{k=1}^{\infty} x^{k} \left(- \frac{1}{k} - \frac{\left(-1\right)^{- k}}{k}\right)$$

In [ ]:

```
g-h
```

Out[ ]:

$$\sum_{k=1}^{\infty} x^{k} \left(\frac{1}{k} - \frac{\left(-1\right)^{- k}}{k}\right)$$

3.4 Composition of functions

Function *compose* returns the truncated terms of the formal power series of the **composed function**, up to specified n.

In [ ]:

```
f1.compose(f2, x).truncate(10)
```

Out[ ]:

$$1 + x + \frac{x^{2}}{2} - \frac{x^{4}}{8} - \frac{x^{5}}{15} - \frac{x^{6}}{240} + \frac{x^{7}}{90} + \frac{31 x^{8}}{5760} + \frac{x^{9}}{5670} + O\left(x^{10}\right)$$

3.5 Integrating Formal Power Series

In [ ]:

```
from sympy import fps, sin, integrate
from sympy.abc import x
#f2 = fps(sin(x))
integrate(f2, x)
```

Out[ ]:

$$\frac{x^{2}}{2} + \left(\sum_{k=3}^{\infty} \begin{cases} \frac{\left(- \frac{1}{4}\right)^{\frac{k}{2} - 1} x^{k}}{k \left(\frac{3}{2}\right)^{\left(\frac{k}{2} - 1\right)} \left(\frac{k}{2} - 1\right)!} & \text{for}\: \left(k + 1\right) \bmod 2 = 1 \\ 0 & \text{otherwise} \end{cases}\right) - 1$$

In [ ]:

```
integrate(f2, (x,0,1))
```

Out[ ]:

$$1 - \cos{\left(1 \right)}$$

SymPy for Calculus I

SymPy is a Python library for symbolic mathematics. The most well-known commercial programs is certainly Mathematica, while SymPy is one of the most popular free-software computer algebra systems. It aims to become a full-featured computer algebra system (CAS) while keeping the code as simple as possible

1. Getting started

To use sympy, we need to import it to Python first. Let's first see some constants defined in sympy.

In [ ]:
```
from sympy import *
print(pi**2)
print(pi.evalf())  #evaluate pi
print(exp(1))
print(exp(1).evalf())
print(oo) #infinity
print(oo>99999)
print(oo+99999)
```

```
pi**2
3.14159265358979
E
2.71828182845905
oo
True
oo
```

In [ ]:
```
#help(sin)   #get help about sine function
```

2. Symbols

In SymPy we have to declare symbolic variables explicitly to conduct symbolic calculations.

In [ ]:
```
x = Symbol('x')
y = Symbol('y')
# or x,y=symbols("x y")
(x+y)**2
```

Out[ ]: $(x+y)^2$

In [ ]:
```
expand((x+y)**2)  #expand
```

Out[ ]: $x^2+2xy+y^2$

In [ ]:
```
simplify(sin(x)/cos(x))  #simplify
```

Out[ ]: $\tan(x)$

3. Limits

In [ ]:
```python
limit(sin(x)/x, x, 0)
```

Out[ ]: 1

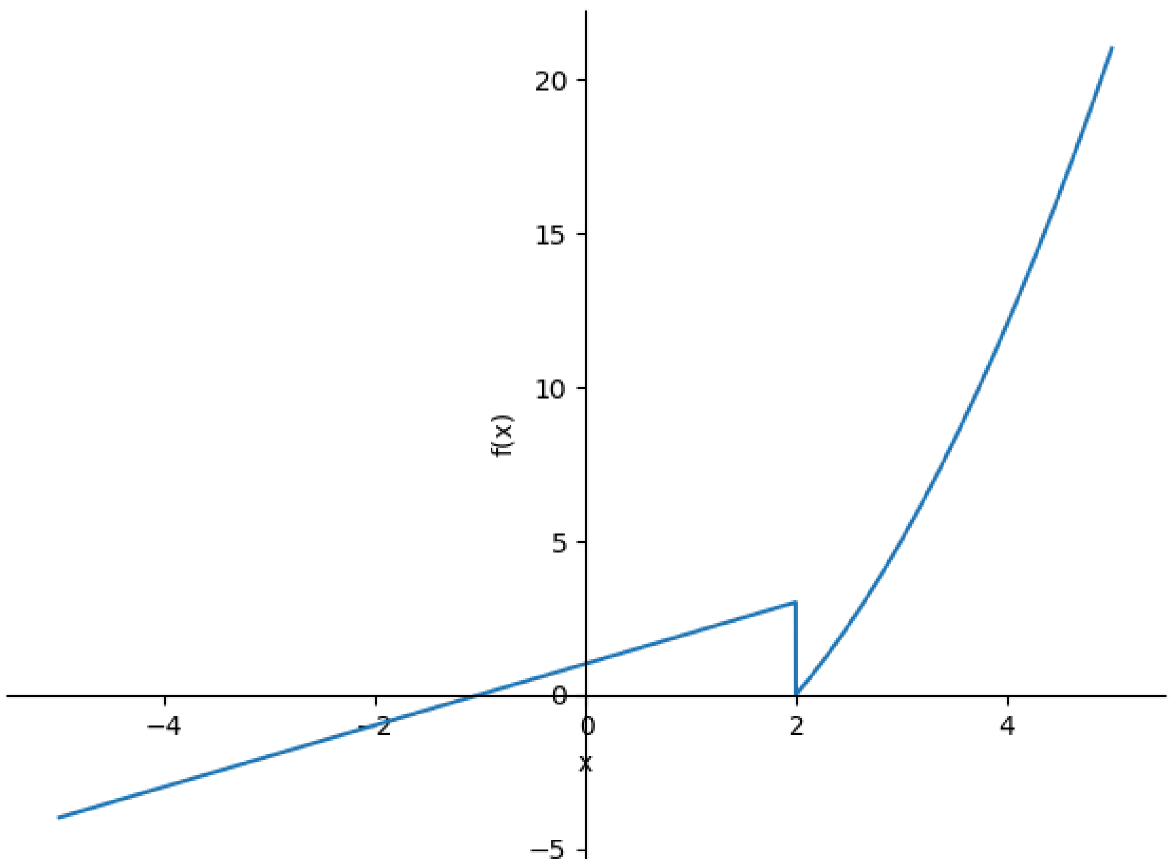
In [ ]:
```python
limit((1-cos(x))/x, x, 0)
```

Out[ ]: 0

In [ ]:
```python
# Define a  piecewise function
# fx = Piecewise( (x+1, x < 2), (x**2-4, True) ) #this does not work
fx = Piecewise( (x+1, x < 2), (x**2-4, x>2), (0, x==2) )
print(fx)
```

```
Piecewise((x + 1, x < 2), (x**2 - 4, x > 2))
```

In [ ]:
```python
plot(fx, (x, -5, 5), xlabel='x', ylabel='f(x)') #it is not  nice for the discontinu
```



Out[ ]: <sympy.plotting.plot.Plot at 0x1c1902e8c70>

In [ ]:
```python
# .subs method substitutes a variable with a specific value.
for x_val in [1.9, 1.99, 1.999, 1.9999]:
    print(f'f({x_val}) = {fx.subs(x, x_val)}')
```

```
f(1.9) = 2.90000000000000
f(1.99) = 2.99000000000000
f(1.999) = 2.99900000000000
f(1.9999) = 2.99990000000000
```

In [ ]:
```
limit(fx, x, 2, dir='-')
```

Out[ ]: 3

In [ ]:
```
limit(fx, x, 2, dir='+')
```

Out[ ]: 0

4. Differentiation

In [ ]:
```
diff(sin(x), x)
```

Out[ ]: $\cos(x)$

In [ ]:
```
diff(exp(sin(x)), x)
```

Out[ ]: $e^{\sin(x)} \cos(x)$

In [ ]:
```
diff(x**x, x)
```

Out[ ]: $x^x (\log(x) + 1)$

In [ ]:
```
#implicit differentiation
x = symbols('x')
y = Function('y')(x)
eqn = sin(y)+y**3-6+x**3  #The equation is sin(y)+y**3-6+x**3=0
idiff(eqn, y, x)
```

Out[ ]:
$$-\frac{3x^2}{3y^2(x) + \cos(y(x))}$$