Neuro1 Praktikum

Chao Huang
chao.huang@uni-leipzig.de
2021.01.11

Outline

- Programming with Python
- Digital signals from the brain
- Neural simulations
- Analog signals from the brain
- Psychophysics experiments

Introduction to Python

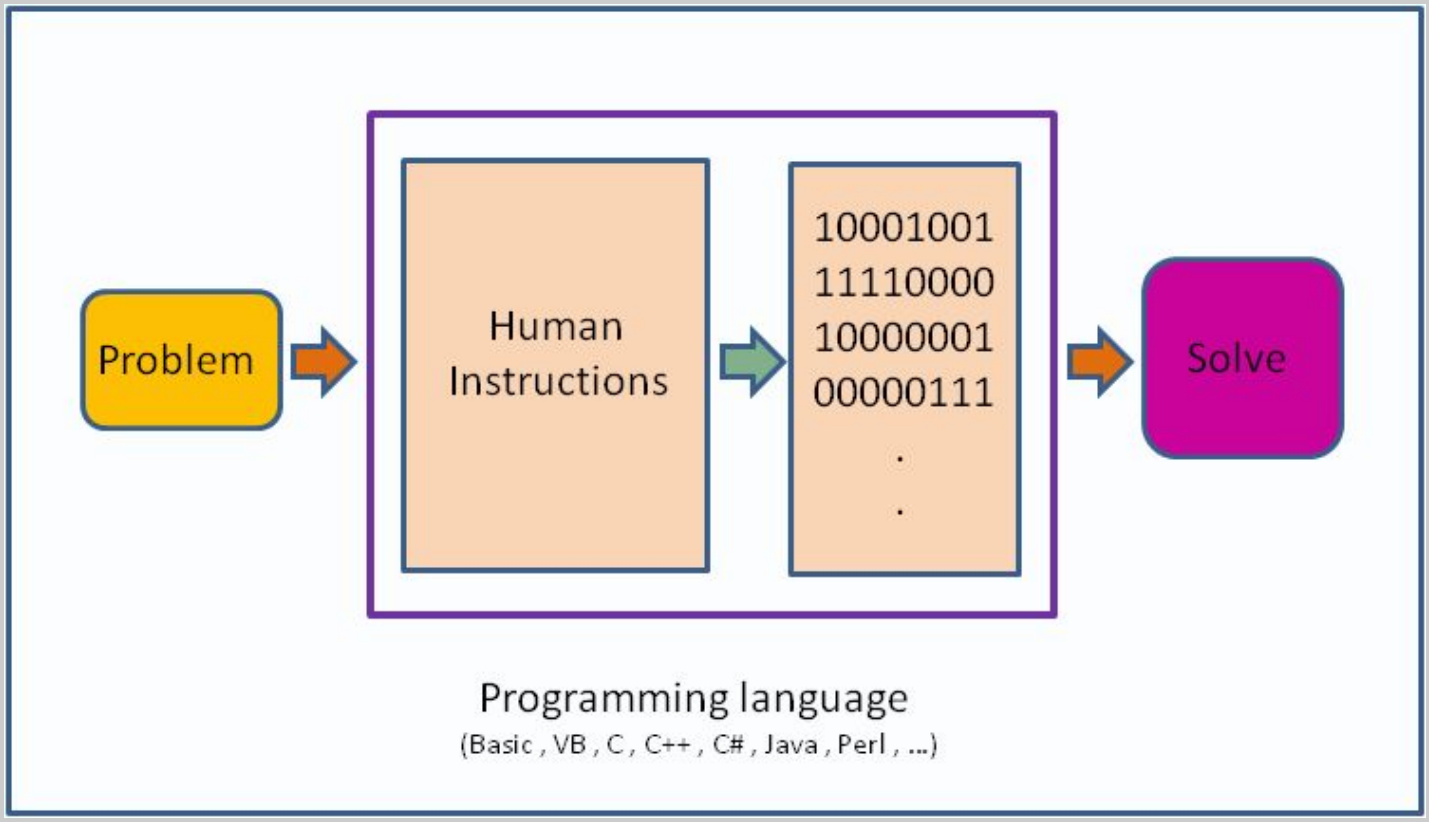
Chao Huang

2021.01.11

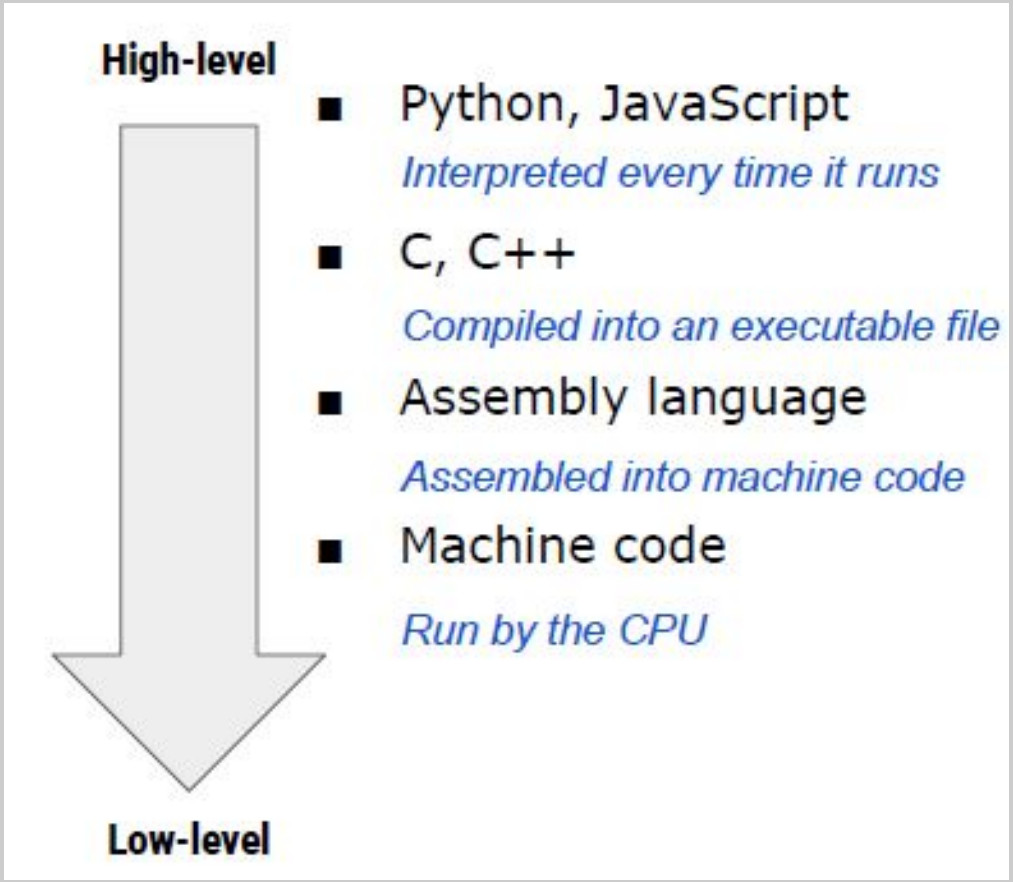
Why programming?

- Almost necessary to work in science
  - larger and larger data sets
  - hardware control, communication
  - design your own experiments
- Good for job hunting
- “It teaches you how to think”

What is programming?



What is programming?

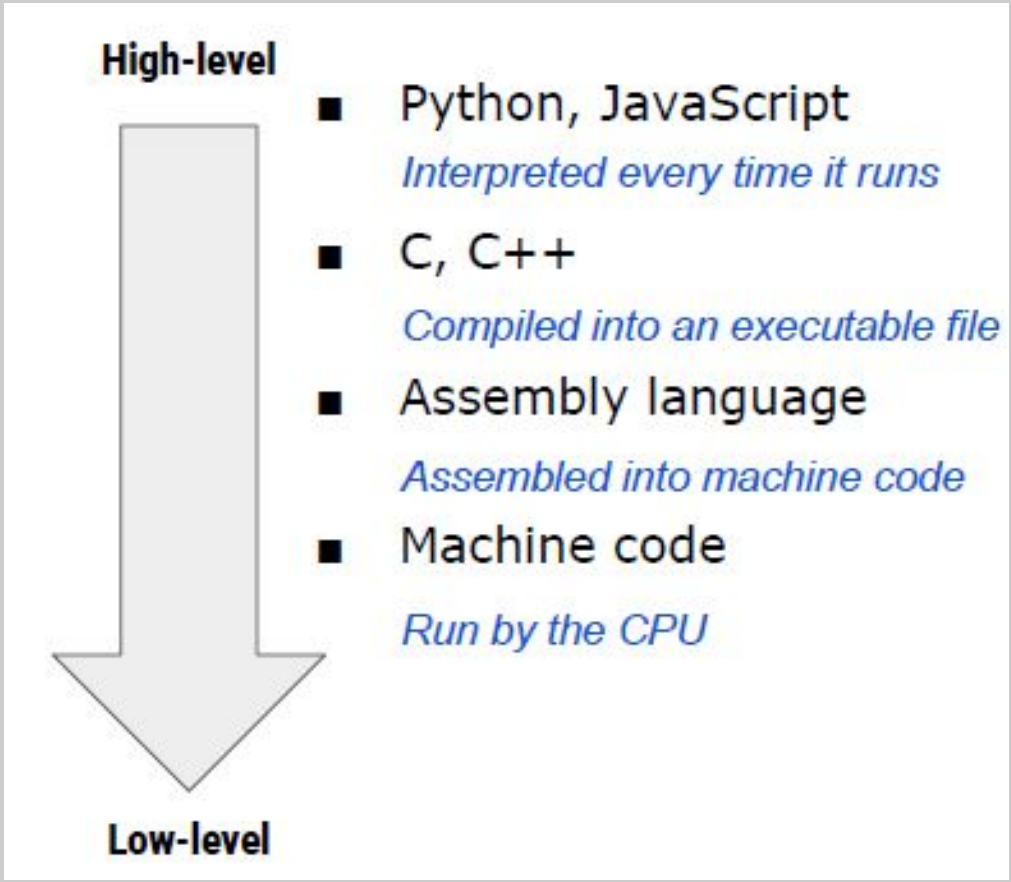


Why Python?

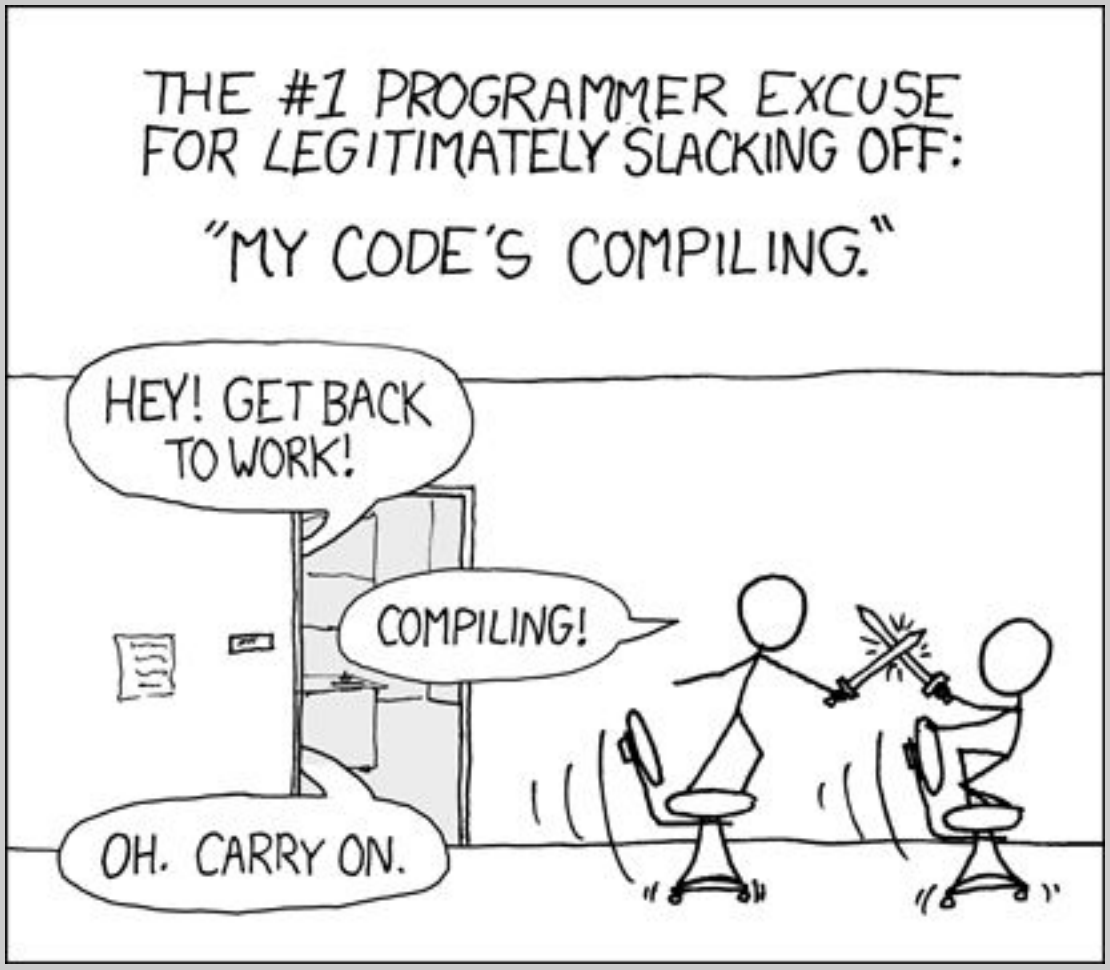
- Simple, easy to read & learn
- Interactive
- Free & Open source
- Portable

- High-level language; Object-oriented
- Very popular (lots of useful packages available)

Why Python?


High-level
Low-level
Python, JavaScript
Interpreted every time it runs
C, C++
Compiled into an executable file
Assembly language
Assembled into machine code
Machine code
Run by the CPU

Why Python?


THE #1 PROGRAMMER EXCUSE FOR LEGITIMATELY SLACKING OFF:
"MY CODE'S COMPILING."
HEY! GET BACK TO WORK!
COMPILING!
OH. CARRY ON.

What you need to learn Python?

- Python
- An editor to write Python code

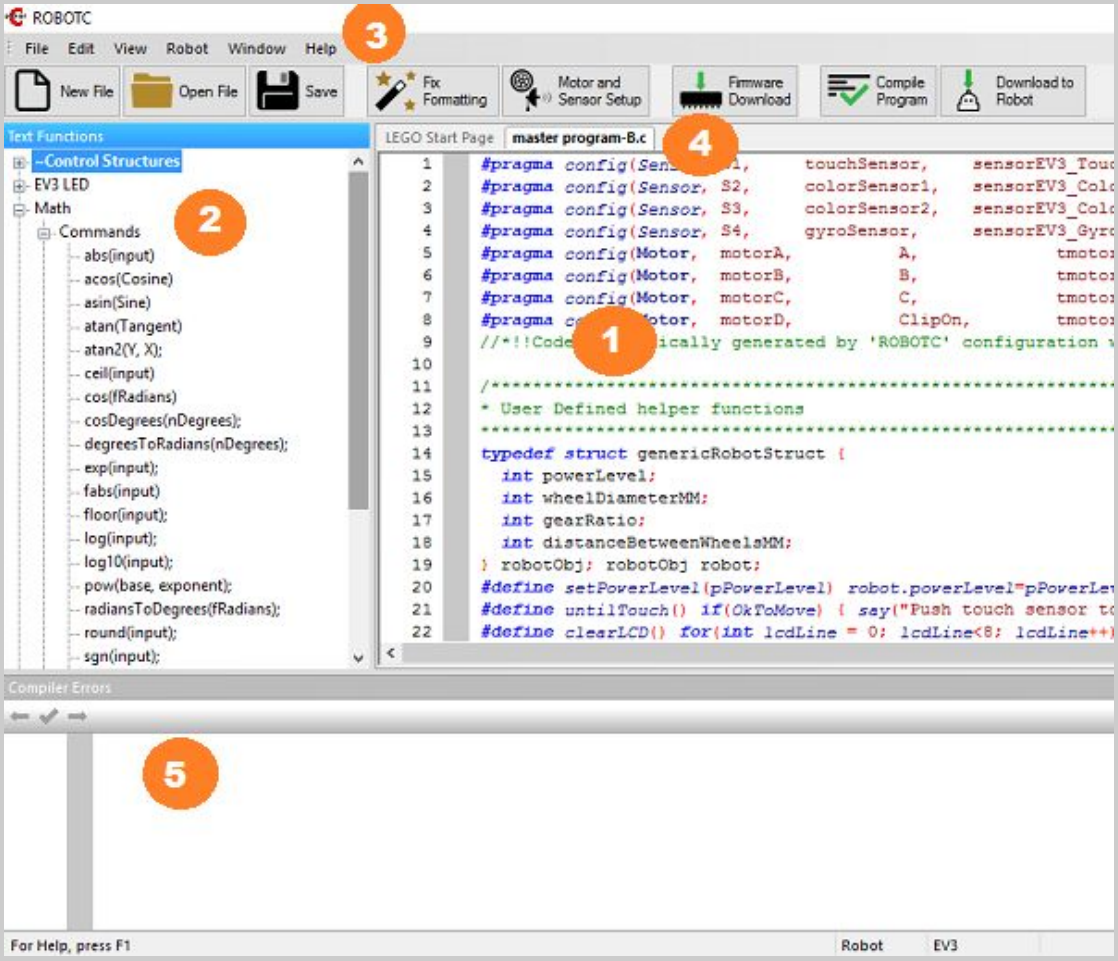
- Tools for testing, debug, ...

Install Python

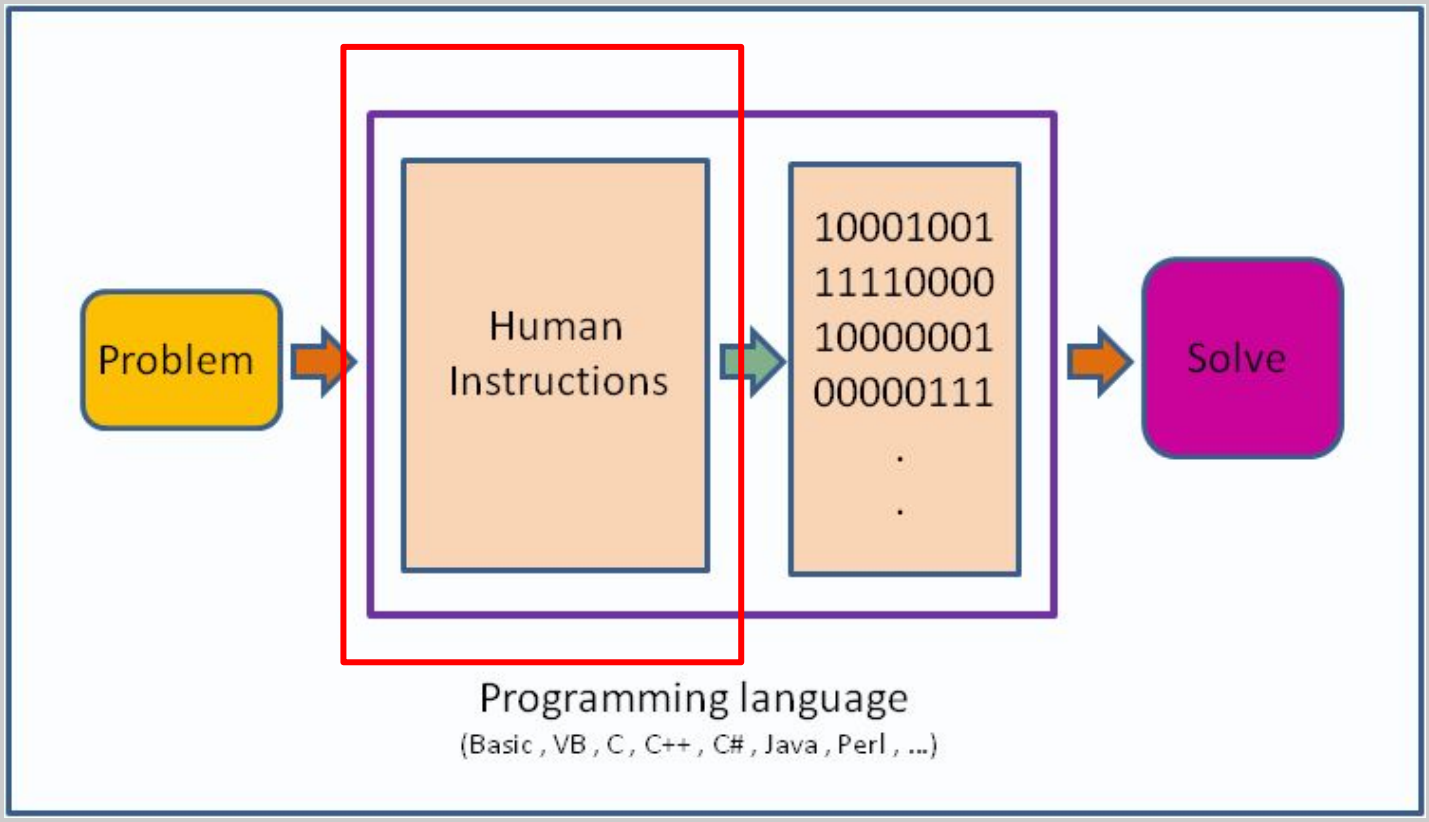
- Pure Python
  - http://www.python.org/download/
- Platforms
  - Anaconda: https://www.anaconda.com/
  - Python(x, y): https://python-xy.github.io/
  - …

Python IDEs

- Integrated Development Environment



Writing Python code

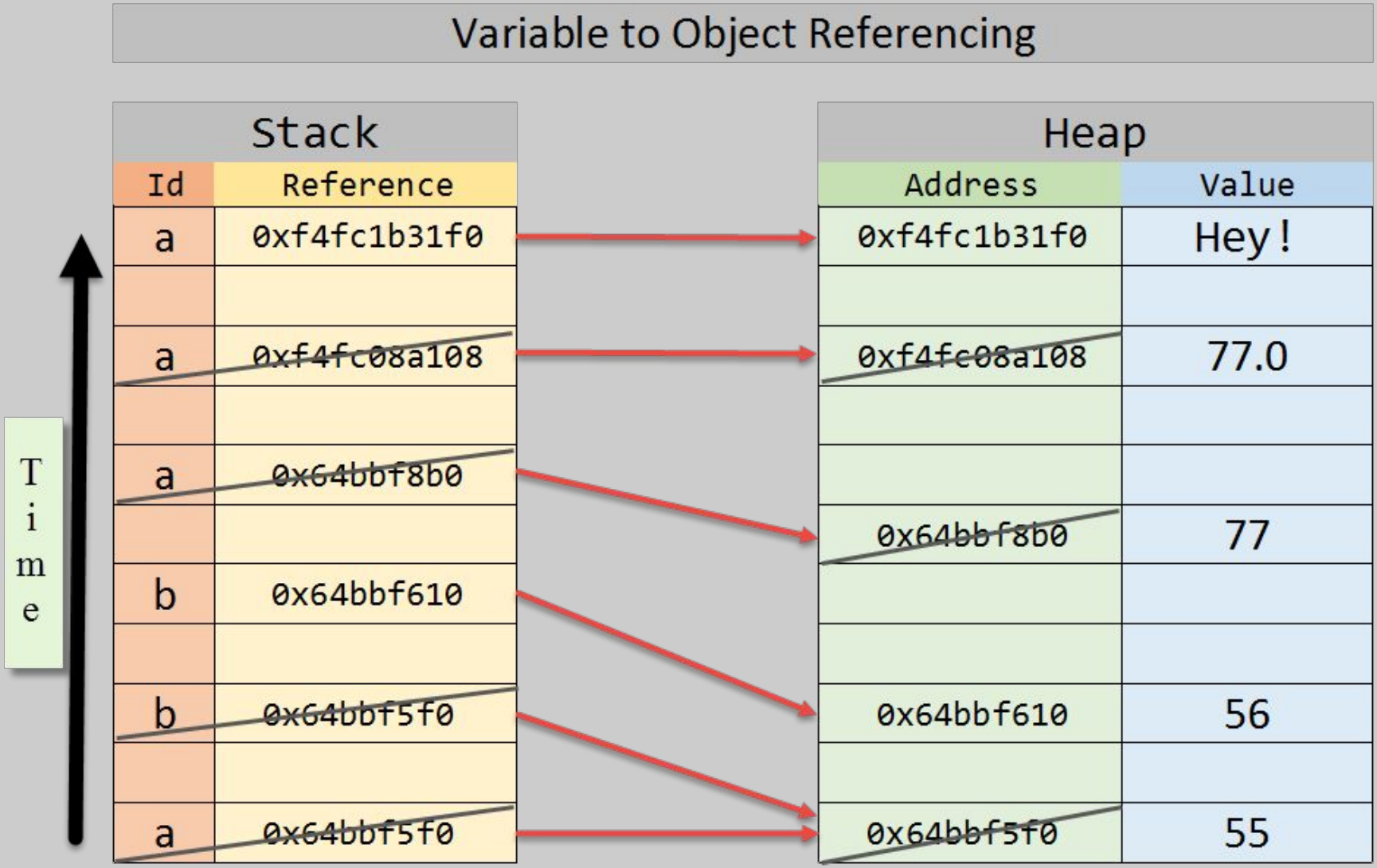


Writing Python code

Generally speaking, a piece of program:

- take in some information
  - variables
- do some operations on the information
  - operators
- give some output

Variables and values



Assignment

- **You create a name the first time it appears on the left side of an assignment expression:**

```
x = 3
```

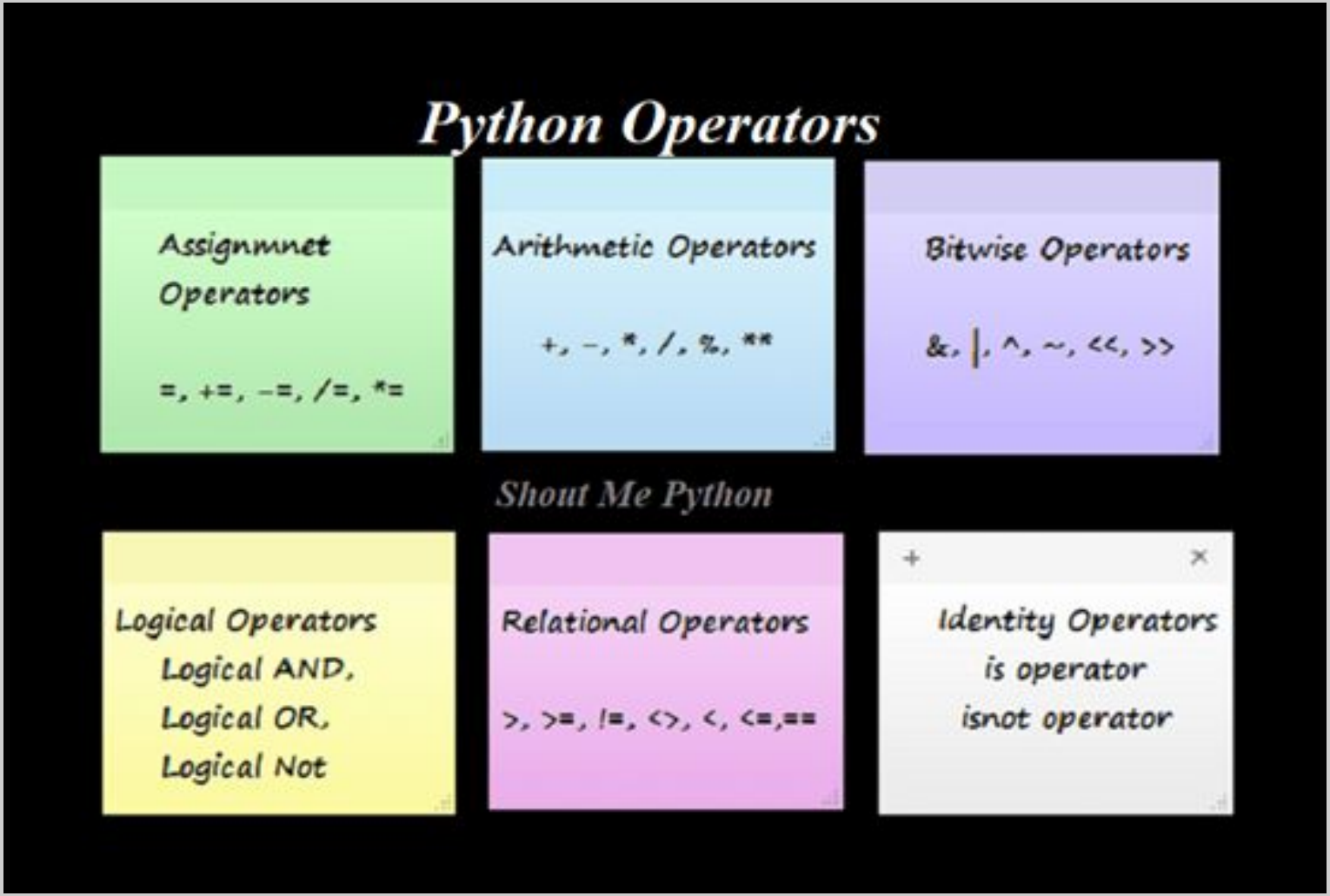
- **Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores.**

```
bob     Bob     _bob     _2_bob_     bob_2     BoB
```

- **There are some reserved words:**

```
and, assert, break, class, continue, def, del,
elif, else, except, exec, finally, for, from,
global, if, import, in, is, lambda, not, or, pass,
print, raise, return, try, while
```

Operators


Python Operators
Assignmnet Operators
=, +=, -=, /=, *=
Arithmetic Operators
+, -, *, /, %, **
Bitwise Operators
&, |, ^, ~, <<, >>
Shout Me Python
Logical Operators
Logical AND,
Logical OR,
Logical Not
Relational Operators
>, >=, !=, <>, <, <=,==
Identity Operators
is operator
isnot operator

Basic syntax

- **Assignment uses `=` and comparison uses `==`, `>`, `<`**
- **For numbers `+ - * / %` are as expected.**

  Special use of **`+`** for string concatenation.

  Special use of **`%`** for string formatting (as with printf in C)
- **Logical operators are words (`and, or, not`) *not* symbols**
- **The basic printing command is `print`.**
- **The first assignment to a variable creates it.**

  Variable types don't need to be declared.

  Python figures out the variable types on its own.

Whitespace/indentation

- **Whitespace is meaningful in Python**
- **No braces { } to mark blocks of code in Python…**
- **Use consistent indentation instead.**

Writing Python code

```
x = 34 - 23 # A comment.
y = “Hello” # Another one.
z = 3.45
if z == 3.45 or y == “Hello”:
   x = x + 1
   y = y + “ World” # String concat.
print(x)
print(y)
```

Running Python code

- Terminal
  - python myscript.py
- Python interpreter
  - execfile('myscript.py') # python 2
  - exec(open('myscript.py').read()) # python 3
- Python IDE (integrated development environment)
  - PyCharm
  - Jupyter Notebook
  - Eclipse
  - Spyder
  - Some text editors (Atom, …)

Basic elements of programs

- Data type and input/output
  - How to receive, store, handle, present or transmit different data
- Algorithms
  - How to perform operations on certain data fast and efficiently
- Control flow
  - How to decide the order of different operations

Data types

- **Simple types**

  `integer, float, bool, complex`

- **Sequence types**

  `string, tuple, list`

- **Dictionary**

Simple types

- **Integer**

```
x = 5, bool, complex
```

- **Float**

```
y = 5.1
```

- **Bool**

```
a = True, b = False
```

- **Complex**

```
z = 3 + 2j
```

Sequence types

- **Tuple**

  A simple ***immutable*** ordered sequence of items

  Items can be of mixed types, including collection types
- **String**

  Immutable

  Conceptually very much like a tuple
- **List**

  Mutable ordered sequence of items of mixed types

Sequence types

- **Tuples are defined using parentheses (and commas).**

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

- **Lists are defined using square brackets (and commas).**

```
>>> li = ["abc", 34, 4.34, 23]
```

- **Strings are defined using quotes (", ', or """").**

```
>>> st = "Hello World"
>>> st = 'Hello World'
>>> st = """This is a multi-line string that uses triple quotes."""
```

Sequence types

- **We can access individual members of a tuple, list, or string using square bracket “array” notation.**
- ***Note that all are 0 based…***

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1]   # Second item in the tuple.
  'abc'
>>> li = ["abc", 34, 4.34, 23]
>>> li[1]   # Second item in the list.
  34
>>> st = "Hello World"
>>> st[1]   # Second character in string.
  'e'
```

Sequence types

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Positive index: count from the left, starting with 0.

```
>>> t[1]
  'abc'
```

Negative lookup: count from right, starting with –1.

```
>>> t[-3]
  4.56
```

Sequence types

Slicing: [start:step:stop]

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Return a copy of the container with a subset of the original members. Start copying at the first index, and stop copying *before* the second index.

```
>>> t[1:4]
  ('abc', 4.56, (2,3))
```

You can also use negative indices when slicing.

```
>>> t[1:-1]
  ('abc', 4.56, (2,3))
```

Sequence types

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Omit the first index to make a copy starting from the beginning of the container.

```
>>> t[:2]
  (23, 'abc')
```

Omit the last index to make a copy starting at the first index and going to the end of the container.

```
>>> t[2:]
  (4.56, (2,3), 'def')
```

Sequence types

To make a *copy* of an entire sequence, you can use `[:]`.

```
>>> t[:]
  (23, 'abc', 4.56, (2,3), 'def')
```

Note the difference between these two lines for mutable sequences:

```
>>> list2 = list1        # 2 names refer to 1 ref
            # Changing one affects both
>>> list2 = list1[:]        # Two independent copies,
two refs
```

The ‘in’ operator

- **Boolean test whether a value is inside a container:**

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
  False
>>> 4 in t
  True
>>> 4 not in t
  False
```

- **For strings, tests for substrings**

```
>>> a = 'abcde'
>>> 'c' in a
  True
>>> 'cd' in a
  True
>>> 'ac' in a
  False
```

The + operator

- **The + operator produces a *new* tuple, list, or string whose value is the concatenation of its arguments.**

```
>>> (1, 2, 3) + (4, 5, 6)
  (1, 2, 3, 4, 5, 6)
>>> [1, 2, 3] + [4, 5, 6]
  [1, 2, 3, 4, 5, 6]
>>> "Hello" + " " + "World"
  'Hello World'
```

The * operator

- **The * operator produces a *new* tuple, list, or string that "repeats" the original content.**

```
>>> (1, 2, 3) * 3
  (1, 2, 3, 1, 2, 3, 1, 2, 3)
>>> [1, 2, 3] * 3
  [1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> "Hello" * 3
  'HelloHelloHello'
```

Sequence types

To make a *copy* of an entire sequence, you can use `[:]`.

```
>>> t[:]
  (23, 'abc', 4.56, (2,3), 'def')
```

Note the difference between these two lines for mutable sequences:

```
>>> list2 = list1        # 2 names refer to 1 ref
            # Changing one affects both
>>> list2 = list1[:]        # Two independent copies,
two refs
```

Iterations

```
>>> sentence = ['Marry', 'had', 'a', 'little', 'lamb']

>>> for word in sentence:
        print(word, len(word))
```

- **Same syntax works for list and tuple**
- **Similar syntax for dictionary**

Dictionary

- **Dictionaries store a mapping between a set of keys and a set of values.**
  - Keys can be any immutable type.
  - Values can be any type
  - A single dictionary can store values of different types
- **You can define, modify, view, lookup, and delete the key-value pairs in the dictionary.**

Dictionary

```
>>> d = {'user':'bozo',
'pswd':1234}
>>> d['user']
  'bozo'
>>> d['pswd']
  1234
>>> d['bozo']
  Traceback (innermost last):
  File '<interactive input>'
line 1, in ?
  KeyError: bozo

>>> d['user'] = 'clown'
>>> d
  {'user':'clown', 'pswd':1234}
>>> d['id'] = 45
>>> d
  {'user':'clown', 'id':45,
'pswd':1234}
```

```
>>> del d['user']     # Remove
one.
>>> d
  {'pswd':1234, 'id':45}
>>> d.clear()     # Remove all.
>>> d
  {}


>>> d = {'user':'bozo',
'p':1234, 'i':34}
>>> d.keys()              # List
of keys.
  ['user', 'p', 'i']
>>> d.values()          # List
of values.
  ['bozo', 1234, 34]
>>> d.items()             # List
of item tuples
  [('user','bozo'), ('p',1234),
('i',34)]
```

Saving and loading data

- **built in open()**
  - only works with strings or byte type
  - human readable
- **pickle**
  - directly save as python object, easy to re-load
  - not human readable
  - can save numpy data types
- **json**
  - Java based encoding/decoding schema
  - normally human readable (open with chrome, notepad, ...)
  - cannot save numpy data types directly (only works for basic python types)

Saving data

```
import json, pickle
import numpy as np

l = [1, 2, 3, [2, 3]] # a basic list
nl = np.random.randn(5) # a numpy array

with open('test.pkl', 'wb') as fh:
    pickle.dump(l, fh)
    pickle.dump(nl, fh)  # both works

with open('test.json', 'w') as fh:
    json.dump(l, fh)   # this works
    json.dump(nl, fh)  # this does not work, nl is not basic type
    json.dump(nl.tolist(), fh)    # this works after type conversion

# be careful using mode 'w', it will truncate existing file
```

Loading data

```
import json, pickle
import numpy as np

with open('test.pkl', 'rb') as fh:
    l = pickle.load(fh)
    nl = pickle.load(fh)  # the same order as saved

with open('test.json', 'r') as fh:
    l = json.load(fh)
    nl = json.load(fh)
```

Control of flow

- **Loops**
  - `for loop`
  - `while loop`
  - `continue/break`
- **Conditional**
  - `if …elif…else`
  - `assert`

Control of flow

```
if x == 3:
    print "X equals 3."
elif x == 2:
    print "X equals 2."
else:
    print "X equals something
else."
print "This is outside the
'if'."
```

```
x = 3
while x < 10:
    if x > 7:
        x += 2
        continue
    x = x + 1
    print "Still in the loop."
    if x == 8:
        break
print "Outside of the loop."
```

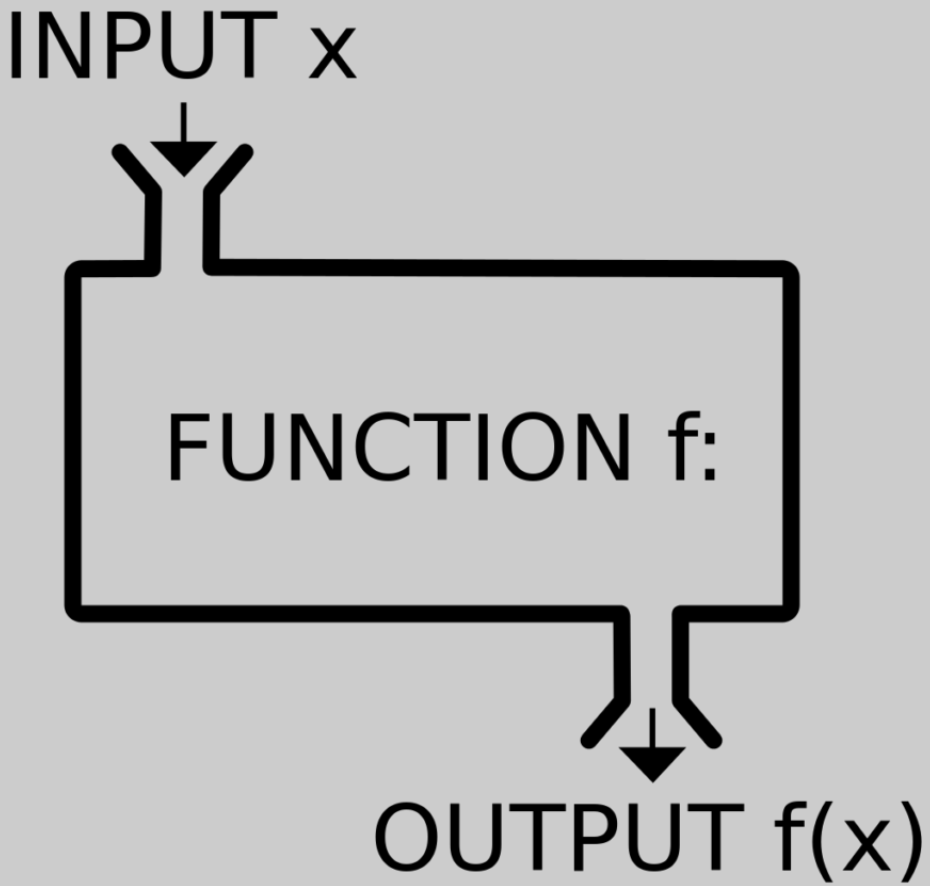
```
assert(number_of_players < 5)
```

```
for x in range(10):
    if x > 7:
        x += 2
        continue
    x = x + 1
    print "Still in the loop."
    if x == 8:
        break
print "Outside of the loop."
```

Functions

- A function is a block of code which only runs when it is called
- You can pass data, known as parameters, into a function
- A function can return data as a result
- For user, it is like a ‘blackbox’ and the user does not need to care what happens inside

Functions



Why functions?

- They allow us to conceive of our program as a bunch of sub-steps. Easy to write, read and understand.
- They allow us to reuse code instead of rewriting it.
- Functions allow us to keep our variable namespace clean (local variables only "live" as long as the function does)
- Functions allow us to test small parts of our program in isolation from the rest.

Functions

- ***def* creates a function and assigns it a name**
- **return sends a result back to the caller**
- **Arguments are passed by assignment**

```
def <name>(arg1, arg2, ..., kwarg1=val1, ..., *args, **kwargs ):
    <statements>
    return <value>

def times(x,y):
    return x*y

def times(x=5, y=10)
    return x*y
```

Functions

```
def print_args(*args):
    for arg in args:
        print(arg)

def times(x, y, a=3, b=2, **kwargs):
    return a*x + b*y
    print(kwargs)
```

Functions

```
def print_args(*args):
    for arg in args:
        print(arg)

def times(x, y, a=3, b=2, **kwargs):
    return a*x + b*y
    print(kwargs)    # this line will not be executed

def times(x, y, a=3, b=2, **kwargs):
    print(kwargs)
    return a*x + b*y
```

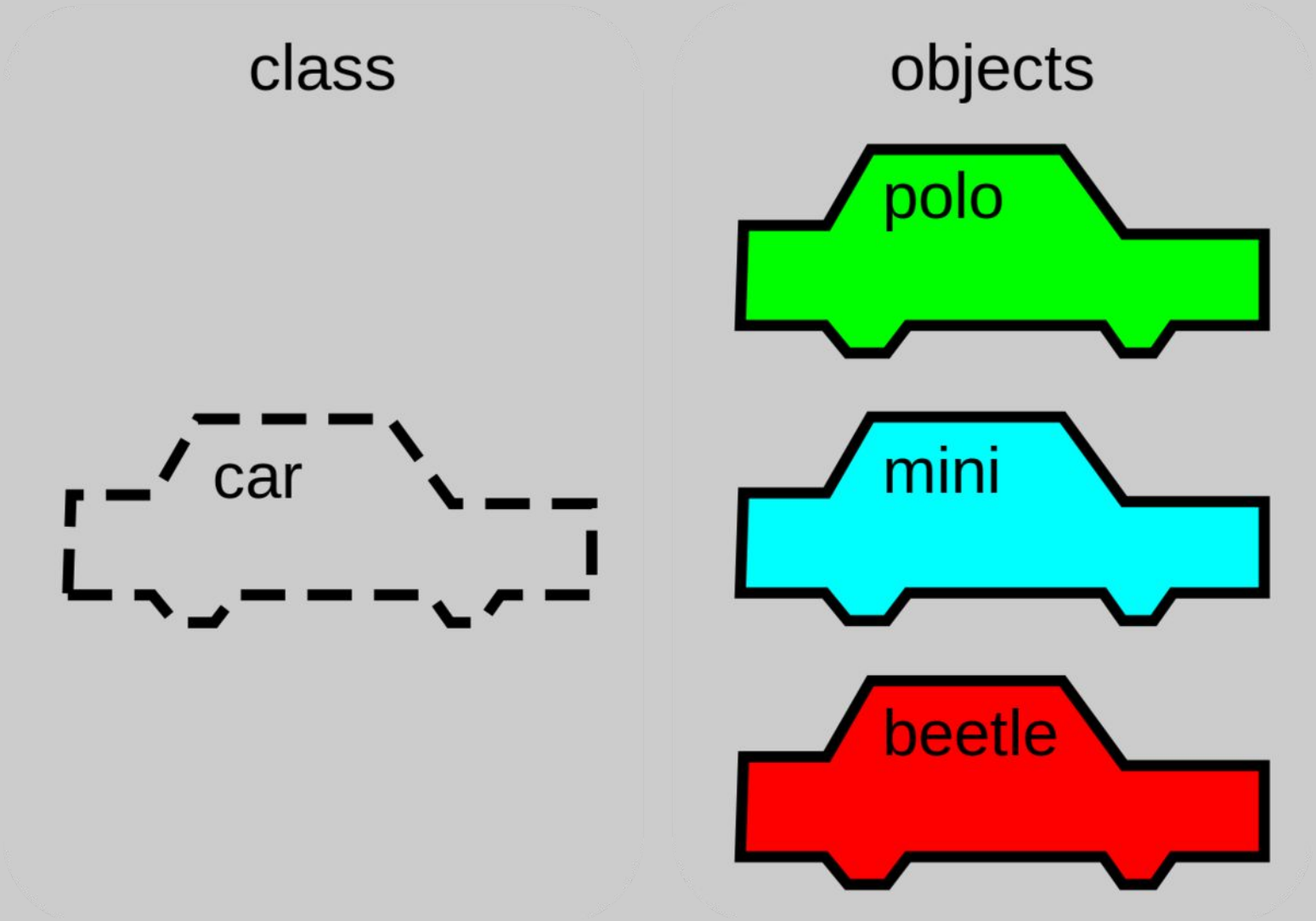
Functions

```
def print_args(*args):
    for arg in args:
        print(arg)

def times(x, y, a=3, b=2, **kwargs):
    return a*x + b*y
    print(kwargs)    # this line will not be executed

def times(x, y, a=3, b=2, **kwargs):
    print(kwargs)
    return a*x + b*y
```

Writing functions

- Understand the purpose of the function.
- Define the data that comes into the function from the caller (in the form of parameters)!
- Define what data variables are needed inside the function to accomplish its goal.
- Decide on the set of steps that the program will use to accomplish this goal. (The Algorithm)

Classes and instances



Modules

- **Modules are functions and variables defined in separate files**
- **Items are imported using from or import**

```
from module import function
function()
import module
module.function()
```

- **Modules are namespaces**

Can be used to organize variable names, i.e.

```
atom.position = atom.position - molecule.position
```

Numpy module

- Fundamental package for scientific computing with Python
- N-dimensional array object
- Linear algebra, Fourier transform, random number capabilities
- Building block for other packages (e.g. Scipy)
- Open source

Array creation

```
import numpy as np
A = np.array([[1, 2, 3], [4, 5, 6]])
A
# [[1 2 3]
# [4 5 6]]

Af = np.array([1, 2, 3], float)
Af.dtype
# dtype('float64')
```

Array creation

```
np.arange(0, 1, 0.2)
# array([ 0. , 0.2, 0.4, 0.6, 0.8])

np.linspace(0, 2*np.pi, 4)
# array([ 0.0, 2.09, 4.18, 6.28])

A = np.zeros((2,3))
# array([[ 0., 0., 0.],
# [ 0., 0., 0.]])
# np.ones, np.diag, np.empty, …
A.shape
# (2, 3)
```

Array slicing

- Very similar to list slicing

```
a = np.random.random((4,5))
a[0,0]
# first row, first column
a[2, :]
# third row, all columns
a[1:3]
# 2nd, 3rd row, all columns
a[:, 2:4]
# all rows, columns 3 and 4
```

Array slicing

- Very similar to list slicing
- However, array[:] will NOT create a copy

Array slicing

- NumPy arrays may be used to index into other arrays (not possible for python list)

```
a = np.arange(15).reshape((3,5))
#array([[ 0, 1, 2, 3, 4],
#[ 5, 6, 7, 8, 9],
#[10, 11, 12, 13, 14]])
i = np.array([[0,1], [1, 2]])
j = np.array([[2, 1], [4, 4]])
a[i, j]    # row [0, 1], [1, 2], column [2, 1], [4, 4]
array([[ 2, 6],
       [ 9, 14]])
```

Array slicing

- Boolean arrays can also be used as indices into other arrays

```
a = np.arange(15).reshape((3,5))
b = (a % 3 == 0)
#array([[ True, False, False, True, False],
#[False, True, False, False, True],
#[False, False, True, False, False]], dtype=bool)

a[b]    # elements in a that are True in b
array([ 0, 3, 6, 9, 12])
```

Array functions

- Predicates

`a.any(), a.all()`

- Reductions

`a.mean(), a.argmin(), a.argmax(), a.trace(), a.cumsum(), a.cumprod()`

- Manipulation

`a.argsort(), a.transpose(), a.reshape(...), a.ravel(), a.fill(...), a.clip(...)`

- Complex Numbers

`a.real, a.imag, a.conj()`

Useful Numpy function

- numpy.where()
- numpy.concatenate()
- numpy.append()
- numpy.reshape()

Numpy subpackages

- **numpy.fft** — Fast Fourier transforms
- **numpy.polynomial** — Efficient polynomials
- **numpy.linalg** — Linear algebra

cholesky, det, eig, eigvals, inv, lstsq, norm, qr, svd

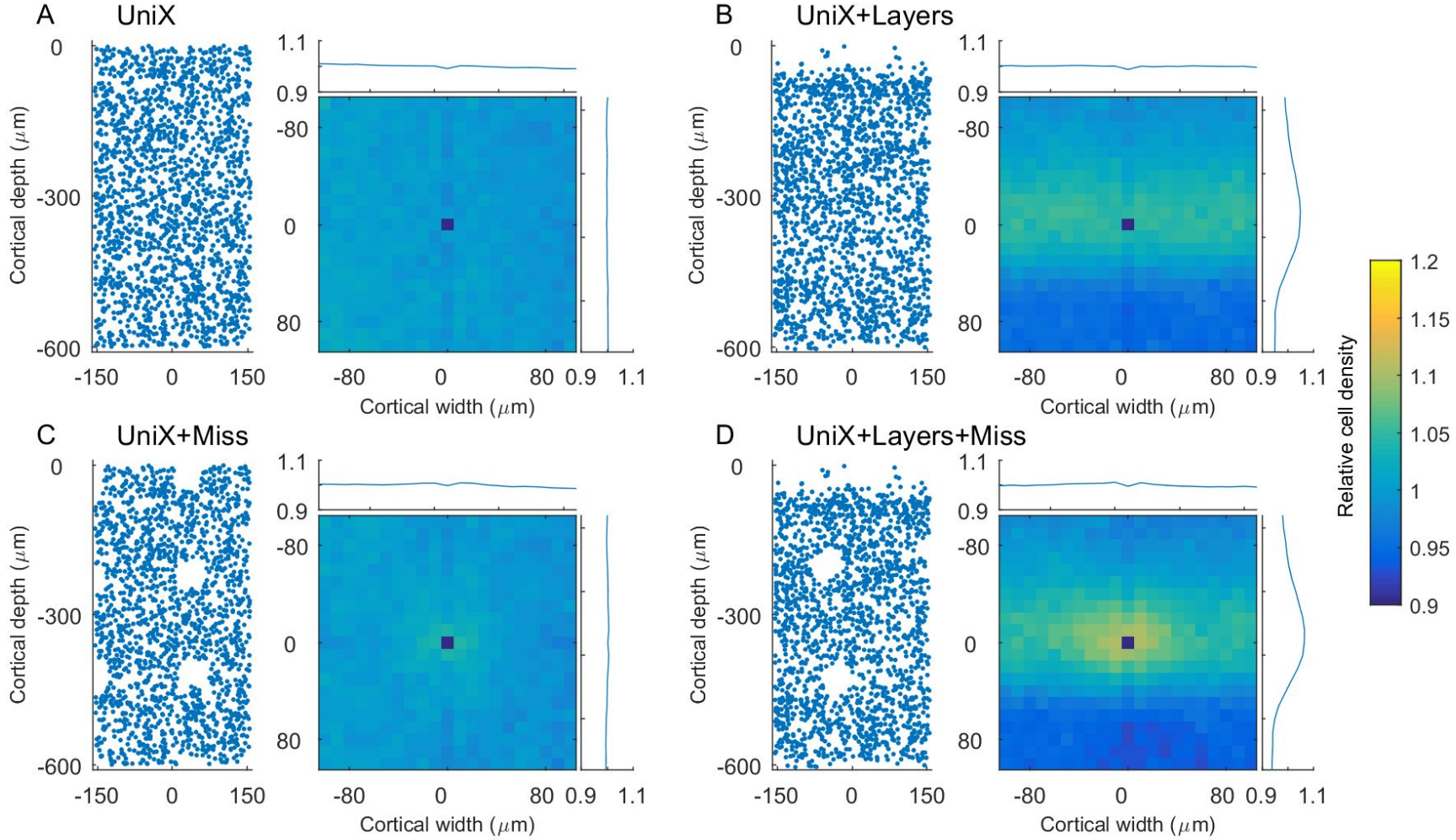
- **numpy.math** — C standard library math functions
- **numpy.random** — Random number generation

beta, gamma, geometric, hypergeometric, lognormal, normal, poisson, uniform, weibull

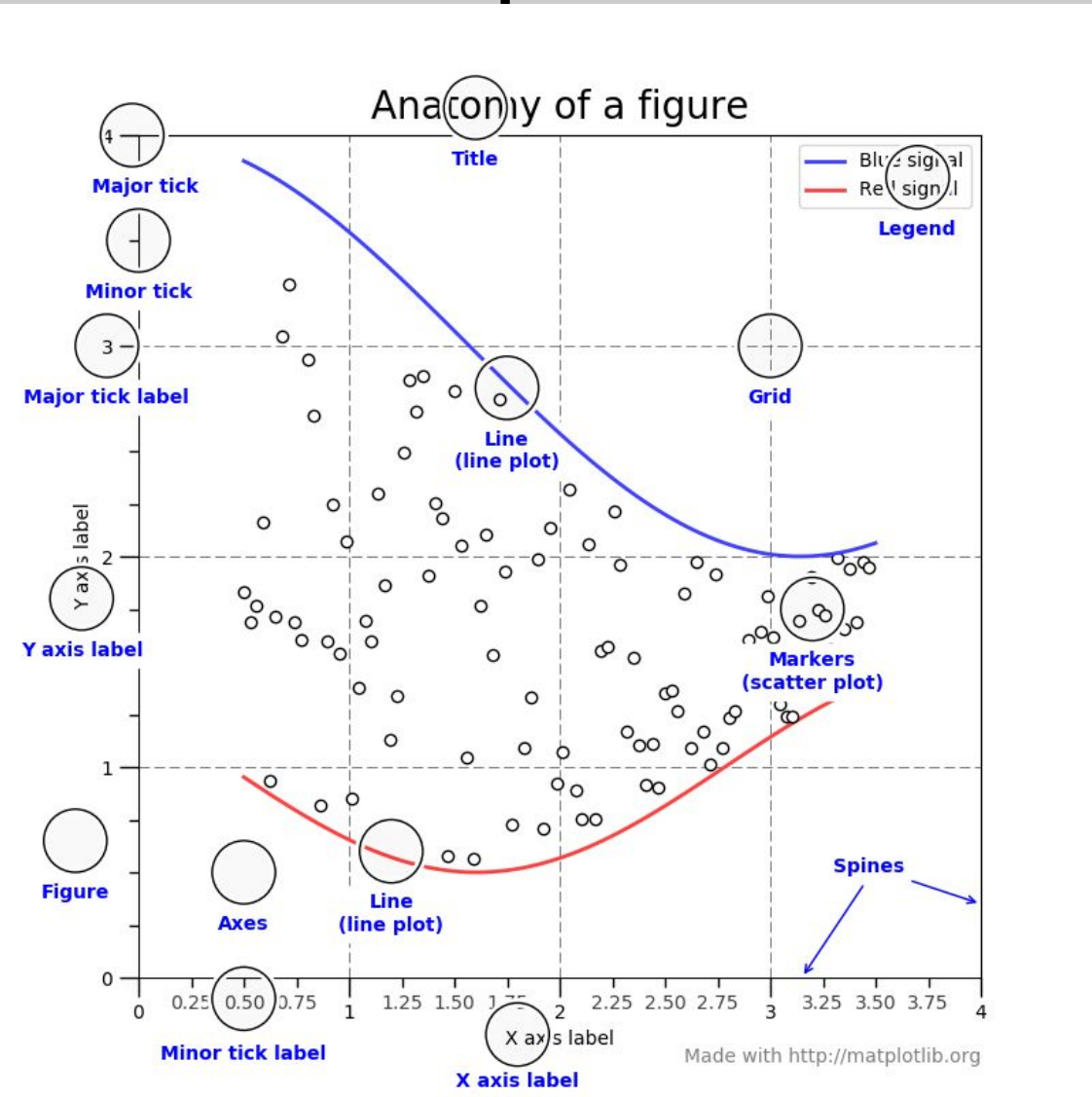
Matplotlib

- **Plotting library for Python**
- **Works well with Numpy**
- **Syntax similar to Matlab**

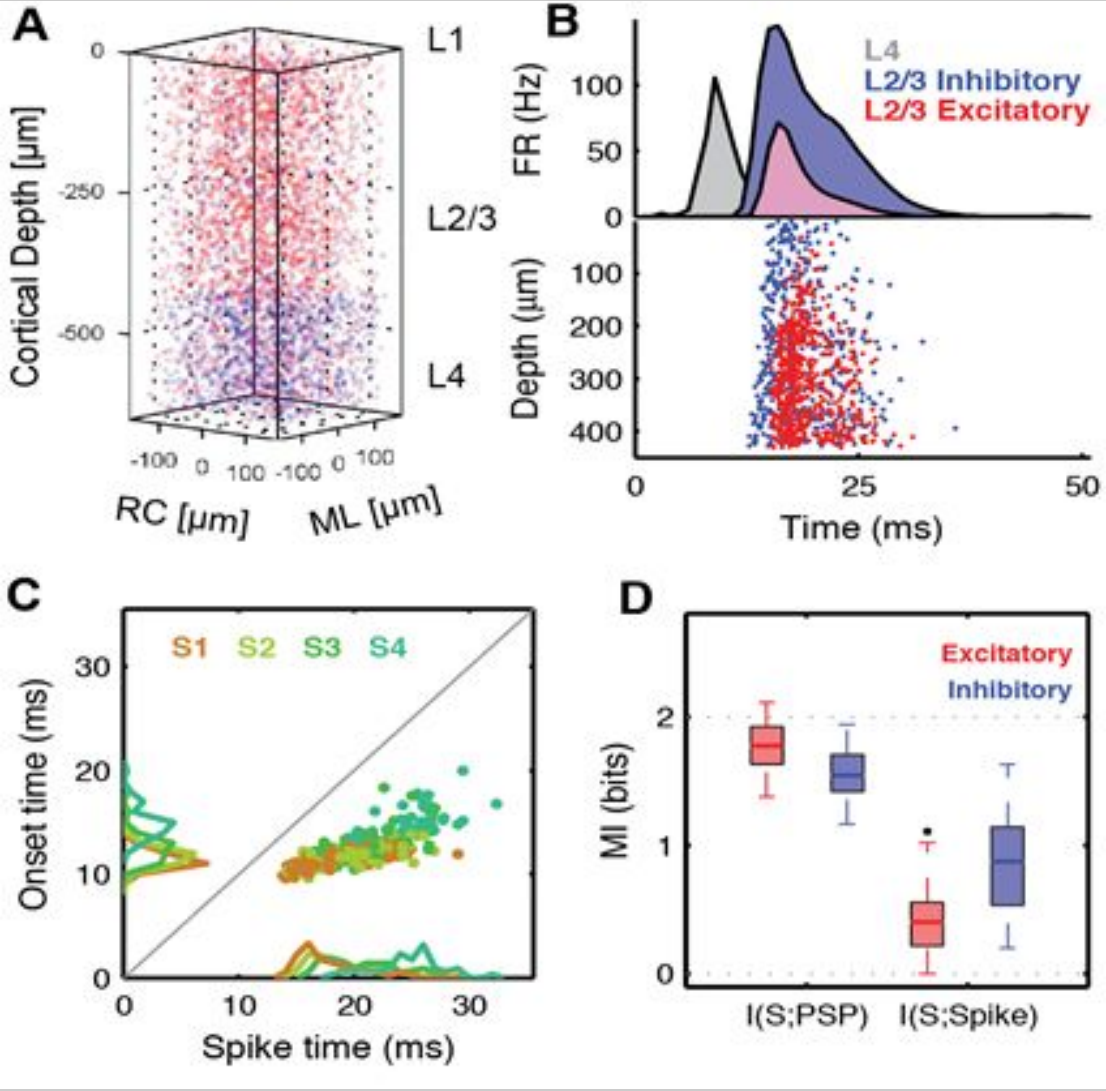
Matplotlib


A
UniX
B
UniX+Layers
C
UniX+Miss
D
UniX+Layers+Miss
Cortical depth (μm)
Cortical width (μm)
Relative cell density
0
-300
-600
-150
150
-80
80
0.9
1.1
1.2
1.15
1.05
1

Matplotlib

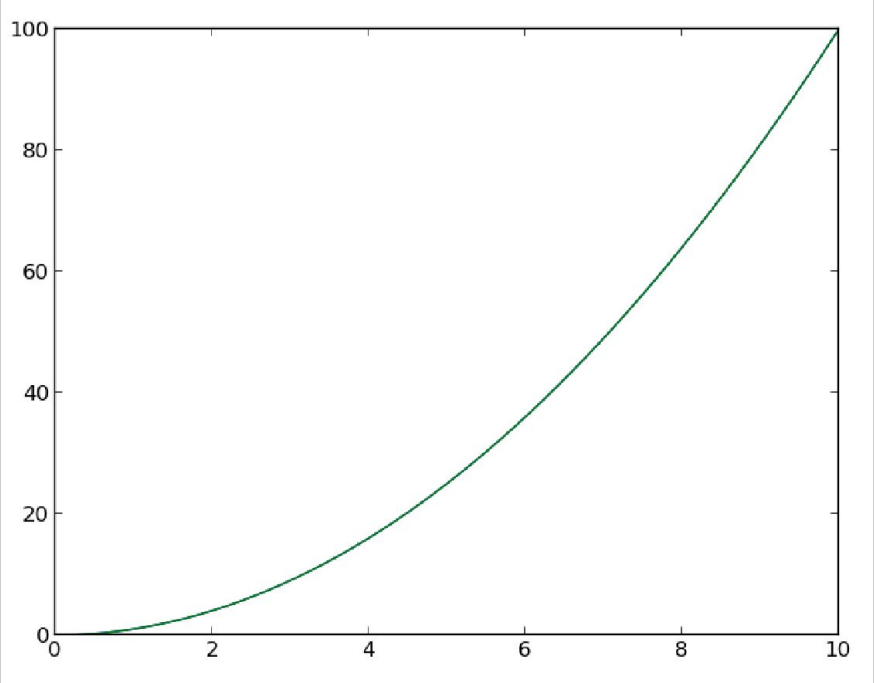

Anatomy of a figure
Title
Major tick
Minor tick
Major tick label
Legend
Grid
Line
(line plot)
Y axis label
Markers
(scatter plot)
Figure
Axes
Line
(line plot)
Spines
Minor tick label
X axis label
Made with http://matplotlib.org

Matplotlib


A
Cortical Depth [µm]
0
-250
-500
L1
L2/3
L4
-100 0 100 -100 0 100
RC [µm]
ML [µm]
B
FR (Hz)
100
50
0
L4
L2/3 Inhibitory
L2/3 Excitatory
Depth (µm)
100
200
300
400
0
25
50
Time (ms)
C
S1 S2 S3 S4
Onset time (ms)
30
20
10
0
0 10 20 30
Spike time (ms)
D
Excitatory
Inhibitory
MI (bits)
2
1
0
I(S;PSP)
I(S;Spike)

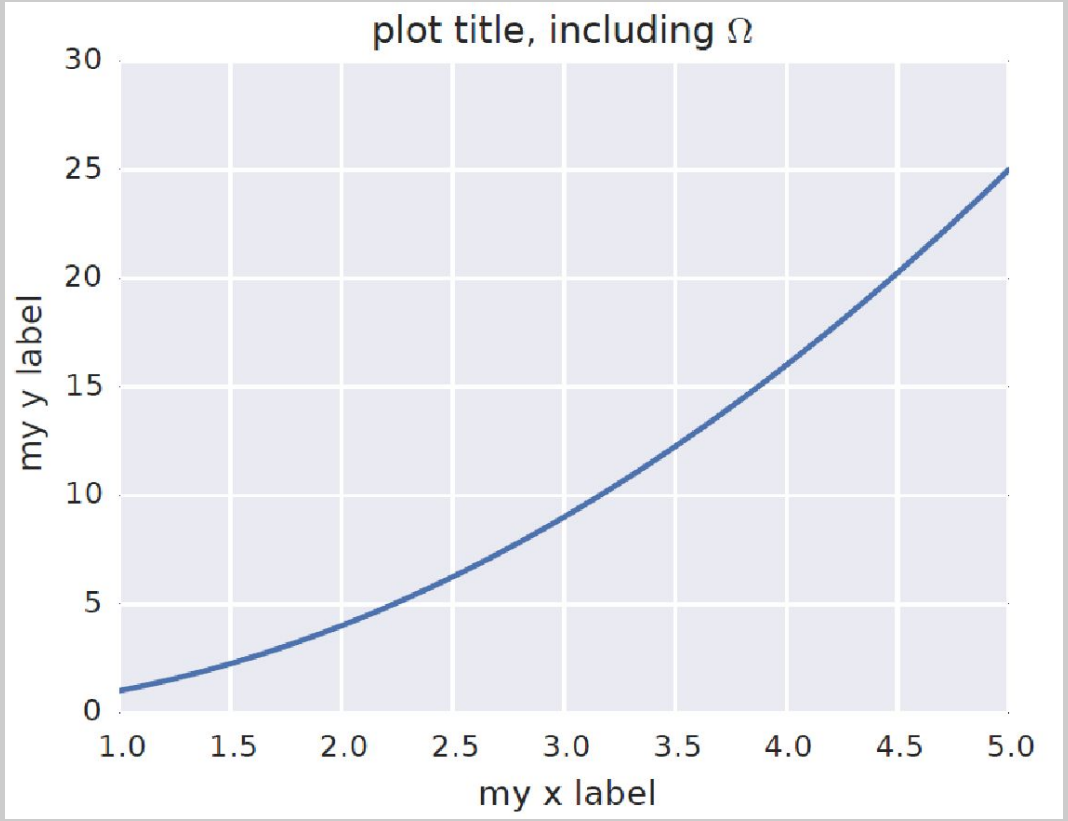
Line plot

```
import numpy as np
import matplotlib.pyplot as plt
x = np. linspace (0 , 10, 1000)
y = np.power(x , 2)
plt.plot (x , y)
plt.show()
```



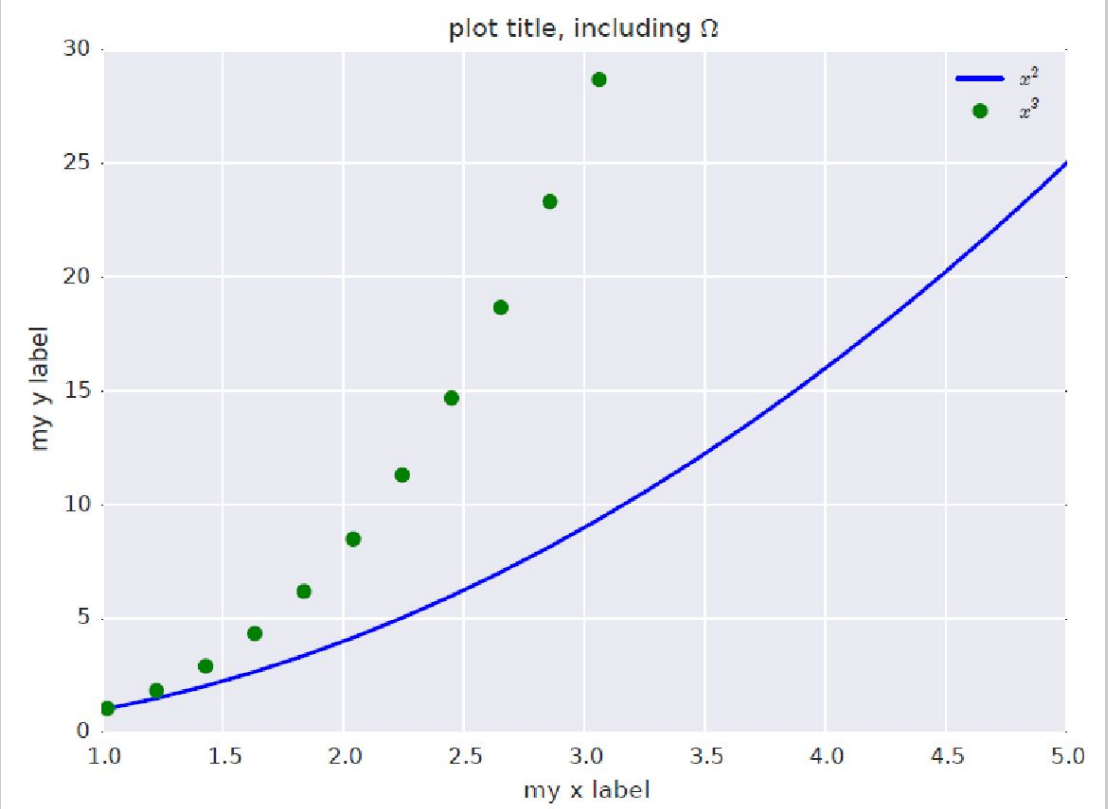
Axis label, title and saving

```
import numpy as np
import matplotlib.pyplot as plt
f, ax = plt.subplots(1, 1, figsize=(5,4))
x = np.linspace(0, 10, 1000)
y = np.power(x, 2)
ax.plot(x, y)
ax.set_xlim((1, 5))
ax.set_ylim((0, 30))
ax.set_xlabel('my x label')
ax.set_ylabel('my y label')
ax.set_title('plot title, including $\Omega$')
plt.tight_layout()
plt.savefig('line_plot_plus.pdf')
```



Multiple lines and legend

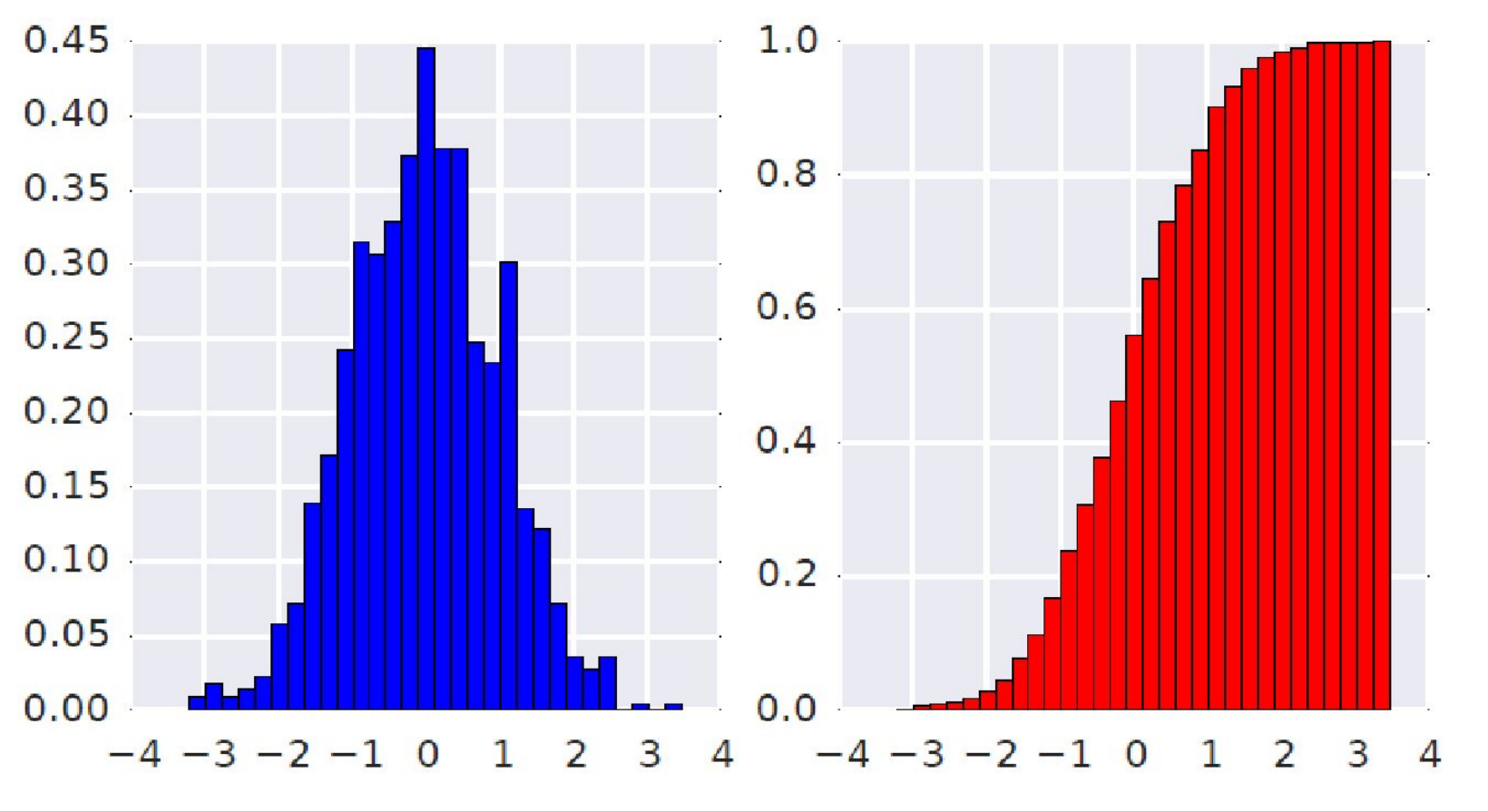
```
x = np.linspace(0 , 10, 50)
y1 = np.power(x , 2)
y2 = np.power(x , 3)
plt.plot (x , y1 , 'b–' ,
label='$x^2$ ' )
plt.plot (x , y2 , 'go ' ,
label='$x^3$ ' )
plt.xlim((1 , 5))
plt.ylim((0 , 30))
plt.xlabel ( 'my x label ' )
plt.ylabel ( 'my y label ' )
plt.title ( 'plot title, including
$\Omega$' )
plt.legend()
```



Histogram

```
data = np.random.randn(1000)

f , (ax1 , ax2) = plt.subplots (1 , 2, figsize=(6,3))
# histogram (pdf)
ax1.hist (data , bins=30, normed=True, color='b ' )
# empirical cdf
ax2.hist (data , bins=30, normed=True, color=' r ' , cumulative=True)
```

Histogram


0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
−4 −3 −2 −1 0 1 2 3 4
1.0
0.8
0.6
0.4
0.2
0.0
−4 −3 −2 −1 0 1 2 3 4

Box plot

```
samp1 = np.random.normal(loc=0., scale=1., size=100)
samp2 = np.random.normal(loc=1., scale=2., size=100)
samp3 = np.random.normal(loc=0.3, scale=1.2, size=100)

f, ax = plt.subplots(1, 1, figsize=(5,4))
ax.boxplot((samp1, samp2, samp3))
ax.set_xticklabels(['sample 1', 'sample 2', 'sample 3'])
```

Box plot

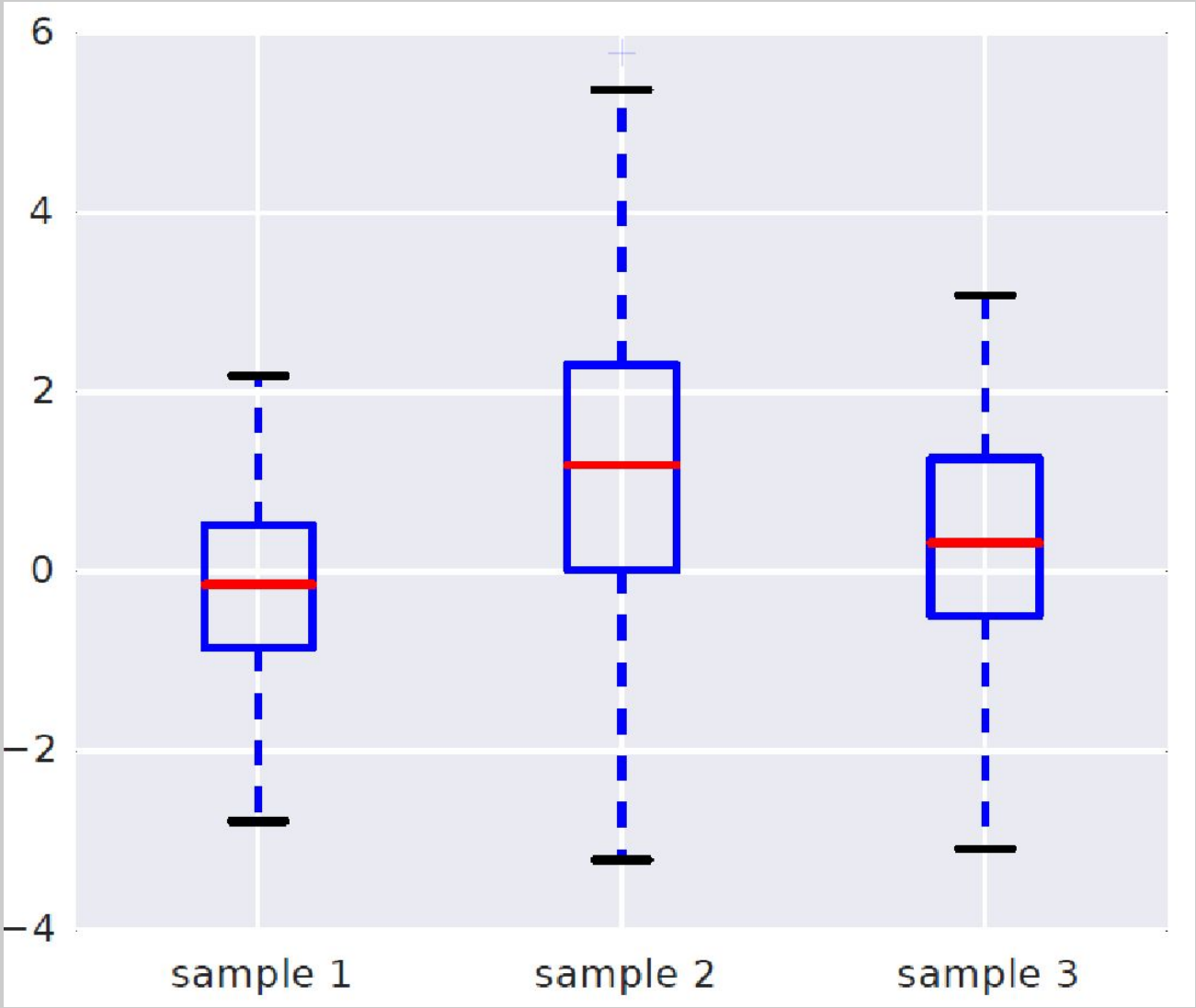
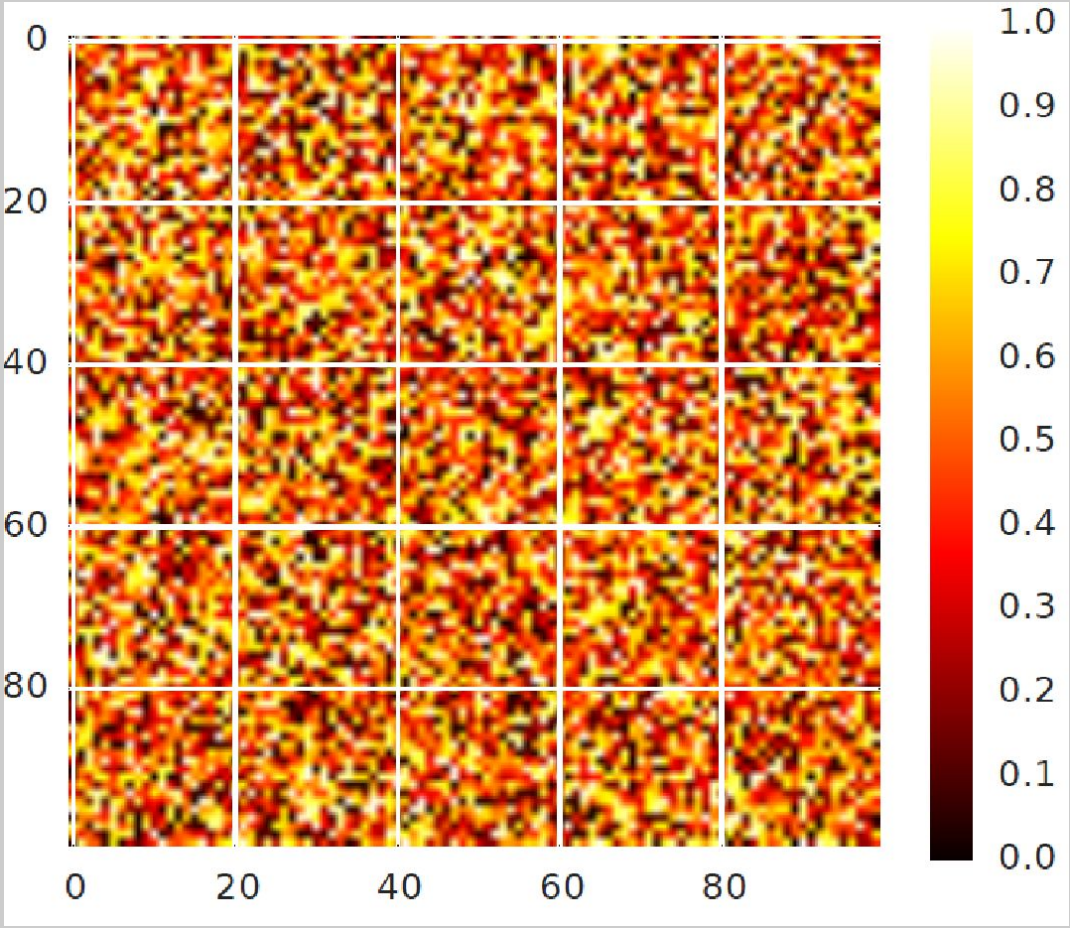


Image plot

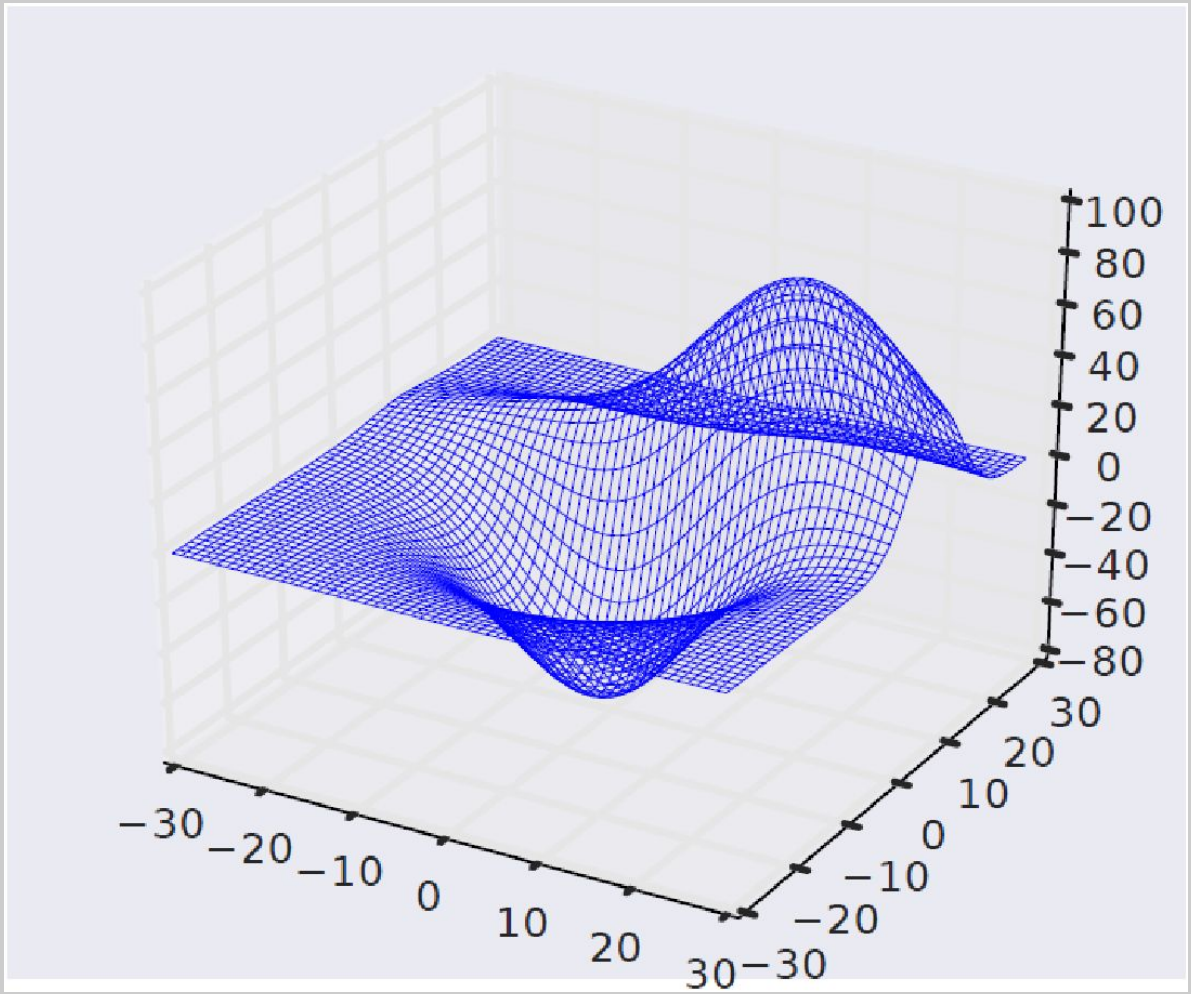
```
A = np.random.random((100,
100))

plt.imshow(A)
plt.hot()
plt.colorbar()
```



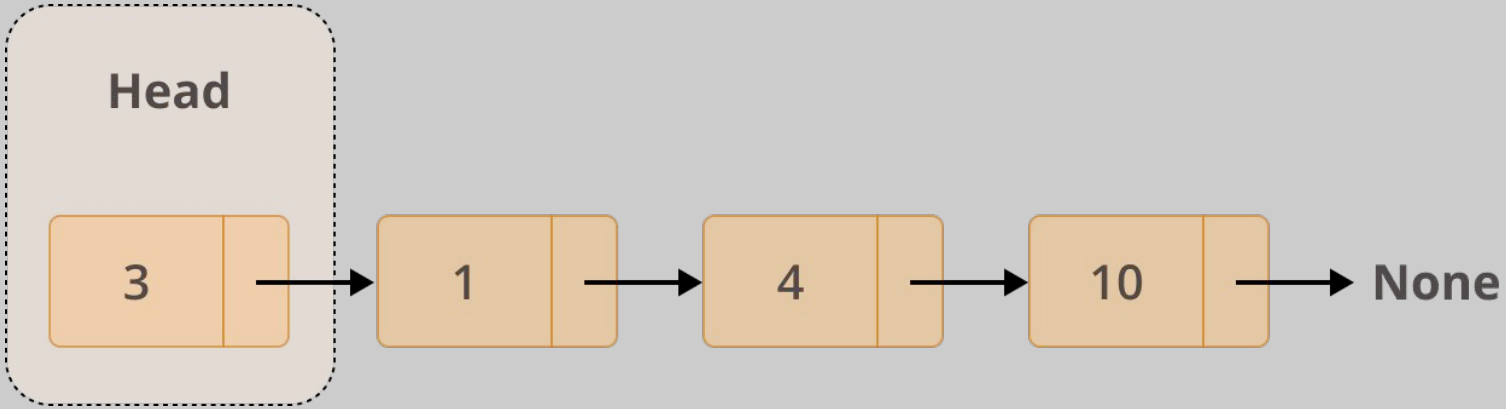
Wire plot (surface plot)

```
from mpl_toolkits.mplot3d import axes3d

ax = plt.subplot(111, projection='3d')

X, Y, Z = axes3d.get_test_data(0.1)

ax.plot_wireframe(X, Y, Z, linewidth=0.1)
```

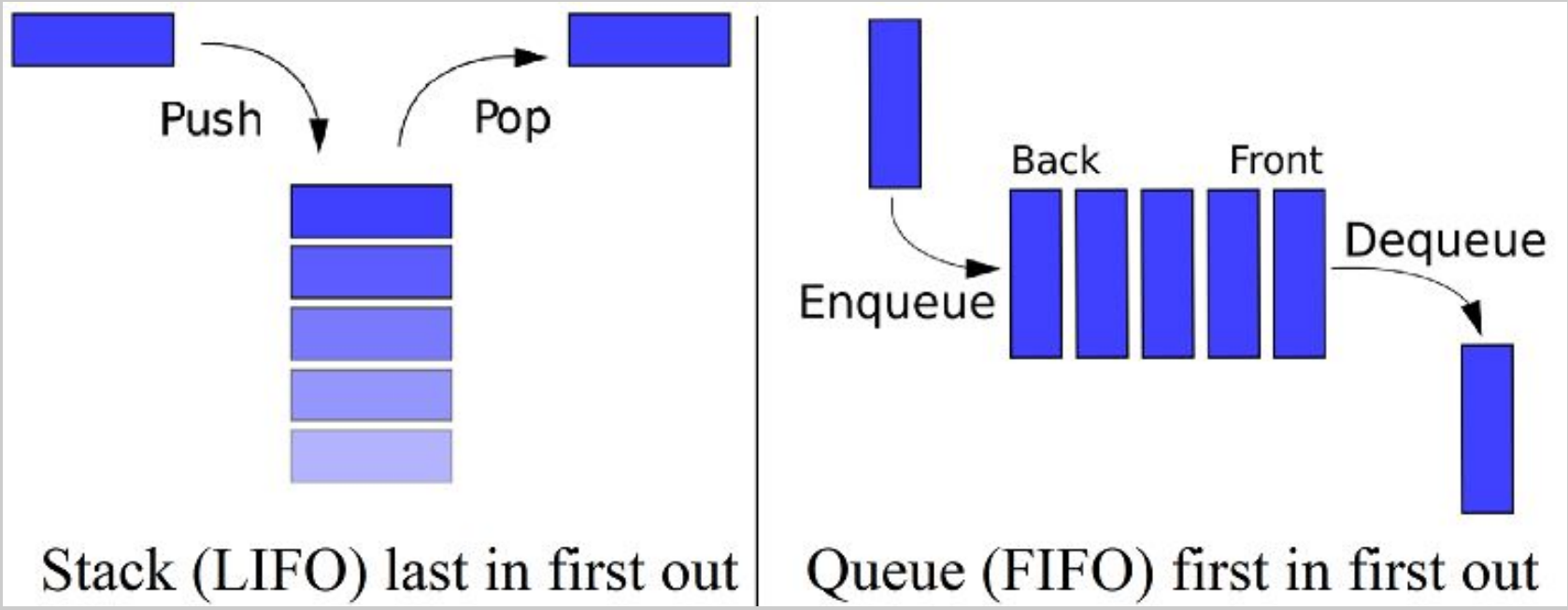
Wire plot (surface plot)



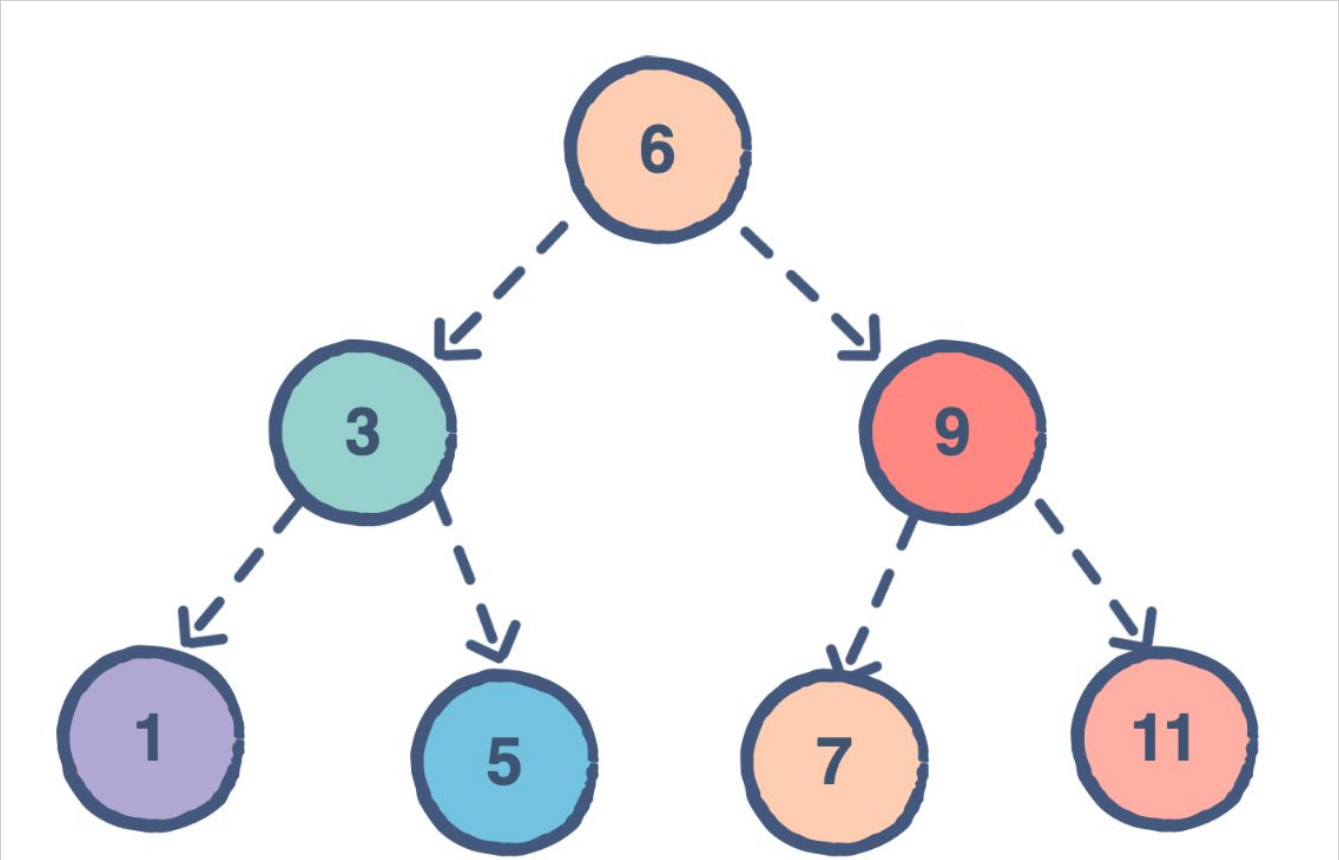
Linked list



Linked list, queue and stack



Binary (search) tree



An example of a binary search tree