Notes

July 3, 2025

1 Python Paradigms

Liste des différents paradigmes, syntaxes, procédures et modules qui composent le language

---

1.1 Print

Quelque détails au sujet de `print()`

Arguments Par défaut, print() termine avec un retour de ligne `\n`

[1]:
```
print("Hello")
print("There")
```

```
Hello
There
```

Et sépare les variables par `" "`

[2]:
```
print("Hello", "There")
```

```
Hello There
```

Mais on peut changer ça avec les arguments `end` et `sep`

[3]:
```
print("Hello", end=" ")
print("There")
```

```
Hello There
```

[4]:
```
print("Hello", "There", sep="\n")
```

```
Hello
There
```

F-string Il est possible d'imprimer `"var = ..."` directement dans une f-string

[5]:
```
x = 1
print(f"Valeur de x: {x = }")
```

```
Valeur de x: x = 1
```

String comparaison Des comparaisons booléenes et de taille peuvent être faites avec des `string`

[6]:
```
print("A" > "a")
print("a" < "b")
print("*" < "B")
```

```
False
True
True
```

Python trie comme suit (du plus petit au plus grand): - ' ',';','#', ... ,'/', - '0', '1', '2', ... , '9', - ':', ';', '<', ... , '@', - 'A', 'B', 'C', ... , 'Z', - '[', '^', '_', ... ,'', - 'a', 'b', 'c', ... , 'z', - '{', '|', '}', '~'

1.2 Functions

Les fonctions devraient respecter quelques principes: - Avoir une seule responsabilité - Être testée! - Avoir des noms de variables internes différents des variables externes - Être correctement documentées avec une docstring - Si possible, ne pas utiliser `global`

Docstring Les docstring doivent suivre la même structure standard

[7]:
```
def foo():
    '''
    Description du rôle de la fonction

    Args:
        arg1 (type): description
        arg2 (type): description
        [...]

    Returns:
        output (type): 'return' format, if [...] or not [...]

    Raises:
        error_type: when [...] happens
    '''
```

Exemple

[8]:
```python
def convert_temp(t, unit):
    '''
    Convert a temperature between Celsius, Fahrenheit, and Kelvin.

    Args:
        t (float): The temperature value to convert.
        unit (str): The current unit of temperature; must be one of "celsius",
                    "fahrenheit", or "kelvin".

    Returns:
        tuple: Contains converted temperatures as (int). The specific format␣
 ↪depends on the input unit:
            - If input is celsius, returns: (kelvins, fahrenheit).
            - If input is fahrenheit, returns: (celsius, kelvin).
            - If input is kelvin, returns: (celsius, fahrenheit).

    Raises:
        "c'est quoi cette unité??": If the provided unit is not recognized.
        "error": If the temperature value is too low to be physically possible.
    '''
    pass
```

La documentation d'une fonction peut être accédée par `convert_temp.__doc__`

Tests Il est possible de tester une fonction en vérifiant que pour un paramètre connu, elle ait un return connu.
Pour ce faire, on utilise la syntaxe suivante (pytest cherche des fonction préfixées par `test_\<fonction>` ou des classes `Test`)

[9]:
```python
import pytest

def divise(a, b):
    return a / b

# testing function for divise()
def test_divise():
    assert divise(3, 3) == 1.0
    assert divise(3, 2) != 69
```

Ensuite, on peut run les tests comme suit: `pytest file.py`

Global variables Une fonction peut seulement lire les variables du code principal, pas les modifier, pour palier à ça, on peut déclarer une variable comme globale, qui permet à la fonction de modifier cette variable

[10]:
```
x = 0
def foo():
    global x
    x += 1

foo()
print(x)
```

```
1
```

Cette méthode est déconseillée car il est possible d'effacer d'autres variable sans le vouloir, ou causer des problèmes d'intégration, etc…

Lambda functions Pour déclarer des fonctions courtes efficacement, il est possible de déclarer une fonction `lambda` comme une variable. Ces fonctions sont anonymes et non répértoriées

[11]:
```
func = lambda x, y: (3 * x + y)
print(func(2, 3))
```

```
9
```

1.3 Lists

Les listes sont des collections de pointeurs vers des variables dans la mémoire système, par exemple, `lst = [0, "foo", ["a", 4]]` contient in pointeur vers `0` et `"foo"` (non-mutables) ainsi qu'un pointeur vers la liste (mutable) `["a", 4]`.

Si une copie de `lst` est faite (`lst1 = lst.copy()`) une modification aux variables non-mutables de style `lst[0] += 1` ne sera pas reproduite sur `lst1[0]`.
Alors que `lst[2] += [6]` modifiera la liste *référencée* par `lst` et `lst1`, changeant donc visuellement le contenu de `lst` et `lst1`.

List methods Les listes sont munies de plusieurs différentes méthodes.

Celles qui modifient la liste sur laquelle elles sont appliquées: - `lst.sort()` : Trie la liste - `lst.reverse()` : Inverse la liste `lst = lst[::-1]` - `lst.append(var)` : Ajoute var à la fin de lst `lst += [var]` - `lst.pop(index)` : Supprime et retourne l'élément à l'index spécifié - `lst.insert(index, var)` : Insère var à index `lst[index:index] = [var]` - `lst.remove(var)` : Supprime la première occurrence de var - `lst.extend(lst1)` : Ajoute tous les éléments d'une autre liste à la fin de la liste `lst = lst + lst1`

[12]:
```
lst = [1, 2, 3, 1, 2, 3]

lst.sort()
print(lst)
```

```
lst.reverse()
print(lst)

lst1 = lst.copy()

lst.append(8)
print(lst)

lst.pop(4)
print(lst)

lst.insert(4, 7)
print(lst)

lst.remove(7)
print(lst)

lst.extend(lst1)
print(lst)
```

```
[1, 1, 2, 2, 3, 3]
[3, 3, 2, 2, 1, 1]
[3, 3, 2, 2, 1, 1, 8]
[3, 3, 2, 2, 1, 8]
[3, 3, 2, 2, 7, 1, 8]
[3, 3, 2, 2, 1, 8]
[3, 3, 2, 2, 1, 8, 3, 3, 2, 2, 1, 1]
```

Celles qui retounent une liste ou valeur: - `count = lst.count(var)` : Conte le nobre d'occurences de var - `index = lst.index(var)` : Retourne l'index de la première occurence de var - `list = lst.copy()` : Retourne une copie superficielle de la liste - `list = lst.deepcopy()` : Du module copy, copie itérativement tout le contenu de lst

[13]:
```
lst = [1, 2, 3, 1, 2, 3]

print(lst.count(1))
print(lst.index(2))
print(lst.copy())
```

```
2
1
[1, 2, 3, 1, 2, 3]
```

Il est aussi possible de vérifier si un élément est dans une liste:

[14]:
```
lst = [1, 2, 3]
print(1 in lst)
```

```
True
```

List operations Il est possible d'utiliser des opértateurs mathématiques sur les listes: - `lst = lst1 + lst2` : crée une nouvelle liste composée des pointeurs de lst1 et lst2 - `lst = [lst1, lst` : cette liste ne contient que des pointeurs vers lst1 et lst2 - `lst = lst1 * 2` : lst = [lst1, lst1] - `lst1 = lst2` : crée un pointeur vers lst2

List comprehension La compréhension de liste permet de construire des listes sur une seule ligne. Plusieurs opérateurs sont permis: - `for` - `if`, `elif` ou `else`

[15]:
```
lst = [i * 4 for i in range(3)]
# Equivalent to:
# for i in range(3):
#     lst.append(i * 4)

lst1 = [i * 4 for i in lst if i % 2 == 0]
# Equivalent to:
# for i in lst:
#     if i % 2 == 0:
#         lst1.append(i * 4)

lst2 = [i + j for i in lst for j in lst1]
# Equivalent to:
# for i in lst:
#     for j in lst1:
#         lst2.append(i + j)
```

1.4 Dicts and Tuples

Manipulation de dictionnaires et tuples

Tuples Les tuples sont comme des listes, sauf qu'ils ne sont pas mutables. on peut aussi transformer n'importe quel objet itérable en un tuple `tuple()`
Toutes les méthodes qui ne modifient pas les listes peuvent donc être utilisés sur des tuples, par exemple: - `len()` - `count(var)` - `index(var)` - etc…

Il est possible de "unpack" des items d'un tuple:

[16]:
```
a = (2, 3)
b, c = a

print(b, c)
```

```
2 3
```

Dictionaries Ils agissent comme les listes et ne stockent que des pointeurs vers les valeurs stockées, les clés doivent être **uniques** et **non-mutables** car elles vont être hashées.
Il est aussi possible de créér un dictionnaire par la compréhension: `{i: j for i, j in enumerate(foo)}`

Il y a quelque méthodes associées aux dictionnaires: - `d.get(key)` : retourne la valeur assossiée à la clé `key`, `None` si la clé n'existe pas - `del d[key]` : supprime la paire située à `key` (aussi possible: `d.pop(key)`) - `d.values()` : itérable contenant toutes les valeurs de `d` - `d.keys()` : itérable contenant toutes les clés de `d` - `d.items()` : itérable contenant des tuples `(clé, valeur)`

1.5 Exceptions

Il faut distinguer les erreurs (`SyntaxError`) dues à une erreur de syntaxe, et toute autre exception, ici on parle des exceptions

---

Hierarchy Il existe toute une série d'exceptions qui sont classées sous forme de groupes et sous-groupes:
(Une exception comme `ModuleNotFoundError` est aussi une `ImportError`, une `Exception` et une `BaseException` par exemple) BaseException BaseExceptionGroup GeneratorExit KeyboardInterrupt SystemExit Exception ArithmeticError FloatingPointError OverflowError ZeroDivisionError AssertionError AttributeError BufferError EOFError ExceptionGroup [BaseExceptionGroup] ImportError ModuleNotFoundError LookupError IndexError KeyError MemoryError NameError UnboundLocalError OSError BlockingIOError ChildProcessError ConnectionError BrokenPipeError ConnectionAbortedError ConnectionRefusedError ConnectionResetError FileExistsError FileNotFoundError InterruptedError IsADirectoryError NotADirectoryError PermissionError ProcessLookupError TimeoutError ReferenceError RuntimeError NotImplementedError PythonFinalizationError RecursionError StopAsyncIteration StopIteration SyntaxError IndentationError TabError SystemError TypeError ValueError UnicodeError UnicodeDecodeError UnicodeEncodeError UnicodeTranslateError Warning BytesWarning DeprecationWarning EncodingWarning FutureWarning ImportWarning PendingDeprecationWarning ResourceWarning RuntimeWarning SyntaxWarning UnicodeWarning UserWarning

---

Try Pour gérer les exceptions, on utilise la syntaxe `try: ... except:`

[17]:
```python
try:
    '''Things to do that might fail'''
except IndexError: # or any other exception
    '''Do this if it fails'''
except (OtherException, TooPoorError, TryHarderError): # any tuple of exceptions
    '''Do smth else'''
```

```
else: # optionnal
    '''Do this if none of the except blocks were triggered'''
finally: # optionnal
    '''What to do after the try block'''
```

Le bloc `finally` est **toujours** exécuté même si un statement `return`, `exit()` ou autre est appelé par `except` ou `else`

Raise and `Exception()` Il est possible de créér une exception non-vanilla en utilisant `raise` dans le bloc `try`

[18]:
```
try:
    '''do stuff here'''
    something_went_wrong = True

    if something_went_wrong:
        raise Exception("Description: the user is stoopid")
except Exception:
    '''what to do if Exception'''
```

Il est aussi possible de récupérer la description de n'importe quelle exception:

[19]:
```
try:
    '''do stuff here'''
    something_went_wrong = True

    if something_went_wrong:
        raise Exception("The user is stoopid")
except Exception as string:
    print(string)
```

```
The user is stoopid
```

1.6 Numpy and Matplotlib

Introduction à numpy et matplotlib

Numpy "The fundamental package for scientific computing with Python", utilisé pour toute sorte d'opérations mathématiques avec des variables

[20]:
```
import numpy as np
```

Fondamentalement, numpy introduit un nouveau type de données: `ndarray`, qui se comporte comme un vecteur/matrice, on peut par exemple appliquer n'importe quelle fonction sur tous les éléments d'un array d'un coup

[21]:
```
arr = np.array([1, 2, 3, 4])
square = lambda x: x**2
print(square(arr))
```

```
[ 1  4  9 16]
```

Noter que la taille d'un `ndarray` est non-mutable et doit être définie à sa création! De plus, un array va se régler au type le plus général dans l'itérable utilisé pour sa création

[22]:
```
print(np.array([1, 2, "3"]))
```

```
['1' '2' '3']
```

One-dimensional arrays Il y a plusieurs méthodes pour créer des vecteurs à une dimension (une ligne)

[23]:
```
dim = 6
val = "aaa"
print(np.zeros(dim))
print(np.ones(dim))
print(np.full(dim, val))
print(np.empty(dim)) # valeurs aléatoires
print(np.linspace(start=0, stop=5, num=4)) # distibution régulière ~ "range"
```

```
[0. 0. 0. 0. 0. 0.]
[1. 1. 1. 1. 1. 1.]
['aaa' 'aaa' 'aaa' 'aaa' 'aaa' 'aaa']
[1. 1. 1. 1. 1. 1.]
[0.         1.66666667 3.33333333 5.        ]
```

Multi-dimensional arrays Noter que `dim` peut être autre chose qu'un simple `int`

[24]:
```
dim = [2, 3, 2]
print(np.zeros(dim))
```

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

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

Il est possible d'adresser différentes parties d'un array via du "slicing étendu"

[25]:
```
arr = np.array([["a", "b", "c"], ["d", "e", "f"], ["g", "h", "i"]])
print(arr)
```

```
print()
print(arr[1:, :]) # lignes 3 et 4
print()
print(arr[:, 1]) # deuxièmme colonne
```

```
[['a' 'b' 'c']
 ['d' 'e' 'f']
 ['g' 'h' 'i']]

[['d' 'e' 'f']
 ['g' 'h' 'i']]

['b' 'e' 'h']
```

On peut connaître le nombre de dimensions (“profondeur”) ou la forme d’un array

[26]:
```
print(arr.ndim)
print(arr.shape)
```

```
2
(3, 3)
```

Et même créer des matrices identités

[27]:
```
print(np.eye(3))
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

Array operations Comme vu au dessus, les opérations simultanées sur tous les éléments sont très faciles avec numpy
Pour effectuer des opérations comme un produit matriciel, il faut utiliser les méthodes de `np.linalg`

[28]:
```
matrix = np.eye(3)
print(np.linalg.det(matrix)) # détérminant
print(np.linalg.eigvals(matrix)) # valeurs propres
```

```
1.0
[1. 1. 1.]
```

Il est aussi possible d’effectuer des opérations booléennes

[29]:
```
arr = np.random.rand(4, 4)
print(arr)
print()
print(arr > 0.3)
```

```
[[0.20456773 0.60157433 0.27216678 0.41474678]
 [0.84289226 0.06649502 0.83801668 0.35078642]
```

```
 [0.75210005 0.89619871 0.20776613 0.10865761]
 [0.01356904 0.79033208 0.54444792 0.41447216]]

[[False  True False  True]
 [ True False  True  True]
 [ True  True False False]
 [False  True  True  True]]
```

[30]:
```
rounded = np.empty_like(arr)
rounded[arr >= 0.5] = 1; rounded[arr < 0.5] = 0
print(rounded)
```

```
[[0. 1. 0. 0.]
 [1. 0. 1. 0.]
 [1. 1. 0. 0.]
 [0. 1. 1. 0.]]
```

On peut **reshape** des arrays si le nombre d'éléments reste le même

[31]:
```
arr = np.reshape(arr, [2, 2, 4]) # noter que 4x4 = 16 = 2x2x4
print(arr)
```

```
[[[0.20456773 0.60157433 0.27216678 0.41474678]
  [0.84289226 0.06649502 0.83801668 0.35078642]]

 [[0.75210005 0.89619871 0.20776613 0.10865761]
  [0.01356904 0.79033208 0.54444792 0.41447216]]]
```

Enfin, on peut concaténer des arrays entre eux

[32]:
```
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])
print(np.concatenate((x, y)))
```

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

[33]:
```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])
print(np.concatenate((a, b), axis=0))
print(np.concatenate((a, b), axis=1))
```

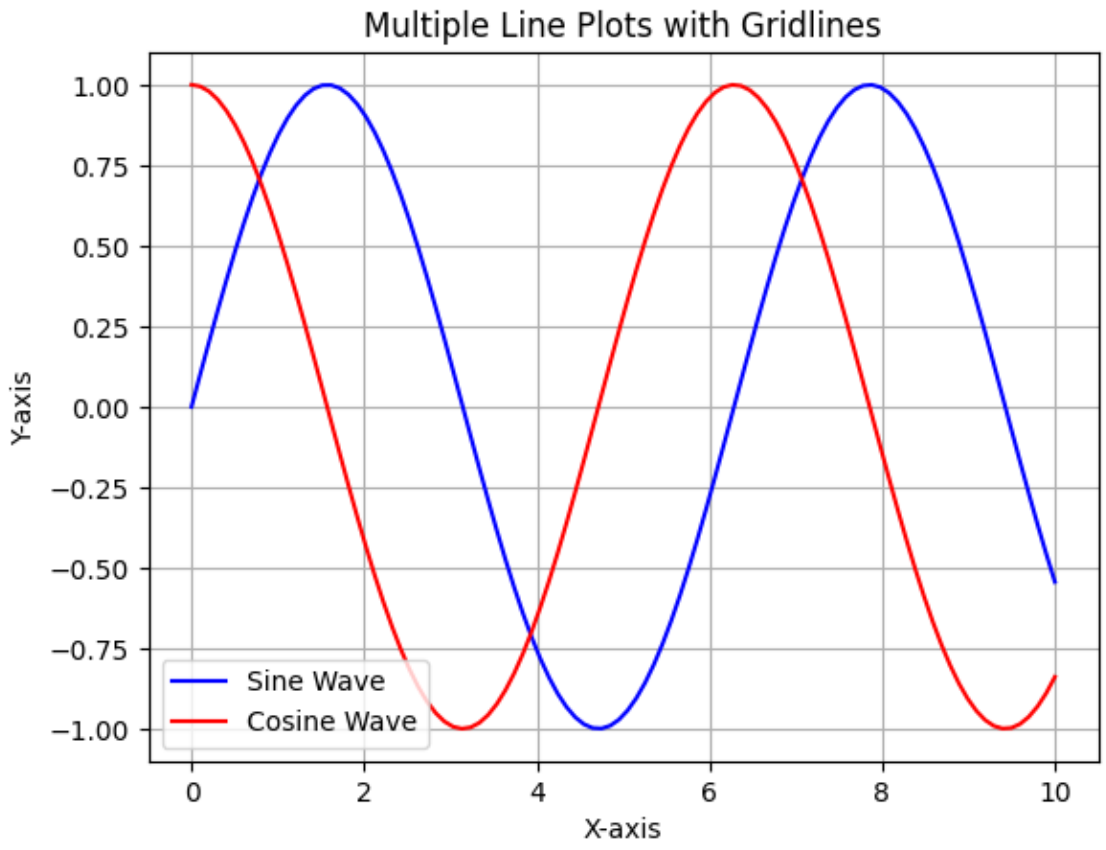
```
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
[[1 2 5 6]
 [3 4 7 8]]
```

Matplotlib "Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. Matplotlib makes easy things easy and hard things possible."

[34]:
```
import matplotlib.pyplot as plt
```

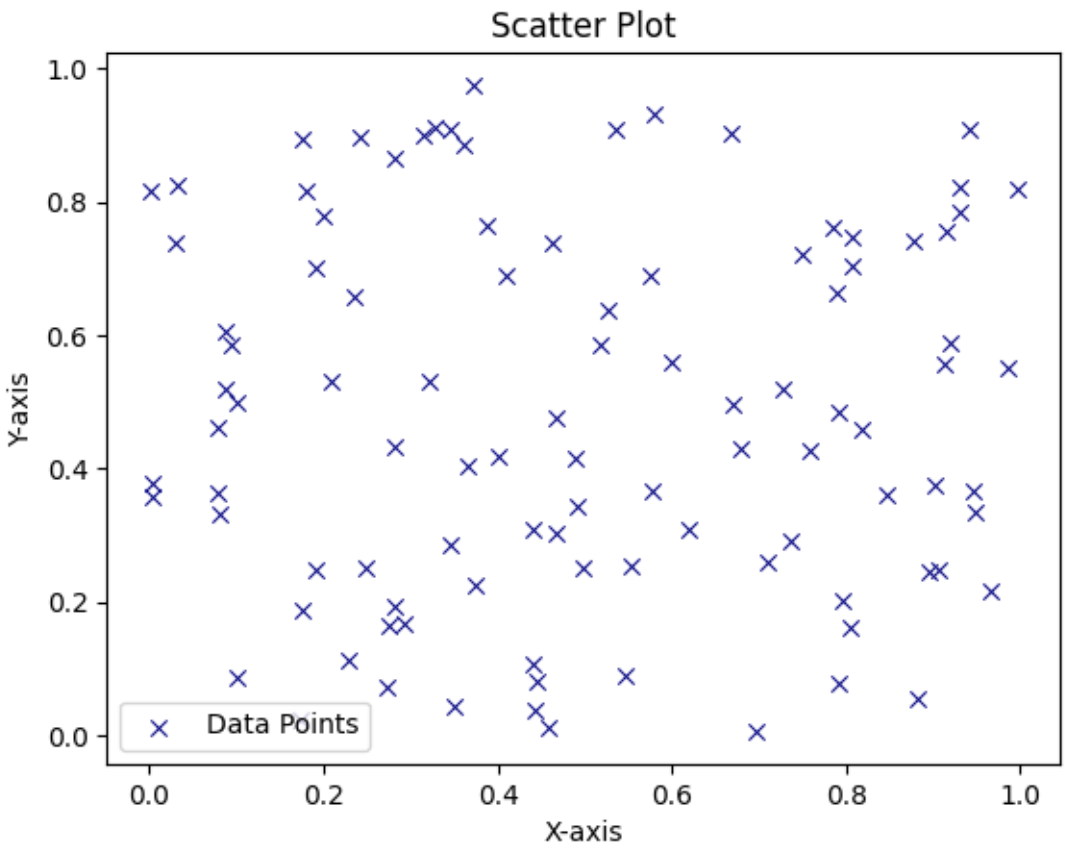
Multiple line plot

[35]:
```
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)

plt.figure()
plt.plot(x, y1, label='Sine Wave', color='blue')
plt.plot(x, y2, label='Cosine Wave', color='red')

# Add title and labels
plt.title('Multiple Line Plots with Gridlines')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')

# Display legend
plt.legend()

# Enable gridlines
plt.grid()

# Show the plot
plt.show()
```



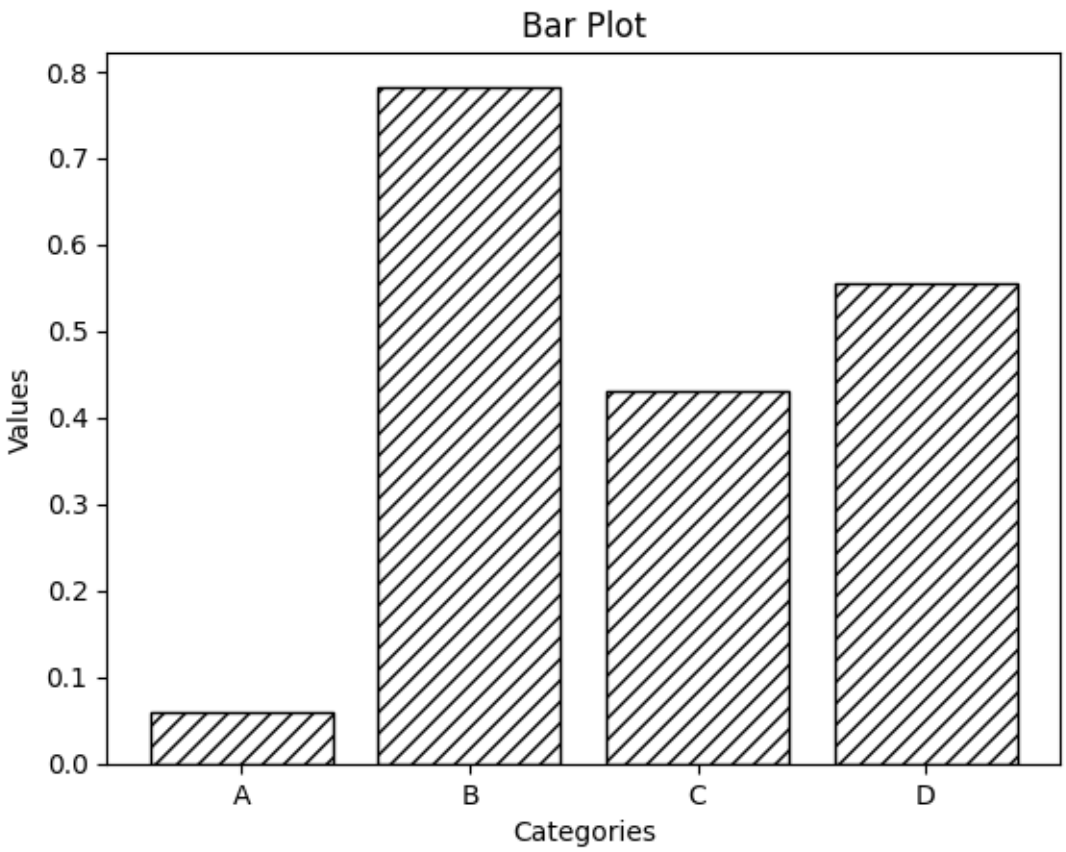
Scatter plot

[36]:
```
x = np.random.rand(100)
y = np.random.rand(100)

plt.figure()
plt.scatter(x, y, color='navy', marker='x', linewidth=0.75, label='Data Points')

# Add title and labels
plt.title('Scatter Plot')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')

# Display legend
plt.legend(loc='lower left')

# Show the plot
plt.show()
```



Histogram plot

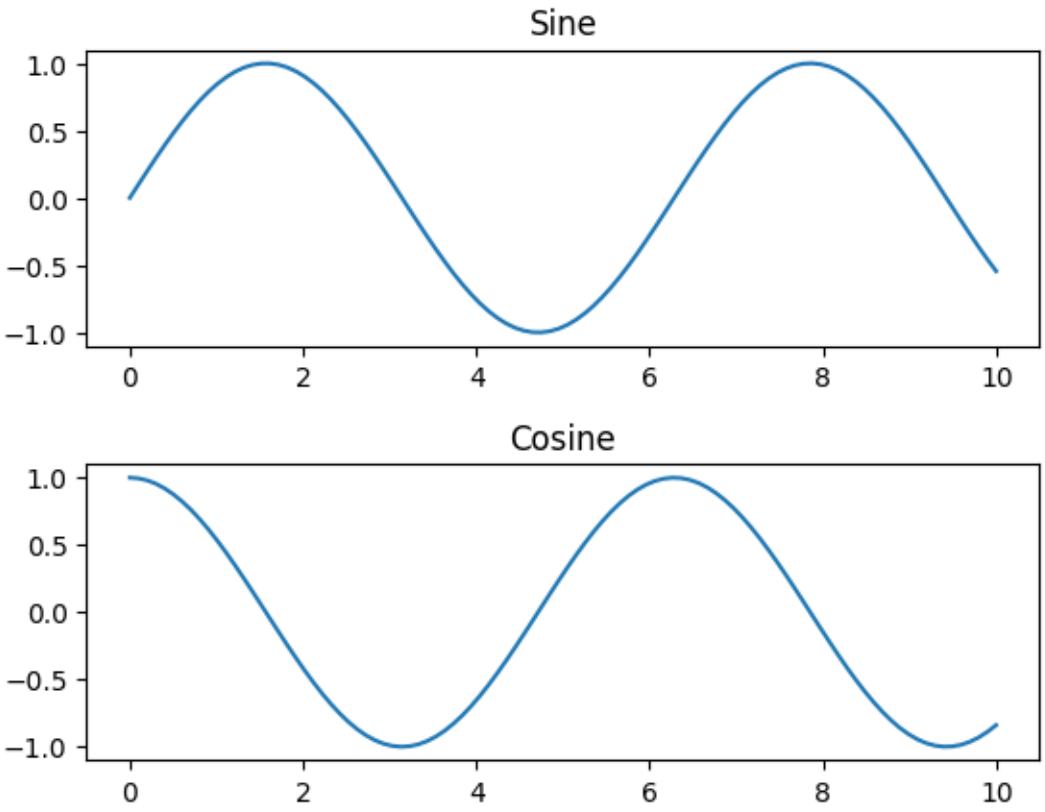
[37]:
```
categories = ['A', 'B', 'C', 'D']
values = np.random.rand(4)

plt.figure()
plt.bar(categories, values, color='none', edgecolor='black', hatch='///')

# Add title and labels
plt.title('Bar Plot')
plt.xlabel('Categories')
plt.ylabel('Values')

# Display the plot
plt.show()
```



Subplots

[38]:
```
x = np.linspace(0, 10, 100)

# Create a figure with two subplots
fig = plt.figure()

# First subplot (top-left corner)
plt.subplot(2, 1, 1)
plt.plot(x, np.sin(x))
plt.title('Sine')

# Second subplot (bottom-left corner)
plt.subplot(2, 1, 2)
plt.plot(x, np.cos(x))
plt.title('Cosine')

fig.subplots_adjust(wspace=0.3, hspace=0.4)

# Show the plot
```

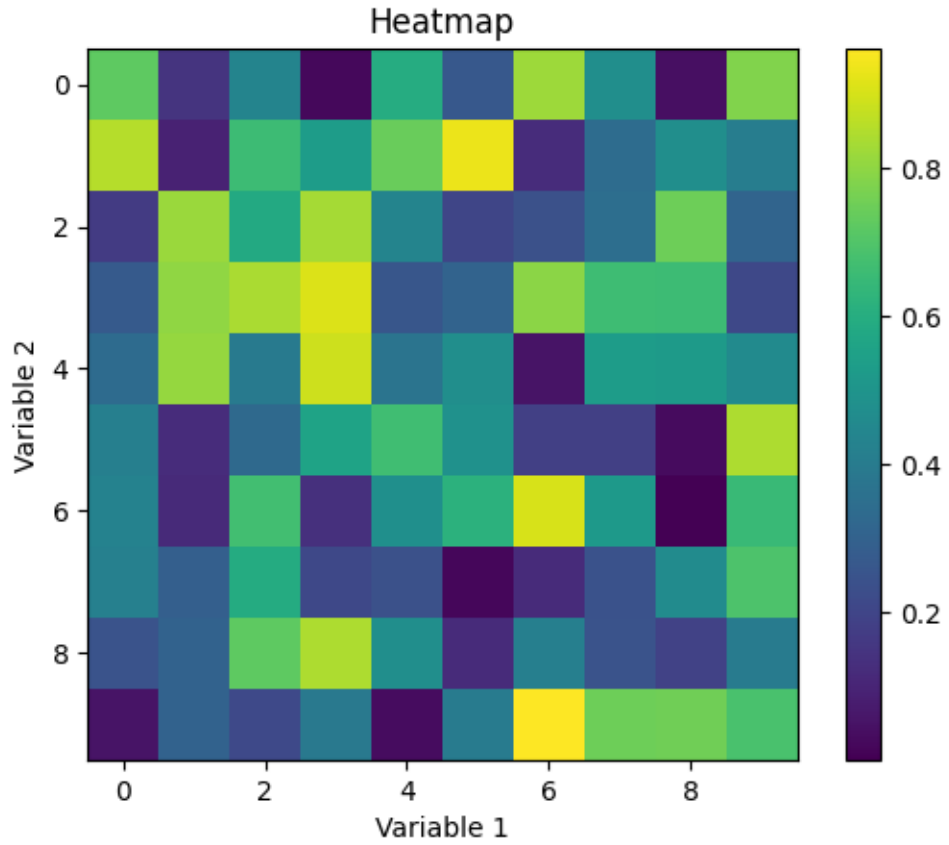
```
plt.show()
```



Heatmap

[39]:
```
import matplotlib.pyplot as plt
import numpy as np

# Sample data: A 2D array where each row and column is indexed by variables,␣
 ↪and the value of the array represents the intensity
data = np.random.rand(10, 10)

# Create a figure and axis
fig, ax = plt.subplots()

# Create the heatmap
cax = ax.imshow(data, cmap='viridis', interpolation='nearest')

# Add colorbar to represent values
fig.colorbar(cax)

# Add titles and labels
```

```
ax.set_title('Heatmap')
ax.set_xlabel('Variable 1')
ax.set_ylabel('Variable 2')

# Display the heatmap
plt.show()
```

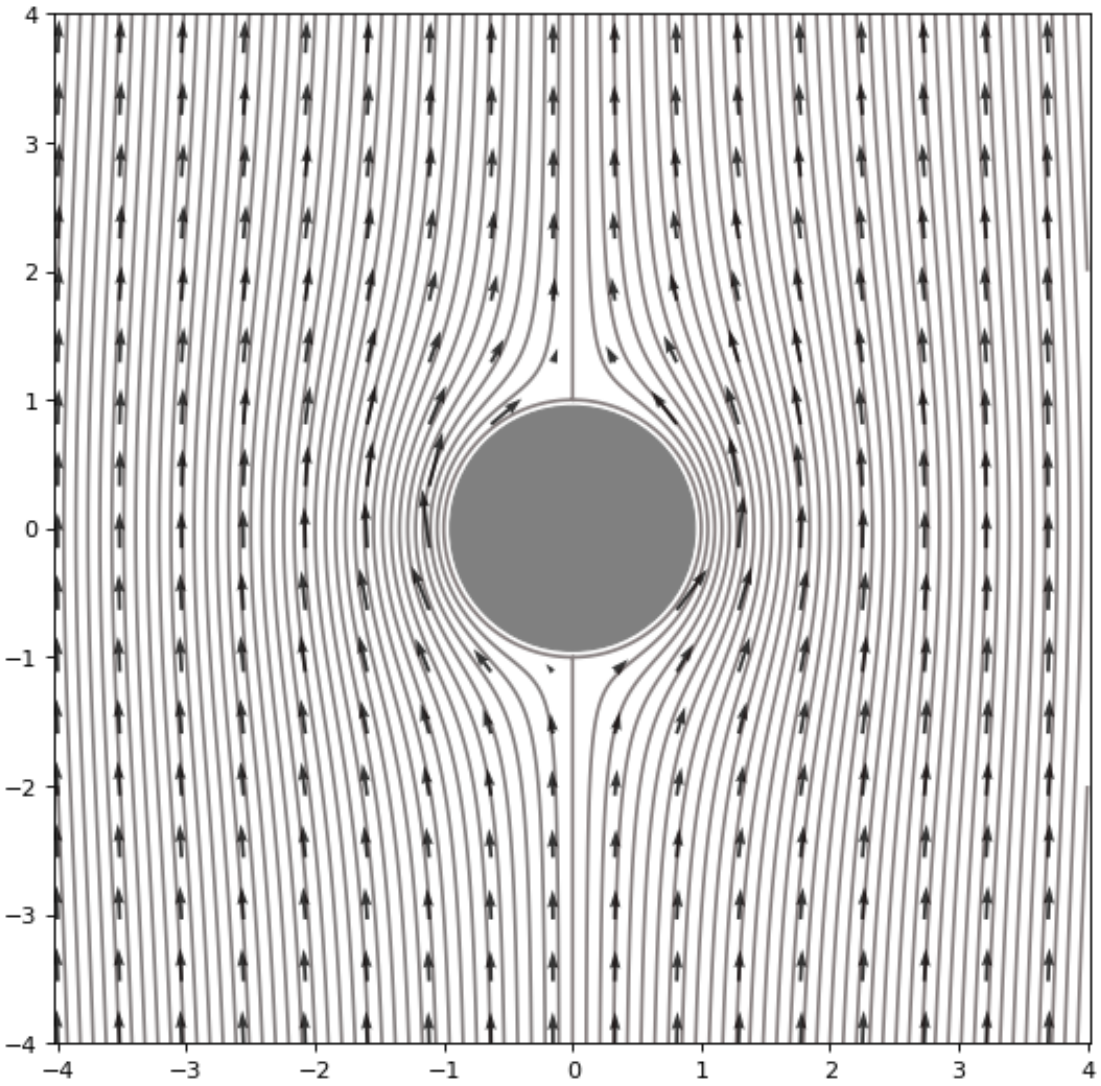


Advanced usage

[40]:
```
from matplotlib import colors

# variables 1
v_density = 30
l_density = 80
p_density = 80
d_density = 500
c_pos = (1, 1)
c_radius = 0.95

# variables 2
```

```
l_viscosity = 148.1408929
rho_a = 7850
rho_g = 1260
radius = 0.0005
pi = np.pi
g = 9.81
sum_f = (
    (
        -
        (rho_g * 2 * (4/3) * pi * radius**3 * g)
        -
        (6 * pi * radius * l_viscosity)
        +
        (rho_a * (4/3) * 2 * pi * radius**3 * g)
    )
)

# main
def flow(psi, mask = None, x=np.linspace(-4, 4, d_density), y=np.linspace(-4,␣
 ↪4, d_density), h=1e-10):
    x_axis, y_axis = np.meshgrid(x, y)
    if mask:
        x_axis, y_axis = (
            np.ma.masked_where(mask(x_axis, y_axis), x_axis),
            np.ma.masked_where(mask(x_axis, y_axis), y_axis)
        )

    # complex analysis navier-stokes
    u_axis = - (psi(x_axis, y_axis + h) - psi(x_axis, y_axis - h)) / (2 * h)
    v_axis = (psi(x_axis + h, y_axis) - psi(x_axis - h, y_axis)) / (2 * h)

    plt.figure(figsize=(8, 8), dpi=p_density)
    plt.quiver(
        x_axis[::v_density, ::v_density],
        y_axis[::v_density, ::v_density],
        u_axis[::v_density, ::v_density],
        v_axis[::v_density, ::v_density],
        alpha=0.8
    )
    cmap = colors.ListedColormap(['black', 'black'])
    bounds = [0, 0, 0]
    norm = colors.BoundaryNorm(bounds, cmap.N)
    plt.contour(
        x_axis,
        y_axis,
        psi(x_axis, y_axis),
        levels=l_density,
```

```
        cmap='hot',
        norm=norm,
        alpha=0.5
    )

    plt.axis('equal')

    centre = plt.Circle((0, 0), 0.95, color='gray')
    plt.gca().add_patch(centre)

# flow lines functions
ball = lambda y, x : y - y / (x**2 + y**2)
ball_mask = lambda y, x : x**2 + y**2 < 0.95
flow(ball, ball_mask)

plt.show()
```



1.7 Useful tricks

De multiples manières d'accélérer l'écriture de certaines opérations

---

Walrus operator Assigne une valeur à l'intérieur d'une expression

[41]:
```
numbers = [1, 2, 3, 4, 5]

while (n := len(numbers)) > 0:
    print(numbers.pop())
```

```
5
4
3
2
1
```

Ternary operator Comme pour la compréhension de listes, permet l'utilisation de `if-else` sur une ligne

[42]:
```
status = "Adult" if (age := 20) >= 18 else "Minor"
print(status)
```

```
Adult
```

Set & Dict comprehension Comme pour les listes, permet de créer des dictionnaires et ensembles en une ligne

[43]:
```
num_set = {x for x in [1, 2, 2, 3]}  # {1, 2, 3}
squared_dict = {x:x**2 for x in range(3)}  # {0: 0, 1: 1, 2: 4}
```

Variable unpacking Il est possible d'assigner des valeurs à plusieurs variables à la fois

[44]:
```
a, b, c = [1, 2, 3]
print(a, b, c)
```

```
1 2 3
```

[45]:
```
first, *middle, last = [1, 2, 3, 4, 5]
print(first, middle, last)
```

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

Et d'échanger des variables

[46]:
```
x, y = 10, 20
x, y = y, x
print(x, y)
```

```
20 10
```

Ainsi que d'assigner des valeurs par défaut

[47]:
```
user_input = ""
name = user_input or "Guest"
print(name)
```

```
Guest
```

Iterable tricks `zip()` permet de combiner deux itérables élément par élément

[48]:
```
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']
zipped = zip(list1, list2)
print(list(zipped))
```

```
[(1, 'a'), (2, 'b'), (3, 'c')]
```

`enumerate()` retourne une liste qui associe chaque élément d'une liste à son index dans un tuple

[49]:
```
names = ['Alice', 'Bob', 'Charlie']
for idx, name in enumerate(names):
    print(idx, name)
```

```
0 Alice
1 Bob
2 Charlie
```

`all()` vérifie si tous les éléments d'un set sont `True`

[50]:
```
numbers = [2, 4, 6, 8]
print(all(x % 2 == 0 for x in numbers))
```

```
True
```

`any()` vérifie si au moins un des éléments d'une liste est `True`

[51]:
```
numbers = [0, 1, 2, 3]
print(any(x % 2 == 0 for x in numbers))
```

```
True
```

`map()` applique une fonction à chaque élément d'une liste, peut prendre une lambda function comme argument aussi

[52]:
```
numbers = [1, 2, 3, 4]
squared = map(lambda x: x ** 2, numbers)
print(list(squared))
```

```
[1, 4, 9, 16]
```

`filter()` applique une fonction à chaque élément et retourne une liste qui ne contient que les éléments pour lequels la fonction retourne `True`

[53]:
```
numbers = [1, 2, 3, 4, 5]
even_numbers = filter(lambda x: x % 2 == 0, numbers)
print(list(even_numbers))
```

```
[2, 4]
```

`sorted()` trie et retourne une liste

[54]:
```
unsorted_list = [3, 1, 4, 5, 2]
print(sorted(unsorted_list))
```

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

`reversed()` inverse une liste, équivaut à `lst[::-1]`

[55]:
```
seq = [1, 2, 3, 4, 5]
print(list(reversed(seq)))
```

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

`reduce()` applique une fonction arbitraire au deux premiers éléments d'une liste, et les remplace par l'output de la fontion jusqu'à avoir condensé la liste originale à un seul élément

[56]:
```
from functools import reduce
numbers = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, numbers)
print(product)
```

```
24
```

---

2 Algorithmics

Un *algorithme* est une série d'étapes logiques qui établissent une procédure réalisant un objectif, un *programme* est une *implémentation* d'un algorithme

---

2.1 Evaluation criteria

Il existe plusieurs critères pour évaluer la qualité d'un algorithme: - Précision - Performance - Coût - …

2.2 Time complexity

Un programme prend un certain temps à exécuter, ce temps dépend de la performance du programme ainsi que du nombre d'itérations ou d'opérations qu'il doit effectuer
La relation entre le temps pris et le nombre `n` d'opération, est appelé le complexité temporelle

Si un programme évolue en un temps $T(n)$ (où `n` est soit l'input size, le nombre d'itérations), il est possible de majorer et minorer cette fonction $T(n)$
C'est à dire: "$T(n)$ ne prendra jamais plus/moins de temps que…"
Cela permet de caractériser le *comportement asymptotique* de $T(n)$

Big $\mathcal{O}(\cdot)$ Soient $n \in \mathbb{N}$ et $g(n)$, une fonction positive de n, on dit que $T(n)$ est $\mathcal{O}(g(n))$, si:
$\exists\, C > 0 \in \mathbb{R}$ et $N > 0 \in \mathbb{N}$ tels que $\forall\ n > N$, $T(n) \leq C \cdot g(n)$

Ici, $g(n)$ majore $T(n)$

Big $\Omega(\cdot)$ Soient $n \in \mathbb{N}$ et $g(n)$, une fonction positive de n, on dit que $T(n)$ est $\Omega(g(n))$, si:
$\exists\, C > 0 \in \mathbb{R}$ et $N > 0 \in \mathbb{N}$ tels que $\forall\ n > N$, $T(n) \geq C \cdot g(n)$

Ici, $g(n)$ minore $T(n)$

Big $\Theta(\cdot)$ Si $T(n)$ est à la fois $\mathcal{O}(g(n))$ et $\Omega(g(n))$, on dit alors qu'elle est $\Theta(g(n))$

La progression temporelle de l'implémentation evolue exactement comme $g(n)$

What is actually used En pratique, il est souvent trop compliqué de définir une limite inférieure $\Omega(g(n))$ à un algorithme (on ne sait souvent pas si on peut faire mieux entre autres), et il est surtout plus utile de donner un "worst case scenario", donc il est conventionnel de caractériser le temps d'évolution avec une borne supérieure le plus proche $\mathcal{O}(g(n))$

Evaluating time complexitiy of actual code Pour évaluer la complexité temporelle d'un programme, il suffit d'évaluer la complexité temporelle de chaque ligne du code, puis de combiner les complexités de chaque instructions (pour ce qui nous concerne, il suffit de prendre un polynôme du plus grand degré de `n`)

1. **Constant time -** $\mathcal{O}(1)$

[57]:
```python
def constant_time():
    return True
```

2. **Linear Time -** $\mathcal{O}(n)$

[58]:
```
def linear_time(arr):
    for i in arr:
        print(i)
```

3. **Logarithmic Time Complexity -** $\mathcal{O}(\log n)$

[59]:
```
def logarithmic_time(n):
    return int(math.log2(n))
```

4. **Quadratic Time Complexity -** $\mathcal{O}(n^2)$

[60]:
```
def quadratic_time(arr):
    for i in arr:
        for j in arr:
            print(i, j)
```

5. **Cubic Time Complexity -** $\mathcal{O}(n^3)$

[61]:
```
def cubic_time(arr):
    for i in arr:
        for j in arr:
            for k in arr:
                print(i, j, k)
```

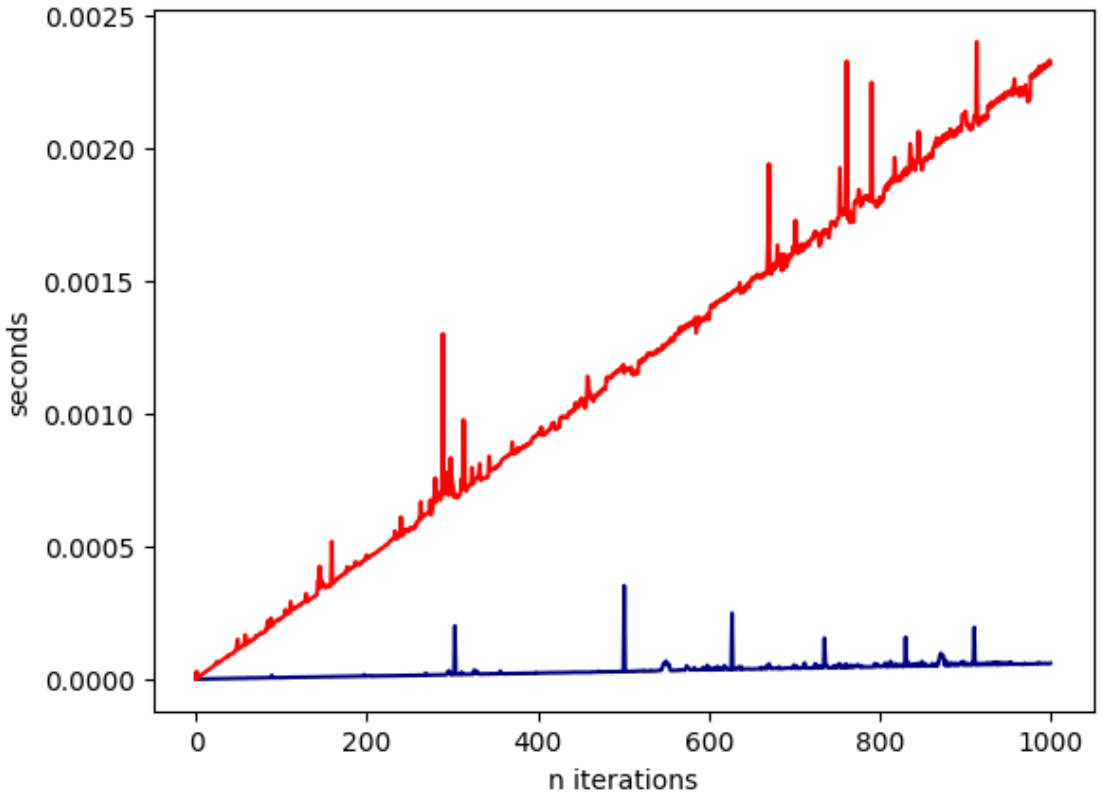
Un bout de code contenant tous ces morceaux de code aurait donc une complexité de $\mathcal{O}(n^3)$

Some more examples `append` est de type $\mathcal{O}(n)$, on voit que l'implémentation de numpy prend plus de temps (toujours $\mathcal{O}(n)$) car l'array doit être aggrandi à chaque fois:

[62]:
```
import time as tm

ntest = 1000
time_array = np.empty(ntest)
time_array_np = np.empty_like(time_array)

# Python lst.append()
for i in range(ntest):
    lst = []
    a = tm.time()
    for k in range(i):
        lst.append(k)
    b = tm.time()
    time_array[i] = b - a

# Numpy np.append()
for i in range(ntest):
```

```
    arr = np.empty(0)
    a = tm.time()
    for k in range(i):
        np.append(arr, k)
    b = tm.time()
    time_array_np[i] = b - a
```

[63]:
```
plt.plot(time_array, color='navy')
plt.plot(time_array_np, color='red')
plt.xlabel('n iterations')
plt.ylabel('seconds')
plt.show()
```

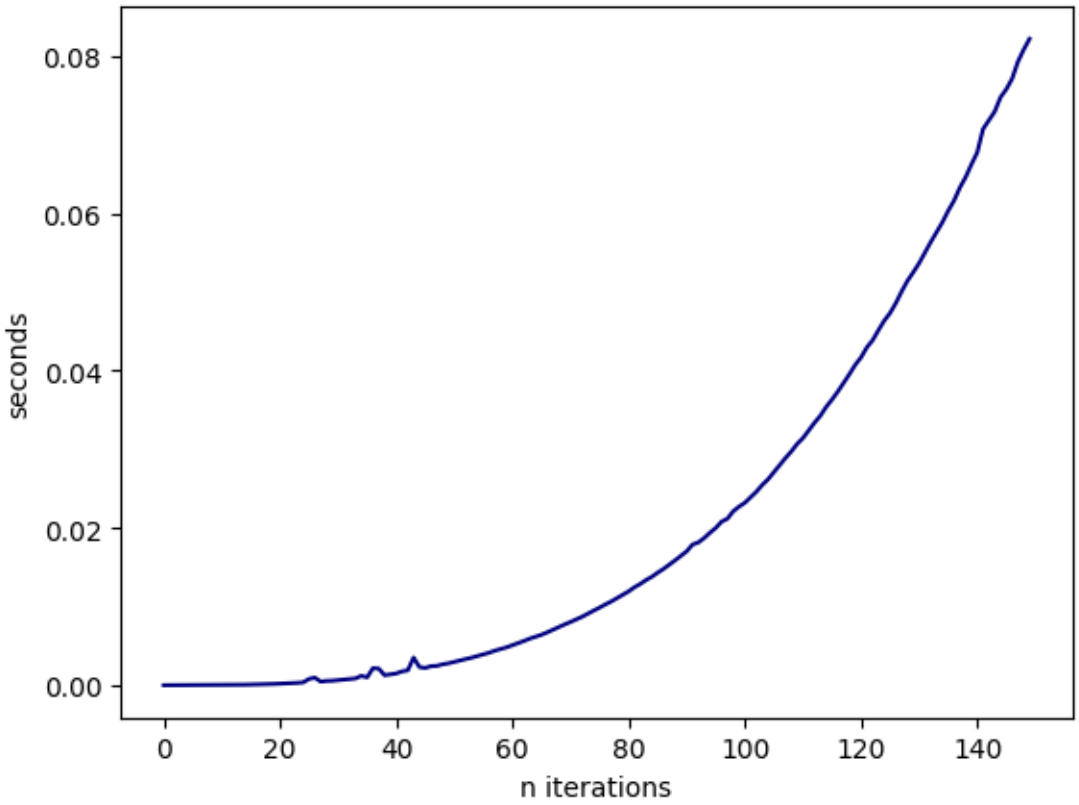


Sommer des cubes dans trois boucles a une complexité de $\mathcal{O}(n^3)$:

[64]:
```
ntest = 150
time_array_cubic = np.empty(ntest)

# Cubic Time Complexity function: sum of squares using nested loops
for i in range(ntest):
    a = tm.time()
```

```
    sum_of_squares = 0
    for j in range(i):
        for k in range(j):
            for l in range(k):
                sum_of_squares += (j ** 2)
    b = tm.time()
    time_array_cubic[i] = b - a
```

[65]:
```
plt.plot(time_array_cubic, color='navy')
plt.xlabel('n iterations')
plt.ylabel('seconds')
plt.show()
```



2.3 Searching and sorting

Regardons quelques algorithmes

List search On veut une fonction qui retourne l'indice `i` de la première occurence d'un élément `x` si la liste le contient, `None` sinon

[66]:
```python
def list_search(lst, x):
    length = len(lst)

    for index in range(length):
        if x == lst[index]:
            return index

    return None
```

[67]:
```python
list_search(sorted([1, 5, 8, 9, 3, 2, 4, 5]), 8)
```

[67]: 6

Cette fonction est de complexité temporelle $\mathcal{O}(n)$

Dichotomic search (Binary search) Pour trouver un certain elément dans un ensemble, on: 1. Vérifie si il est dans l'ensemble 2. Si oui, on divise l'ensemble en deux 3. On continue l'opération avec le demi-ensemble conteneant l'élément 4. On finit par arriver à l'élément lui-même

Note: on peut diviser différement qu'en deux (en 3, 4, ...)

Attention! la liste en entrée doit avoir été triée préalablement pour pouvoir détérminer où x se trouve

[68]:
```python
def binary_search(lst, x):
    length = len(lst)
    lower_bound = 0
    upper_bound = length - 1

    while upper_bound >= lower_bound:
        middle = (upper_bound + lower_bound) // 2

        if lst[middle] == x:
            return middle
        elif lst[middle] > x:
            upper_bound = middle - 1
        else:
            lower_bound = middle + 1

    return None
```

[69]:
```python
binary_search(sorted([1, 5, 8, 9, 3, 2, 4, 5]), 8)
```

[69]: 6

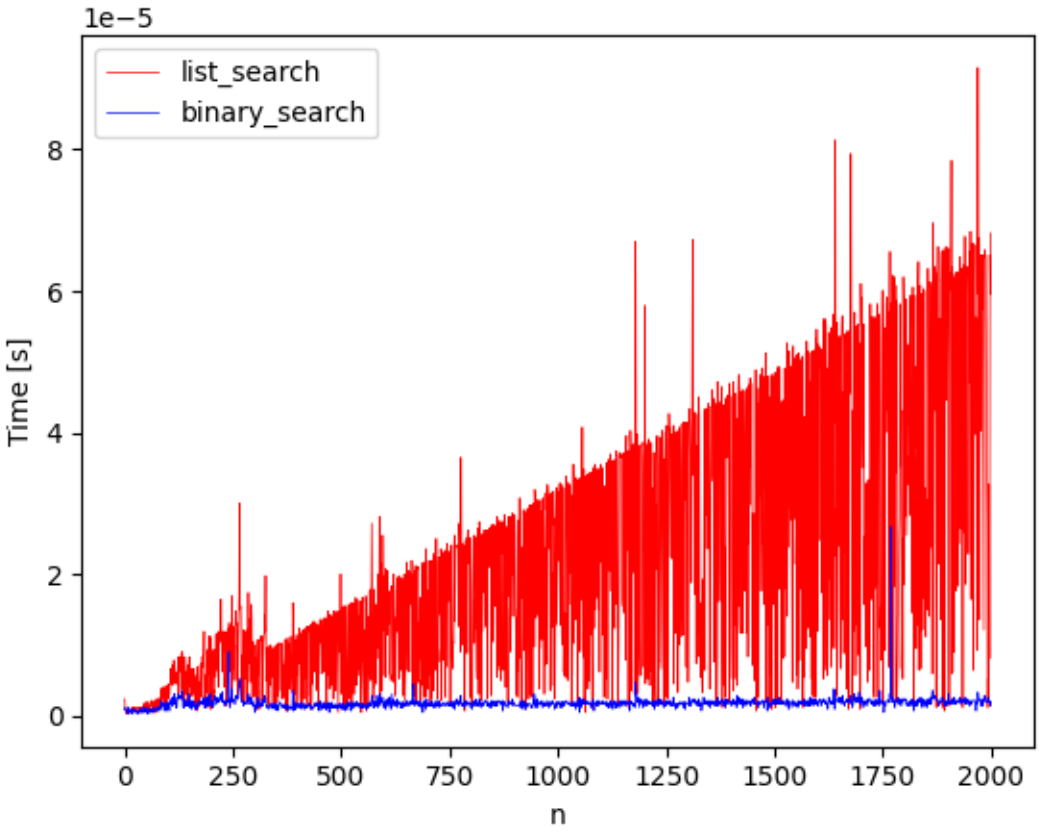
Cette fonction est de complexité temporelle $\mathcal{O}(\log_2 n)$, sachant qu'il est possible de trier une liste en minimum $\mathcal{O}(n \cdot \log n)$, cette méthode n'a aucun avantage si les listes ne sont pas déjà triées

---

Search times Regardons expérimentalement les complexités temporelles de ces deux algorithmes

[70]:
```
import random as rd

n_tests = 2000

list_search_times = []
binary_search_times = []

for n in range(n_tests):
    temp_list = sorted([rd.randint(0, n) for _ in range(n)])
    random_find = rd.randint(0, n)

    # list_search
    a = tm.time()
    list_search(temp_list, random_find)
    b = tm.time()

    list_search_times.append(b - a)

    # binary_search
    a = tm.time()
    binary_search(temp_list, random_find)
    b = tm.time()

    binary_search_times.append(b - a)
```

[71]:
```
plt.plot(list_search_times, color='red', label='list_search', linewidth=0.5)
plt.plot(binary_search_times, color='blue', label='binary_search', linewidth=0.
↪5)
plt.legend()
plt.ylabel('Time [s]')
plt.xlabel('n')
plt.show()
```



On voit clairement que la recherche binaire performe beaucoup mieux!

Selection sort Cet algorithme procède ainsi: 1. Trouver le plus petit élément de la liste 2. Le placer au début 3. Trouver le second plus petit élément 4. etc...

[72]:
```
def selection_sort(lst):
    length = len(lst)

    for i in range(length):
        m = lst[i]
        m_index = i
        for j in range(i + 1, length):
            if lst[j] < m:
                m = lst[j]
                m_index = j
                lst[i], lst[m_index] = lst[m_index], lst[i]
    return lst
```

[73]:
```
selection_sort([1, 5, 8, 9, 3, 2, 4, 5])
```

[73]: [1, 2, 3, 4, 5, 5, 8, 9]

Cette fonction est de complexité temporelle $\mathcal{O}(n^2)$

Insertion sort

[74]:
```
def insertion_sort(lst):
    length = len(lst)
    for i in range(length):
        j = i
        while j > 0 and lst[j] < lst[j-1]:
            lst[j], lst[j-1] = lst[j-1], lst[j]
            j -= 1
    return lst
```

[75]:
```
insertion_sort([1, 5, 8, 9, 3, 2, 4, 5])
```

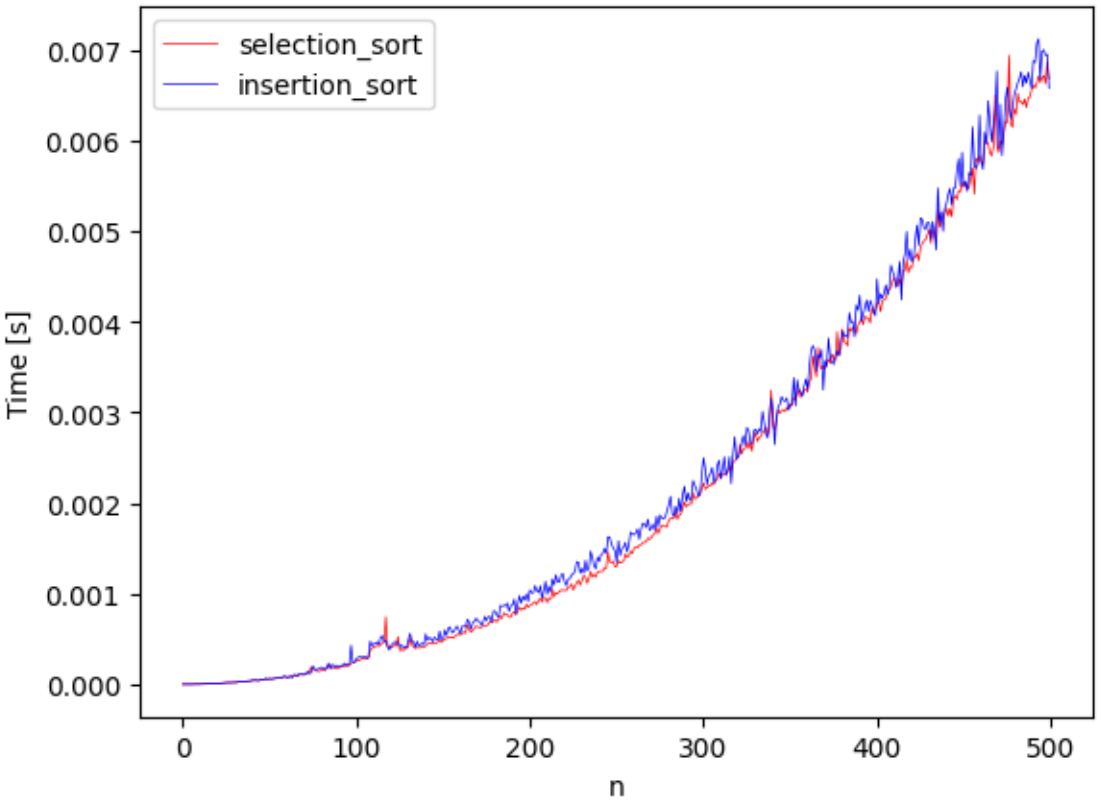
[75]: [1, 2, 3, 4, 5, 5, 8, 9]

Cette fonction est de complexité temporelle $\mathcal{O}(n^2)$ aussi (dans le pire des cas)

Sort times Regardons expérimentalement les complexités temporelles de ces deux algorithmes de tri

[76]:
```
n_tests = 500

selection_sort_times = []
insertion_sort_times = []

for n in range(n_tests):
    temp_list1 = [rd.randint(0, n) for _ in range(n)]
    temp_list2 = temp_list1.copy()

    # selection_sort
    a = tm.time()
    selection_sort(temp_list1)
    b = tm.time()

    selection_sort_times.append(b - a)

    # insertion_sort
    a = tm.time()
    insertion_sort(temp_list2)
    b = tm.time()

    insertion_sort_times.append(b - a)
```

[77]:
```
plt.plot(selection_sort_times, color='red', label='selection_sort', linewidth=0.
↪5)
plt.plot(insertion_sort_times, color='blue', label='insertion_sort',␣
↪linewidth=0.5)
plt.legend()
plt.ylabel('Time [s]')
plt.xlabel('n')
plt.show()
```



2.4 Numerical root methods

Pour résoudre certaines équations très complexes, il nous faut des méthodes d'approximation des solutions. On va essayer de construire une suite $(a_n)_{n\geq 0}$ telle que $\lim_{n\to\infty} a_n = \alpha$ où α est le zéro recherché.

Bien sûr, un ordinateur ne pourra pas effectuer une infinité d'itération, donc on n'obtiendra jamais une solution exacte

---

Numerical error On définit l'erreur absolue à la k-ième itération comme:
$e_{abs,k} = |a_k - \alpha|$

Et l'erreur relative:
$e_{rel,k} = \frac{e_{abs,k}}{a_k}$

Bissection Une méthode de résolution numérique est la méthode de bissection, et elle procède comme cela: 1. Choix d'un intervalle de départ $[a, b]$ tel que la condition de Bolzano soit satisfaite $(f(a) \cdot f(b) < 0)$ 2. Trouver le point milieu $a_k = \frac{a+b}{2}$ 3. Dans vérifier lequel deux demi-intervalles satisfait la condition de Bolzano 4. Répéter tant que une condition d'arrêt n'est pas satisfaite (nombre d'itérations limite, ou tolérance ϵ)

Cette méthode a des désavantages: 1. Convergence lente 2. Possibilité de perdre des solutions 3. À chaque intération, il est possible de diviser l'intervalle plus inteligemment que par deux pour optimiser la convergence

Minimal iterations Le nombre minimal d'itérations, pour une tolérance donnée ϵ est:

$$k_{min} = \log_2 \left(\frac{|b-a|}{\epsilon} \right) - 1$$

Newton's method **Rappel (Séries de Taylor):** Il est possible d'approximer n'importe quelle fonction dérivable par un polynôme de degré n, la formule de Taylor nous donne le développement limité d'une fonction $f : I \to \mathbb{R}$ autour de $\alpha \in I$:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(\alpha)}{n!}(x-\alpha)^n = f(\alpha) + \frac{f'(\alpha)}{1!}(x-\alpha) + \frac{f''(x)}{2!}(x-\alpha)^2 + ...$$

Considérons le développement limité du premier ordre autour de α:

$$f(x) \approx f(\alpha) + f'(\alpha) \cdot (x - \alpha)$$

On en tire (après quelque détails analytiques) une suite:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

C'est la suite de Newton qui, sous certaines conditions, converge vers un zéro α de f

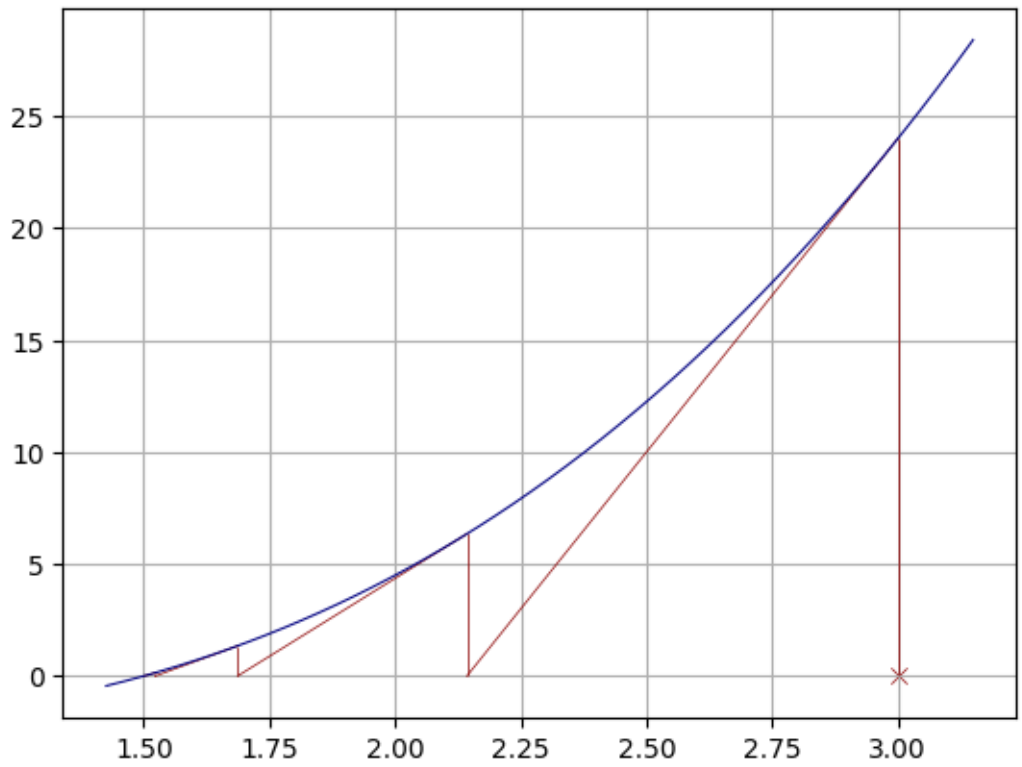
Attention aux exclus de Newton: 1. Si $f'(x_n) = 0$ 2. Si $x_n \notin D_f$ 3. Etc…

En général on s'approche d'abord du zéro avec une métode comme celle de bissection, puis on lance celle de Newton

Example implementation

[78]:
```
def newton(f, fprime, x_0, tol=-1, nmax=-1):
    root = x_0
    n = 0
```

```
    while True:
        n += 1
        if fprime(root) <= 1e-10:
            break

        newroot = root - f(root) / fprime(root)

        if abs(newroot - root) < tol and tol != -1:
            break
        if n >= nmax and nmax != -1:
            break

        ## plot
        plt.vlines(root, 0, f(root), linewidth=0.5, color='darkred')
        plt.plot((root, newroot), (f(root), 0), linewidth=0.5, color='darkred')
        ##

        root = newroot

    ## plot
    array = [x_0, root]
    plt.scatter(x_0, 0, marker='x', linewidth=0.5, color='darkred')
    plt.plot(x_arr:=np.linspace(min(array) - abs(0.05*min(array)), max(array) +␣
 ↪abs(0.05*max(array))), f(x_arr), linewidth=0.75, color='navy')
    plt.grid()
    plt.show()
    ##

    return root, n

print(newton(lambda x: (x + 1)**2 * (x - 3/2), lambda x: 3*x**2 + x - 2, 3,␣
 ↪tol=1e-8))
```




```
(1.5000000000000135, 7)
```

Fixed point method (Picard) Au lieu d'étudier $f(\alpha) = 0$, on étudie $\alpha - \phi(\alpha) = 0$, où ϕ est telle que si $f(\alpha) = 0$, alors $\phi(\alpha) = \alpha$
On cherche alors les points fixes de ϕ

Piccard's method Partant d'une bonne approximation (par bissection) d'un zéro α, on construit la suite:

$$x_{n+1} = \phi(x_n)$$

Cette suite convergera vers α, le point fixe de ϕ et le zéro de f, sous la condition que ϕ soit k-contractante autour du point fixe

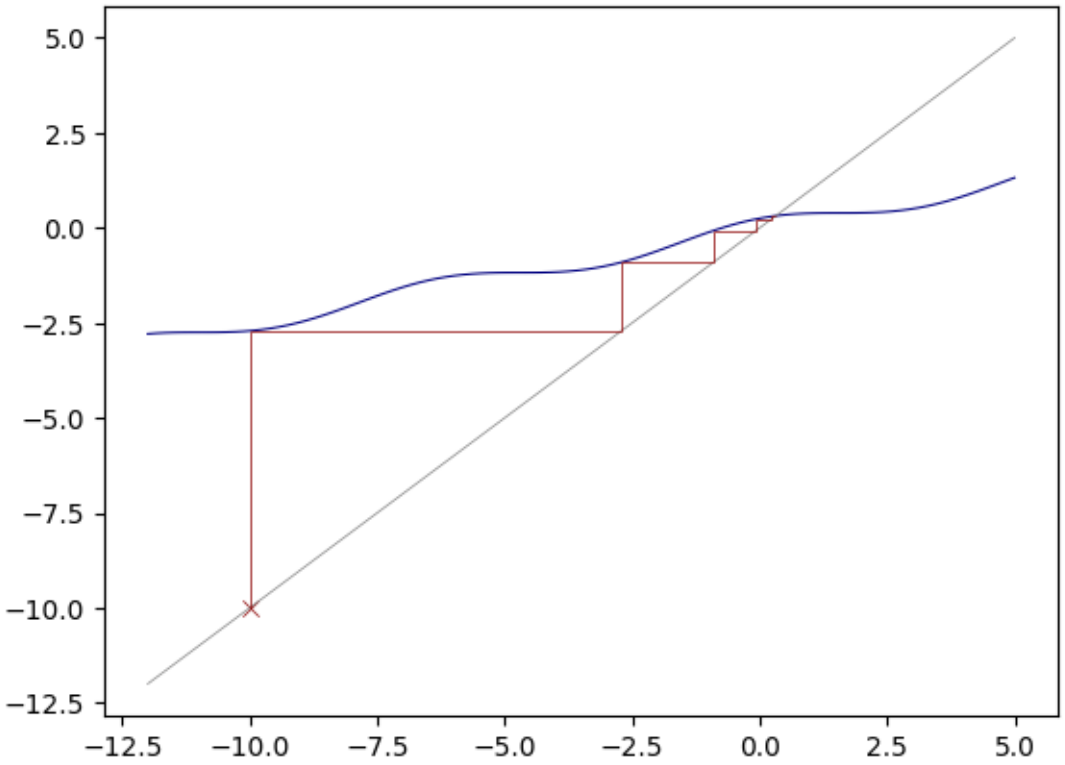
K-Contraction Pour qu'une fonction ϕ soit k-contractante sur un intervalle I, il faut que la valeur absolue de sa pente ($|\phi'(x)|$) soit inférieure à 1 sur I, ça implique entre autre que si la fonction possède un unique point fixe sur cet intervalle, ce point fixe sera unique
La méthode de Picard est garantie de converger pour tout $x_0 \in I$, si ϕ est k-contractante et possède un point fixe sur I, cela n'exclut pas convergence hors de cet intervalle par ailleurs!

Note (Théorème de Brouwer): Une fonction ϕ possède au moins un point fixe sur $[a; b]$ ssi. $\phi(x) \in [a; b]$ $\forall x \in [a; b]$

Example implementation Il est vérifié préalablement que $f(x)$ (joue ici le rôle de $\phi(x)$) est k-contractante sur tout $\mathbb{R}$ et qu'elle possède un point fixe

[80]:
```
f = lambda x: (np.cos(x) + x) / 4
x_arr = np.linspace(-12, 5, 100)

# stop tolerance
tol = 1e-5

plt.plot(x_arr, f(x_arr), color='navy', linewidth=0.75)
plt.plot(x_arr, x_arr, color='gray', linewidth=0.5)

# picard
x_0 = -10
x = x_0

plt.scatter(x, x, marker='x', color='darkred', linewidth=0.5)

while True:
    plt.vlines(x, x, f(x), color='darkred', linewidth=0.5)
    new_x = f(x)
    plt.hlines(f(x), f(x), x, color='darkred', linewidth=0.5)
    if abs(new_x - x) <= tol:
        print(f'Valeur approchée: {new_x}')
        break
    x = new_x

plt.show()
```

```
Valeur approchée: 0.3167504189712746
```



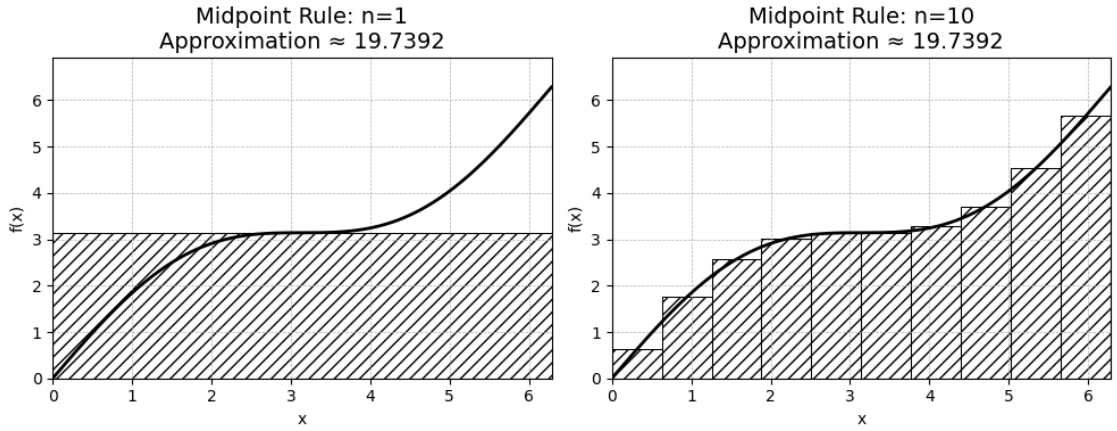
2.5 Numerical integration

Pour calculer des valeurs d'intégrales définies $\int_a^b f(x)dx$ numériquement, plusieurs méthodes existent, mais la plupart se basent sur le même principe: 1. Diviser l'intervalle $[a; b]$ en n sous-intervalles $[k; k+1]$ 2. Approximer l'aire sous la fonction sur chaque sous-intervalle à l'aide d'une autre fonction g (par exemple approximer l'aire sous f par des petits rectangles)

Middle point Voici un example de calcul, en prenant comme hauteur du rectangle la hauteur de f au point milieu de chaque sous intervalle

[14]:
```
f = lambda x: np.sin(x) + x
a, b = 0, 2 * np.pi

# Set up the figure with 2 side-by-side subplots
fig, axs = plt.subplots(1, 2, figsize=(10, 4))

# Different n values: 1 and a higher one (e.g., 4)
for ax, n in zip(axs, [1, 10]):
    # Plot the function curve in black
```

```
    x = np.linspace(a, b, 500)
    ax.plot(x, f(x), color='black', linewidth=2)

    # Midpoint rule rectangles with hatch
    h = (b - a) / n
    midpoints = a + h * (np.arange(n) + 0.5)
    heights = f(midpoints)

    # Draw hatched rectangles
    for xi, hi in zip(midpoints, heights):
        ax.bar(
            xi,
            hi,
            width=h,
            align='center',
            facecolor='none',
            edgecolor='black',
            hatch='///',
            linewidth=0.75
        )

    # Compute approximation
    approx = np.sum(heights * h)

    # Annotate plot
    ax.set_title(f"Midpoint Rule: n={n}\nApproximation   {approx:.4f}",␣
↪fontsize=14)
    ax.set_xlim(a, b)
    y_min = min(0, np.min(f(x)))
    y_max = np.max(f(x)) * 1.1
    ax.set_ylim(y_min, y_max)
    ax.set_xlabel('x')
    ax.set_ylabel('f(x)')
    ax.grid(True, linestyle='--', linewidth=0.5)

plt.tight_layout()
plt.show()
```



D'autres méthodes existent pour approximer la fonction mieux, comme utiliser des trapèzes au lieu de rectangles, ou encore intégrer algèbriquement des polynomes qui ressemblent à la fonction f, pour plus de détails, voir Série 12

Lagrange polynomial fitting Pour trouver l'expression du **seul polynôme** $L(x)$ de degré $\deg(L) \leq n-1$ passant par une liste de n points $(x_1, y_1), \dots, (x_n, y_n)$, on applique la **méthode de Lagrange** (voir Wikipedia pour plus de détails).

L'idée est de construire une base de $\mathbb{P}_{n-1}$, l'espace des polynômes de degré au plus $n-1$, constituée de *polynômes de Lagrange*:

$$\mathcal{B} = \{\ell_1(x), \ell_2(x), \dots, \ell_n(x)\}$$

Ces polynômes vérifient les propriétés suivantes :

$$\ell_k(x_i) = \begin{cases} 1 & \text{si } i = k \\ 0 & \text{si } i \neq k \end{cases} \quad \text{pour } 1 \leq i, k \leq n$$

Chaque $\ell_k(x)$ est défini comme :

$$\ell_k(x) = \prod_{\substack{1 \leq j \leq n \\ j \neq k}} \frac{x - x_j}{x_k - x_j}$$

Autrement dit, c'est un produit de fractions où l'on fait varier tous les indices sauf k.

Formule du polynôme d'interpolation de Lagrange:

Le polynôme $L(x)$ qui interpole les données $(x_1, y_1), \dots, (x_n, y_n)$ est alors donné par :

$$L(x) = \sum_{k=1}^{n} y_k \cdot \ell_k(x)$$

Exemple:

Considérons trois points :
$(1, 2), (2, 3), (4, 1)$

On calcule chaque $\ell_k(x)$, puis :

$$L(x) = 2 \cdot \ell_1(x) + 3 \cdot \ell_2(x) + 1 \cdot \ell_3(x)$$

(à développer pour obtenir un polynôme explicite)

Example implementation: Simpson method La méthode de Simpson, en plus de découper l'intervalle d'integration en n sous-intervalles, se charge d'approximer la fonction sur chaque sous-intervalle par un unique polynome de Lagrange, cette méthode converge très rapidement, elle permet même (après avoir adapté la méthode de Lagrange pour en obtenir les coefficients) d'effectuer une intégration algèbrique sur ce petit sous intevalle

Voir: Série 12 Exercice 3
