Chapter 1: Python introduction

Werner Bauer

UNIVERSITY OF SURREY

Outline

Python installation

- Python is an open source (freely available) programming language. There are many ways to install and use Python. One possible way is:
  - **Anaconda**: a Python distribution for scientific computing,
  - **Spyder**: a Python IDE, particularly suited for Matlab users.
- Using Anaconda, you can install Spyder as follows:

```
conda install spyder # execute this in a shell
```

- Install also already NumPy, SciPy, and Matplotlib:

```
conda install numpy scipy matplotlib
```

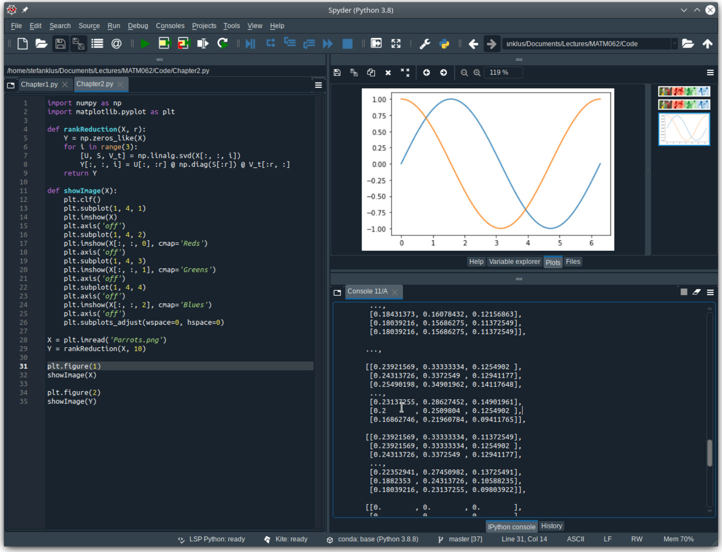
- If you want to use Jupyter notebooks:

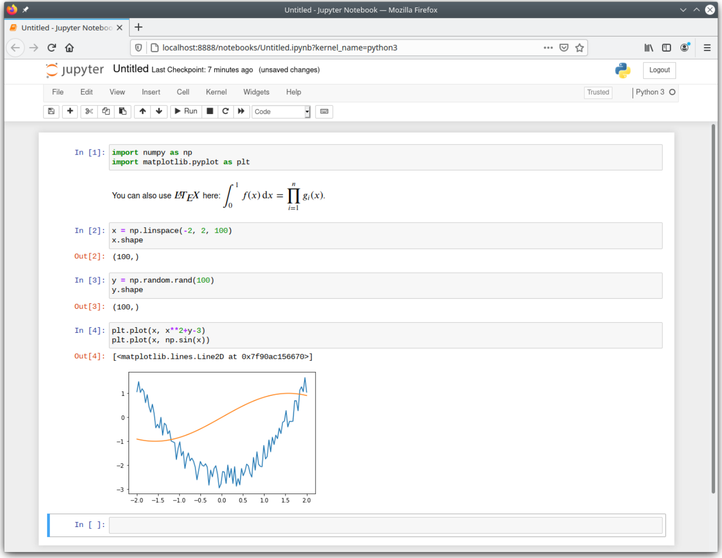
```
conda install jupyter
```

- The obligatory “Hello, World!”program:

```
print("Hello, World!")
```

Spyder




Untitled - Jupyter Notebook — Mozilla Firefox
Untitled - Jupyter Noteboo
localhost:8888/notebooks/Untitled.ipynb?kernel_name=python3
jupyter
Untitled Last Checkpoint: 7 minutes ago (unsaved changes)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
Python 3
Run
Code
In [1]:
import numpy as np
import matplotlib.pyplot as plt
You can also use LaTeX here: $\int_0^1 f(x)\,\mathrm{d}x = \prod_{i=1}^{n} g_i(x)$.
In [2]:
x = np.linspace(-2, 2, 100)
x.shape
Out[2]:
(100,)
In [3]:
y = np.random.rand(100)
y.shape
Out[3]:
(100,)
In [4]:
plt.plot(x, x**2+y-3)
plt.plot(x, np.sin(x))
Out[4]:
[<matplotlib.lines.Line2D at 0x7f90ac156670>]
1
0
−1
−2
−3
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
In [ ]:

Variables and data types

- Variable names must start with a letter or an underscore:

```python
b = True     # boolean
n = 3        # integer
x = 1.41     # floating point number
z = 1 + 2j   # complex number
s = 'text'   # string
t = None     # empty value
```

- Print the variable:

```python
print(b) # True
print(n) # 3
print(x) # 1.41
print(z) # (1+2j)
print(s) # text
print(t) # None
```

- Python automatically assigns a data type:

```python
print(type(b)) # <class 'bool'>
print(type(n)) # <class 'int'>
print(type(x)) # <class 'float'>
print(type(z)) # <class 'complex'>
print(type(s)) # <class 'str'>
print(type(t)) # <class 'NoneType'>
```

```python
b = [False, True]

for i in b:
    print("!{} = {}".format(i, not i))

# !False = True
# !True = False

for i in b:
    for j in b:
        print("{} & {} = {}".format(i, j, i and j))

# False & False = False
# False & True = False
# True & False = False
# True & True = True

for i in b:
    for j in b:
        print("{} | {} = {}".format(i, j, i or j))

# False | False = False
# False | True = True
# True | False = True
# True | True = True
```

A	$\neg A$
0	1
1	0

A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

A	B	$A \vee B$
0	0	0
0	1	1
1	0	1
1	1	1

Tuple, list, set, and dict

- Note that Python uses 0-based indexing (unlike Matlab).
- Tuples are ordered and cannot be modified:

```
t = (2, 'a', 11.2, (3, 4), True)
print(t[2]) # 11.2
```

- Lists are ordered and can be modified:

```
l = [5, 4, 3]
l.append(2)    # add 2 to the list
print(l)       # [5, 4, 3, 2]
print(len(l))  # 4

l = l[::-1]    # revert list
print(l)       # [2, 3, 4, 5]

l.insert(1, 7) # insert 7 at the first index
print(l)       # [2, 7, 3, 4, 5]

l.pop()        # remove last item
print(l)       # [2, 7, 3, 4]
```

- Access the last elements of a list using negative indices:

```
print(l[-1]) # 4
print(l[-2]) # 3
print(l[-3]) # 7
```

Tuple, list, set, and dict

- Sets are unordered collections of variables:

```
s = {'a', 'b', 'c'}

print(s == {'b', 'a', 'c', 'a'}) # True

s.add('d') # add an element to the set
print(s)   # {'c', 'b', 'a', 'd'}

print('a' in s) # True
print('z' in s) # False
```

- Dictionaries are unordered collections of key-value pairs:

```
d = {'red':  [1, 0, 0],
     'green':[0, 1, 0],
     'blue': [0, 0, 1]}

print(d['red'])   # [1, 0, 0]
print(d.keys())   # dict_keys(['red', 'green', 'blue'])
print(d.values()) # dict_values([[1, 0, 0], [0, 1, 0], [0, 0, 1]])

print(d['yellow']) # KeyError: 'yellow'
```

- Use `defaultdict` if you want to provide a default value.

Module import

- Additional modules can be imported as follows:

```python
import math
print(math.factorial(5)) # 120
dir(math) # prints a list of functions contained in math
```

- You can also write your own modules and import them.
- The imported modules need to be in the same directory or contained in the search path for module files.
- In Spyder, open the path manager or use:

```python
sys.path.insert(0, "/home/xyz/FolderName")
```

- Import only specific functions:

```python
from math import factorial
factorial(5) # note that we can now write factorial instead of math.factorial
```

- It is also possible to give imported modules a new name:

```python
import numpy as np              # these are common aliases
import scipy as sp              # which will often be used
import matplotlib.pyplot as plt  # in what follows
```

Python:

```python
for i in range(10):
    print(i)

for i in range(10, 20):
    print(i)

for i in range(20, 30, 2):
    print(i)
```

Matlab:

```matlab
for i = 0:9
    fprintf('%d\n', i);
end
for i = 10:19
    fprintf('%d\n', i);
end
for i = 20:2:28
    fprintf('%d\n', i);
end
```

Loop 1: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$
Loop 2: $10, 11, 12, 13, 14, 15, 16, 17, 18, 19$
Loop 3: $20, 22, 24, 26, 28$

Python:

```python
d = 256
while d > 1:
    d = d//2
    print(d)
```

Matlab:

```matlab
d = 256;
while d > 1
    d = d/2;
    fprintf('%d\n', d);
end
```

Output: $128, 64, 32, 16, 8, 4, 2, 1$

Use / for floating point division and // for integer division.
That is, $5/2 = 2.5$ and $5//2 = 2$.

Break and continue

Python:

```python
s = 0
for i in range(10):
    s = s + i**2
    if s > 50:
        break
print(s) # 55
```

Matlab:

```matlab
s = 0;
for i = 0:9
    s = s + i^2;
    if s > 50
        break
    end
end
disp(s); % 55
```

The break statement terminates the current loop.

Python:

```python
for i in range(10):
    if i % 2 == 0:
        continue
    print(i) # 1, 3, 5, 7, 9
```

Matlab:

```matlab
for i = 0:9
    if mod(i, 2) == 0
        continue;
    end
    disp(i); % 1, 3, 5, 7, 9
end
```

The continue statement skips all the remaining statements and returns the control to the beginning of the loop.

Python:

```python
def fibonacci():
    x, y = 0, 1
    while True:
        x, y = y, x+y
        yield y

g = fibonacci()

print(next(g)) # 1
# ...
print(next(g)) # 2
# ...
l = [next(g) for i in range(10)] # [3, 5, 8, 13, 21, 34, 55, 89, 144, 233]
```

- If a function contains at least one yield statement, it becomes a generator function.
- Yield pauses the function, saves its current state, and continues when the function is called again.
- There is no comparable Matlab feature.

If/else statements

Python:

```python
for i in range(1, 5):
    if i % 2 == 0:
        print('even')
    else:
        print('odd')
```

Matlab:

```matlab
for i = 1:4
    if mod(i, 2) == 0
        fprintf('even\n')
    else
        fprintf('odd\n')
    end
end
```

Output: odd, even, odd, even

Python:

```python
if expression:
    statements
elif expression:
    statements
else:
    statements
```

Matlab:

```matlab
if expression
    statements
elseif expression
    statements
else
    statements
end
```

Indentation

Note that Python relies on the indentation to define a block of code. Other programming languages often use `begin` and `end` or { and } for this purpose.

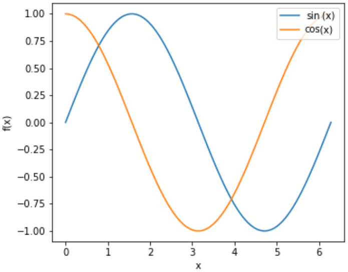
Lambda functions and anonymous functions

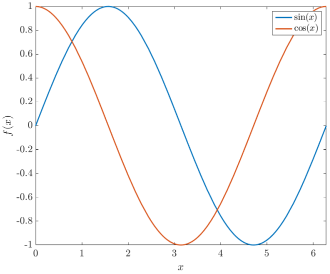
Python:

```python
f = lambda x: math.sin(x)
g = lambda x: math.cos(x)
h = lambda x, y: x**2 + y**2
f(math.pi)
h(2, 3)
x = np.linspace(0, 2*math.pi, 100)
plt.plot(x, np.vectorize(f)(x))
plt.plot(x, np.vectorize(g)(x))
```

Matlab:

```matlab
f = @(x) sin(x);
g = @(x) cos(x);
h = @(x, y) x^2 + y^2;
f(pi)
h(2, 3)
figure; hold on;
fplot(f, [0, 2*pi]);
fplot(g, [0, 2*pi]);
```





- Apply `len` to all items of the list:

```
tuple_lengths = list(map(len, [(1,), (1, 2), (1, 2, 3)])) # tuple_lengths = [1, 2, 3]
```

- Apply lambda function sequentially to pairs of values:

```
from functools import reduce
r = reduce((lambda x, y: x * y), [1, 2, 3, 4]) # r = 24
```

- This corresponds to:

```
r = 1
for i in [1, 2, 3, 4]:
    r = r * i
```

- Filter values for which the lambda function returns `True`:

```
x = [7, -1, 3, 2, -3, -2]
y = [i for i in filter(lambda x: x>0, x)] # y = [7, 3, 2]
```

- You could also write:

```
y = [i for i in x if i > 0]
```

- Generate a list of all permutations of $[1, 2, 3]$:

```python
x = [1, 2, 3]
y = list(itertools.permutations(x)) # the generator is converted to a list

print(y) # [(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]
```

- Generate a list of all combinations of $[1, 2, 3, 4]$:

```python
import itertools

x = [1, 2, 3, 4]
y = list(itertools.combinations(x, 3))

print(y) # [(1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4)]
```

- Compute accumulated sums of $[1, 2, 3, 4, 5]$:

```python
import itertools

x = [1, 2, 3, 4, 5]
y = list(itertools.accumulate(x))

print(y) # [1, 3, 6, 10, 15]
```

Function arguments

- Passing single argument:

```python
def double(n):
    return 2*n
double(2) # 4
```

- Passing multiple arguments to a function:

```python
def func(*args): # args is a tuple that contains all parameters
    for i in args:
        print(i)

func(1, 'a', -2)
# 1
# a
# -2
l = [1,'a',-2]
func(*l) # * unpacks the list, same output as above

func(l) # [1, 'a', -2]
```

- Passing multiple named arguments to a function:

```python
def func(**kwargs): # kwargs is a dictionary that contains keyword/value pairs
    for keyword, value in kwargs.items():
        print(f'{keyword} = {value}')

func(kw1=1, kw2='a', kw3=-2)
# kw1 = 1
# kw2 = a
# kw3 = -2
```

- Can of course be combined.

Classes

```python
class Fraction(object):
    def __init__(self, a, b):
        '''Initializer or constructor, called when you create a new object.'''
        d = math.gcd(a, b) # greatest common divisor of a and b
        self.a = a // d
        self.b = b // d

    def __repr__(self):
        '''Prints fraction.'''
        return "Fraction {}/{}".format(self.a, self.b)

    def __add__(self, rhs):
        '''Adds two fractions.'''
        p = self.a * rhs.b + self.b * rhs.a
        q = self.b * rhs.b
        return Fraction(p, q)

    def __eq__(self, rhs):
        '''Checks if two fractions are equal.'''
        return self.a * rhs.b == rhs.a * self.b

x1 = Fraction(1, 3) # x1 = 1/3
x2 = Fraction(1, 6) # x2 = 1/6
x3 = x1 + x2

print(x3)                   # Fraction 1/2
print(x3 == Fraction(2, 4)) # True

x4 = x1 - x2 # not implemented yet, define __sub__ function
```

```python
class Person(object):
    type = 'Person' # this is a class attribute, everything is by default public

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f'Name: {self.name}\nAge: {self.age}'

    def increaseAge(self):
        self.age += 1

class Student(Person): # inherits properties and functions from Person
    type = 'Student'

    def __init__(self, name, age, urn):
        super().__init__(name, age)
        self.urn = urn

    def __str__(self):
        return super().__str__() + f'\nURN: {self.urn}'

c1 = Person('Alice', 21)
c2 = Student('Bob', 22, 720283)
print(c1)        # Name: Alice, Age: 21
print(c1.type)   # Person
print(c2)        # Name: Bob, Age: 22, URN: 720283
print(c2.type)   # Student
c2.increaseAge() # calls the inherited function
print(c2.age)    # 23
```

Other useful features

- Different string formatting options:

```
x = 5
print("%d^2 = %d" % (x, x**2))      # 5^2 = 25 (old C-style formatting)
print("{}^2 = {}".format(x, x**2)) # 5^2 = 25 (newer)
print(f"{x}^2 = {x**2}")            # 5^2 = 25 (even newer)
```

- Don't add a newline to the string:

```
print("Don't end the line ", end="")
print("here!")
```

- Conversion of data types:

```
a = 123    # integer
b = '456'  # string

a = str(a) # '123'
b = int(b) # 456
```

- In-place operations:

```
a, b, c = 1, 2, 3
a += 2  # 3
b *= 3  # 6
c **= 4 # 81
```

- Conditional expressions:

```python
def maxVal(a, b):
    return a if a > b else b

print(maxVal(2, 5)) # 5
print(maxVal(7, 5)) # 7
```

- This is similar to the ternary operator in C++:

```cpp
#include <iostream>

template<typename T>
T maxVal(T a, T b)
{
    return a > b ? a : b;
}

int main()
{
    std::cout << maxVal(2, 5) << "\n" << maxVal(7, 5) << std::endl;
}
```

- List comprehension:

```python
l1 = [1, 3, 2, 5, 4, 6]
l2 = [i**2 for i in l1]                # [1, 9, 4, 25, 16, 36]
l3 = [2*i for i in l2 if i % 2 == 0] # [8, 32, 72]
```

Other useful features

- Use the keyword `pass` as a placeholder for future code:

```
def toBeImplemented():
    pass
```

- Enumerate items:

```
for index, item in enumerate(['th', 'st', 'nd', 'rd', 'th']):
    print('{}{}'.format(index, item)) # 0th, 1st, 2nd, 3rd, 4th
```

- Create dictionaries:

```
d = dict(x=11, y=22, z=33)
print(d['y']) # 22
```

- Document your functions and classes with docstrings:

```
def sqr(x):
    '''This function returns x squared.'''
    return x**2 # note that in Matlab we would write x^2

print(sqr.__doc__) # This function returns x squared.
```

- Check whether objects are identical:

```python
a = [1, 2, 3]
b = a.copy()
print(a == b) # True  (compares the value of a and b)
print(a is b) # False (compares the identities of a and b)

s = 'text'
t = s
print(s == t) # True (compares the value of s and t)
print(s is t) # True (compares the identities of s and t)
```

- Note that Python uses reference counting:

```python
from sys import getrefcount

s = 'some random text'
print(getrefcount(s)) # 3
t = s
print(getrefcount(s)) # 4
t = 'new text'
print(getrefcount(s)) # 3
```

- You can also compare the addresses:

```python
s = 'some random text'
t = s
print(f"Address of s: {id(s)}") # 140512670168288
print(f"Address of t: {id(t)}") # 140512670168288
```

Outline

Scalars, vectors, and matrices

Python:

```
n = 3
a = 1.5
x = np.array([1, 0, 2])
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
B = np.zeros((n, n))
C = np.ones((n, n))
D = np.eye(n)
y = A @ x
M = A @ C
N = A.T
```

Matlab:

```
n = 3
a = 1.5
x = [1; 0; 2]
A = [1, 2, 3; 4, 5, 6; 7, 8, 9]
B = zeros(n)
C = ones(n)
D = eye(n)
y = A*x
M = A*C
N = A'
```

$$n = 3,\ a = 1.5,\ x = \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, C = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$y = \begin{bmatrix} 7 \\ 16 \\ 25 \end{bmatrix}, M = \begin{bmatrix} 6 & 6 & 6 \\ 15 & 15 & 15 \\ 24 & 24 & 24 \end{bmatrix}, N = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

Accessing vectors and matrices

Python:

```python
n = 4
A = sp.linalg.hilbert(n) # Hilbert matrix of size 4
c = A[:, 1]              # second column
r = A[0, :]              # first row
s = A[::2, :]            # rows one and three, all columns
```

Matlab:

```matlab
n = 4
A = hilb(n)
c = A(:, 2)
r = A(1, :)
s = A(1:2:end, :)
```

$n = 4$

$$A = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \end{bmatrix}, \; c = \begin{bmatrix} \frac{1}{2} \\ \frac{1}{3} \\ \frac{1}{4} \\ \frac{1}{5} \end{bmatrix}$$

$$r = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \end{bmatrix}$$

$$s = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \end{bmatrix}$$

Important differences

Python:

```python
def f(x):
    for i in range(len(x)):
        x[i] = i+1

x = np.zeros(4) # [0, 0, 0, 0]
f(x)
print(x) # [1, 2, 3, 4]
```

Matlab:

```matlab
x = zeros(4, 1);
f(x)
disp(x) % [0, 0, 0, 0]

function f(x)
    for i = 1:length(x)
        x(i) = i;
    end
end
```

Python:

```python
v = np.array(range(6)) # [0 1 2 3 4 5]
w = v[3:]              # [3 4 5]
w[0] = 99              # [99  4  5]
print(v)               # [0  1  2 99  4  5]
```

Matlab:

```matlab
v = [0:5]     % [0 1 2 3 4 5]
w = v(4:end) % [3 4 5]
w(1) = 99     % [99  4  5]
disp(v)       % [0 1 2 3 4 5]
```

Matlab creates copies of vectors and matrices, Python does not. Use the `.copy()` function to create copies of NumPy arrays.

Saving/loading MAT files

- Save variables into a Matlab-style MAT file:

```python
import numpy as np
import scipy as sp
import scipy.io

n = 100
x = np.linspace(-1, 1, n)
y = x**2
A = np.eye(n)

sp.io.savemat('Results.mat', {'n':n, 'x':x, 'y':y, 'A':A})
```

- Load file in Matlab:

```matlab
load Results
plot(x, y)
```

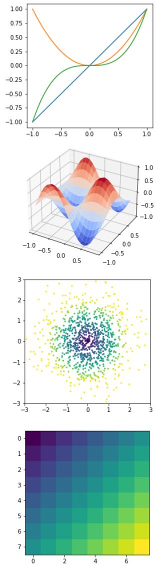
- Load variables from MAT file into main scope:

```python
data = scipy.io.loadmat('Results.mat', squeeze_me=True)
for s in data.keys():
    if s[:2] == '__' and s[-2:] == '__': continue
    exec('%s = data["%s"]' % (s, s))
```

Outline

Plotting

```
import numpy as np
import scipy as sp
import scipy.linalg
import matplotlib.pyplot as plt
from matplotlib import cm

fig = plt.figure(1); plt.clf()

plt.subplot(1, 4, 1)
x = np.linspace(-1, 1, 100)
plt.plot(x, x)
plt.plot(x, x**2)
plt.plot(x, x**3)

ax = fig.add_subplot(1, 4, 2, projection='3d')
x = np.arange(-1, 1, 0.1)
y = np.arange(-1, 1, 0.1)
X, Y = np.meshgrid(x, y)
Z = np.sin(np.pi*X) * np.cos(np.pi*Y)
ax.plot_surface(X, Y, Z, cmap=cm.coolwarm)

plt.subplot(1, 4, 3)
x = np.random.randn(2, 1000)
plt.scatter(x[0, :], x[1, :], s=4, \
            c=np.sqrt(x[0, :]**2 + x[1, :]**2), vmin=0, vmax=2)
plt.xlim(-3, 3); plt.ylim(-3, 3);

plt.subplot(1, 4, 4)
plt.imshow(1/sp.linalg.hilbert(8))
```



Outline

Compute $y = \alpha x$.

Python:

```
import numpy as np
n = 5
alpha = 2
x = np.linspace(1,5,5) # [1., 2., 3., 4., 5.]

y = np.zeros(n)
for i in range(n):
    y[i] = alpha*x[i]
```

Matlab:

```
n = 5;
alpha = 2;
x = [1:n]';

y = zeros(n, 1);
for i = 1:n
    y(i) = alpha*x(i);
end
```

Compute $y = Ax$.

Python:

```
import numpy as np
A = np.random.rand(n, n)
x = np.random.rand(n)

y = np.zeros(n)
for i in range(n):
    for j in range(n):
        y[i] = y[i] + A[i, j]*x[j]
```

Matlab:

```
n = 5;
A = rand(n);
x = rand(n, 1);

y = zeros(n, 1);
for i = 1:n
    for j = 1:n
        y(i) = y(i) + A(i, j)*x(j);
    end
end
```

Compute $C = AB$.

Python:

```python
import numpy as np
n = 5
A = np.random.rand(n, n)
B = np.random.rand(n, n)

C = np.zeros((n, n))
for i in range(n):
    for j in range(n):
        for k in range(n):
            C[i, j] = C[i, j] + A[i, k]*B[k, j]
```

Matlab:

```matlab
n = 5;
A = rand(n);
B = rand(n);

C = zeros(n);
for i = 1:n
    for j = 1:n
        for k = 1:n
            C(i, j) = C(i, j) + A(i, k)*B(k, j);
        end
    end
end
```

Outline

- Importing Pandas (docu: https://pandas.pydata.org/)

```
import pandas as pd
# Creating series object 'data':
data = pd.Series([0.25, 0.5, 0.75, 1.0], index=['a', 'b', 'c', 'd'])
data
# Output:
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64 # NOTE: in the following we skip this line of output
```

- Accessing data (using dictionary-like expressions)

```
data['b'] # 0.5
'a' in data # True
data.keys() # Index(['a', 'b', 'c', 'd'], dtype='object')
list(data.items()) # [('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
```

- Modify series objects (similar to dict-syntax):

```
data['e'] = 1.25 # extend Series by assigning to a new index a value:
data # Output:
a    0.25
b    0.50
c    0.75
d    1.00
e    1.25
```

Pandas: Slicing, indexing, masking of Series objects

- slicing by explicit and implicit index

```
data['a':'c'] # explicit indexing
# Output:
a    0.25
b    0.50
data[0:2] # implicit indexing
# Output:
a    0.25
b    0.50
```

- Masking

```
 data[(data > 0.3) & (data < 0.8)]
# Output:
b    0.50
c    0.75
```

- Fancy indexing (pass a list of indices):

```
data[['a', 'e']]
# Output:
a    0.25
e    1.25
```

Danger of confusion: if a series has explicit integer index an indexing operation such as `data[1]` will use the explicit indices, while a slicing operation like `data[1:3]` will use the implicit Python-style index

Indexers: loc and iloc

Alternative indexing:

- ```
  data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
  ```

- The loc attribute allows indexing and slicing that always references the explicit index:

```
data.loc[1]
# Output:
'a'

data.loc[1:3] # this includes key-value pair with external index 3
# Output:
1    a
3    b
```

- The iloc attribute allows indexing and slicing that always references the implicit Python-style index:

```
data.iloc[1]
# Output:
'b'

data.iloc[1:2] # using implicit indexing, last value for last index not included
# Output:
3    b

data.iloc[1:3]
# Output:
3    b
5    c
```
```

- DataFrame as a dictionary of related Series objects:

```
area = pd.Series({'California': 423967, 'Texas': 695662, 'New York': 141297,
                  'Florida': 170312, 'Illinois': 149995})
pop = pd.Series({'California': 38332521, 'Texas': 26448193, 'New York': 19651127,
                 'Florida': 19552860, 'Illinois': 12882135})
data = pd.DataFrame({'area':area, 'pop':pop})

print(data) # Output:
              area       pop
California  423967  38332521
Texas       695662  26448193
New York    141297  19651127
Florida     170312  19552860
Illinois    149995  12882135

print(data['area']) # Output:
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
```

- Dictonary-style syntax similar to Series objects, e.g.:

```
data['density'] = data['pop'] / data['area']

print(data) # Output:
            area       pop     density
California  423967  38332521   90.413926
Texas       695662  26448193   38.018740
New York    141297  19651127  139.076746
Florida     170312  19552860  114.806121
Illinois    149995  12882135   85.883763
```

- We can view DataFrame as enhanced two-dimensional array:

```
data.T # calculate transpose
# Output:
         California         Texas      New York       Florida      Illinois
area     4.239670e+05  6.956620e+05  1.412970e+05  1.703120e+05  1.499950e+05
pop      3.833252e+07  2.644819e+07  1.965113e+07  1.955286e+07  1.288214e+07
density  9.041393e+01  3.801874e+01  1.390767e+02  1.148061e+02  8.588376e+01

data.values
# Output:
array([[4.23967000e+05, 3.83325210e+07, 9.04139261e+01],
       [6.95662000e+05, 2.64481930e+07, 3.80187404e+01],
       [1.41297000e+05, 1.96511270e+07, 1.39076746e+02],
       [1.70312000e+05, 1.95528600e+07, 1.14806121e+02],
       [1.49995000e+05, 1.28821350e+07, 8.58837628e+01]])
```

- DataFrame indexing follows style of dictionary:

```
data.values[0]
# output
array([4.23967000e+05, 3.83325210e+07, 9.04139261e+01])

data['area']
# Output:
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995

data['area']['California'] # Output:
423967
```

DataFrame (DF): Access and modify data in numpy like fashion:

- `iloc` (implicit indexing): index and column labels maintained

```
data.iloc[:3, :2] # Output:
            area       pop
California  423967  38332521
Texas       695662  26448193
New York    141297  19651127
```

- `loc` (explicit indexing): index and column labels maintained

```
data.loc[:'Illinois', :'pop'] # Output:
              area       pop
California  423967  38332521
Texas       695662  26448193
New York    141297  19651127
Florida     170312  19552860
Illinois    149995  12882135
```

- Numpy masking, filtering, indexing works:

```
data.loc[data.density > 100, ['pop', 'density']] # Output:
               pop     density
New York  19651127  139.076746
Florida   19552860  114.806121

data[0:1] # Output:
              area       pop     density
California  423967  38332521   90.413926

data.iloc[0, 2] = 90 # modify value similar to numpy
data[0:1] # Output:
              area       pop  density
California  423967  38332521     90.0
```

- Concatenating:

```
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'], 'B': ['B0', 'B1', 'B2', 'B3'],
                    'C': ['C0', 'C1', 'C2', 'C3'], 'D': ['D0', 'D1', 'D2', 'D3']},
                         index=[0, 1, 2, 3])

df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],  'B': ['B4', 'B5', 'B6', 'B7'],
                    'C': ['C4', 'C5', 'C6', 'C7'],  'D': ['D4', 'D5', 'D6', 'D7']},
                         index=[4, 5, 6, 7])

df3 = pd.DataFrame({'A': ['A8', 'A9', 'A10', 'A11'], 'B': ['B8', 'B9', 'B10', 'B11'],
                    'C': ['C8', 'C9', 'C10', 'C11'], 'D': ['D8', 'D9', 'D10', 'D11']},
                         index=[8, 9, 10, 11])


pd.concat([df1,df2,df3])

# Output:
      A    B    C    D
0     A0   B0   C0   D0
1     A1   B1   C1   D1
2     A2   B2   C2   D2
3     A3   B3   C3   D3
4     A4   B4   C4   D4
5     A5   B5   C5   D5
6     A6   B6   C6   D6
7     A7   B7   C7   D7
8     A8   B8   C8   D8
9     A9   B9   C9   D9
10   A10  B10  C10  D10
11   A11  B11  C11  D11
```

- Concatenating:

```
pd.concat([df1,df2,df3],axis=1)

# Output:

      A    B    C    D    A    B    C    D    A    B    C    D
0    A0   B0   C0   D0  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
1    A1   B1   C1   D1  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
2    A2   B2   C2   D2  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
3    A3   B3   C3   D3  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
4   NaN  NaN  NaN  NaN   A4   B4   C4   D4  NaN  NaN  NaN  NaN
5   NaN  NaN  NaN  NaN   A5   B5   C5   D5  NaN  NaN  NaN  NaN
6   NaN  NaN  NaN  NaN   A6   B6   C6   D6  NaN  NaN  NaN  NaN
7   NaN  NaN  NaN  NaN   A7   B7   C7   D7  NaN  NaN  NaN  NaN
8   NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN   A8   B8   C8   D8
9   NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN   A9   B9   C9   D9
10  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  A10  B10  C10  D10
11  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  A11  B11  C11  D11
```

- Merging DF: merge via a common Series (same column name by default)

```
left = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                     'A': ['A0', 'A1', 'A2', 'A3'],
                     'B': ['B0', 'B1', 'B2', 'B3']})

right = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                      'C': ['C0', 'C1', 'C2', 'C3'],
                      'D': ['D0', 'D1', 'D2', 'D3']})



pd.merge(left,right,how='inner',on='key')

# Output:
  key   A   B   C   D
0  K0  A0  B0  C0  D0
1  K1  A1  B1  C1  D1
2  K2  A2  B2  C2  D2
3  K3  A3  B3  C3  D3
```

- Merging dataframes: inner, outer, left, right

```
left = pd.DataFrame({'key1': ['K0', 'K0', 'K1', 'K2'],
                     'key2': ['K0', 'K1', 'K0', 'K1'],
                        'A': ['A0', 'A1', 'A2', 'A3'],
                        'B': ['B0', 'B1', 'B2', 'B3']})
right = pd.DataFrame({'key1': ['K0', 'K1', 'K1', 'K2'],
                      'key2': ['K0', 'K0', 'K0', 'K0'],
                         'C': ['C0', 'C1', 'C2', 'C3'],
                         'D': ['D0', 'D1', 'D2', 'D3']})

print(pd.merge(left, right, how='inner', on=['key1', 'key2'])) #Output:
  key1 key2   A   B   C   D
0   K0   K0  A0  B0  C0  D0
1   K1   K0  A2  B2  C1  D1
2   K1   K0  A2  B2  C2  D2

print(pd.merge(left, right, how='outer', on=['key1', 'key2'])) #Output:
  key1 key2    A    B    C    D
0   K0   K0   A0   B0   C0   D0
1   K0   K1   A1   B1  NaN  NaN
2   K1   K0   A2   B2   C1   D1
3   K1   K0   A2   B2   C2   D2
4   K2   K1   A3   B3  NaN  NaN
5   K2   K0  NaN  NaN   C3   D3

print(pd.merge(left, right, how='left',on=['key1', 'key2'])) #Output:
  key1 key2   A   B    C    D
0   K0   K0  A0  B0   C0   D0
1   K0   K1  A1  B1  NaN  NaN
2   K1   K0  A2  B2   C1   D1
3   K1   K0  A2  B2   C2   D2
4   K2   K1  A3  B3  NaN  NaN
# ..., how = 'right', .. works similarly
```

Groupby

To group rows together and call aggregate functions:

```
data = {'Company':['GOOG','GOOG','MSFT','MSFT','FB','FB'],
       'Person':['Sam','Charlie','Amy','Vanessa','Carl','Sarah'],
       'Sales':[200,120,340,124,243,350]}
df = pd.DataFrame(data)
```

Based on column name, .groupby() groups rows together, e.g. by 'Company'. This will create a DataFrameGroupBy object.

```
print(df.groupby('Company').describe()) # Output:
         Sales
         count   mean         std    min     25%    50%     75%    max
Company
FB         2.0  296.5   75.660426  243.0  269.75  296.5  323.25  350.0
GOOG       2.0  160.0   56.568542  120.0  140.00  160.0  180.00  200.0
MSFT       2.0  232.0  152.735065  124.0  178.00  232.0  286.00  340.0

print(df.groupby('Company').count()) # Output:
         Person  Sales
Company
FB            2      2
GOOG          2      2
MSFT          2      2

print(df.groupby('Company').mean()) # Output
          Sales
Company
FB        296.5
GOOG      160.0
MSFT      232.0

print(df.groupby('Company').std()) # Output similar to .mean()
```

Detecting null values with isnull() and notnull()

- isnull() and notnull() return Boolean mask over data

```
import numpy as np
data = pd.Series([1, np.nan, 'hello', None])
data
# Output:
0        1
1      NaN
2    hello
3     None
data.isnull()
# Output:
0    False
1     True
2    False
3     True
data[data.notnull()] # similarly: data.dropna()
# Output:
0        1
2    hello
```

- .dropna() to drop null values

```
data = pd.Series([1, np.nan, 'hello', None])

data.dropna()
# Output:
0        1
2    hello

df = pd.DataFrame({'A':[1,2,np.nan],
                   'B':[5,np.nan,np.nan],
                   'C':[1,2,3]})
df
# Output:
        A       B       C
0       1.0     5.0     1
1       2.0     NaN     2
2       NaN     NaN     3

df.dropna(axis=0)
# Output:
     A    B  C
0  1.0  5.0  1

df.dropna(axis=1)
# Output:
        C
0       1
1       2
2       3
```

Filling null values with .fillna()

- Sometimes we want to fill rather then drop null values

```
df = pd.DataFrame({'A':[1,2,np.nan], 'B':[5,np.nan,np.nan], 'C':[1,2,3]})
print(df) # Output:
     A    B  C
0  1.0  5.0  1
1  2.0  NaN  2
2  NaN  NaN  3

df.fillna(0) # fill NA with single values such as 0
     A    B  C
0  1.0  5.0  1
1  2.0  0.0  2
2  0.0  0.0  3
df.fillna(method='ffill') # forward-fill (within columns); also backward-fill 'bfill' possible
     A    B  C
0  1.0  5.0  1
1  2.0  5.0  2
2  2.0  5.0  3

df['A'].fillna(value=df['A'].mean()) # fill missing values per columns with e.g. mean:
0    1.0
1    2.0
2    1.5

print(df) # NOTE: df is not changed !!!
     A    B  C
0  1.0  5.0  1
1  2.0  NaN  2
2  NaN  NaN  3
```

- NOTE: use `inplace=True` to overwrite values in df itself (default, as above: `inplace=False`)