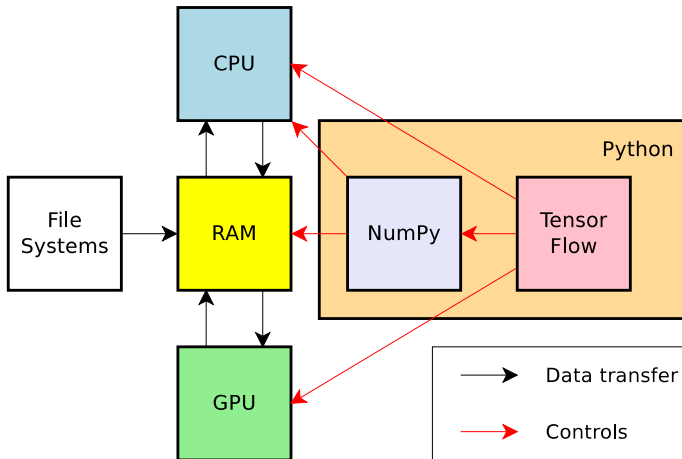


11-695: AI Engineering Python, and NumPy Review

Spring 2019



① Python: a Quick Review

② NumPy: Working with High-Dimensional Data

hello_world.py

```
1 from __future__ import division
2 from __future__ import print_function
3
4 def main():
5     print("Hello World")
6
7 if __name__ == "__main__":
8     main()
```

In your Terminal

```
1 > python hello_world.py
2 Hello World
```

hello_world.py

```
1 from __future__ import division
2 from __future__ import print_function
```

- Clear the nuances between Python2 and Python3
 - `print("Hello World")` instead of `print "Hello World"`
 - `5 // 3` instead of `5 / 3` for integer divisions
 - and much more
- Please always have these lines in Python 2! They are the **future**.

hello_world.py

```
1 def main():
2     print("Hello World")
3
4 if __name__ == "__main__":
5     main()
```

- Indent with **2 spaces** or **1 tab**
 - *Never* mix them! You will get hurt.
- It's **not** the only way to write a Python program
 - but it's the standard. Please always use that¹.
 - You're more than welcome to come up with your convention

¹ <http://google.github.io/styleguide/pyguide.html>

hello_world_with_args.py

```
1 import tensorflow as tf
2
3 flags = tf.app.flags
4 FLAGS = flags.FLAGS
5
6 flags.DEFINE_string(
7     "user_name",           # argument name
8     None,                  # default value
9     "We will greet this person" # help message
10 )
11
12 def main(_args):
13     print("Hello, {}".format(FLAGS.user_name))
14
15 if __name__ == "__main__":
16     tf.app.run()
```

In your Terminal

```
1 > python hello_world_with_args.py --user_name="John"
2 Hello, John!
```

hello_world_with_more_args.py

```
1 # ...imports and others
2 flags.DEFINE_string("user_name", None, "We will greet this person")
3 flags.DEFINE_integer("num_prints", 5, "Number of times to print the message")
4
5 def main(_args):
6     for i in range(FLAGS.num_prints):
7         print("{} Hello, {}!".format(i, FLAGS.user_name))
8
9 if __name__ == "__main__":
10     tf.app.run()
```

In your Terminal

```
1 > python hello_world_with_more_args.py --user_name="John" --num_prints=3
2 1. Hello, John!
3 2. Hello, John!
4 3. Hello, John!
```


- No need to specify types on declaration
- Operations are as normal. Boolean operations are in English.

basic_types.py

```
1 def main(_args):
2     x = 5                                # integer
3     print(type(x))                      # output: <type 'int'>
4     print(x + 1, x - 2, x * 3, x ** 4)  # output: 6 3 15 625
5     x += 6                              # now x = 11
6
7     y = 2.5                             # float
8     print(type(y))                      # output: <type 'float'>
9     print(x + y, type(x + y))           # output: 13.5 <type 'float'>
10
11    z = 25                               # another integer
12    print(z / x, type(z / x))            # float division. output: 2.27273 <type 'float'>
13    print(z // x, type(z // x))          # integer division. output: 2 <type 'int'>
14
15    t = True                             # boolean: True, False
16    print(t, type(t))                    # output: True <type 'bool'>
17    u = (5 > 3)                           # Comparisons return True or False
18    v = (2 == 7)                          # There are: >, <, >=, <=, ==, !=
19    print(u and v, u or v, not u, u and not v) # False True False True
```

string_examples.py

```
1 def main(_args):
2     s = "Tensor"      # this is a string
3     print(type(s))    # output: <type 'str'>
4     print(s[1])       # output: e
5
6     t = 'Flow'        # '...' and "..." are both ok, but do NOT mix them!
7     print(len(t))     # output: 4
8     w = s + t         # + means concatenation. w is "TensorFlow"
9     print(w)         # output: TensorFlow
10
11     m = "{} and {} are {} platforms".format(w, "PyTorch", 2) # "{}".format(...)
12     print(m)         # output: TensorFlow and PyTorch are 2 platforms.
13
14     # A lot of built-in functions. Some examples:
15     print(m.upper())  # output: TENSORFLOW AND PYTORCH ARE 2 PLATFORMS
16     print(m.replace("o", "xx")) # output: Tens?rFl?w and PyT?rch are 2 platf?rms
```

- Strings are *immutable*
 - `s[3] = "h"` won't work!

list_examples.py

```
1 # ...imports and others
2 def main(_args):
3     a = [1, 2, 23, 2, 1, 27, 21] # this is a list
4
5     print(a, type(a))          # [1, 2, 23, 2, 1, 27, 21] <type 'list'>
6     print(a[0], a[3])          # index from 0. output: 1 2
7     print(a[-1], a[-2])        # -1 means last element, -2 means next-to-last. output: 21 27
8
9     a[0] = 7                   # unlike string, list is mutable
10    print(a)                    # output: [7, 2, 23, 2, 1, 27, 21]
11
12    a[1] = "hello"              # list can also contain different types
13    a[4] = ["tensor", "flow"]   # even another list
14    print(a)                    # output: [7, 'hello', 23, 2, ['tensor', 'flow'], 27, 21]
15
16    b = [2.5, 6, 1.7]          # this is another list
17    c = a + b                   # just like for strings, + means contatenation for lists
18    print(c)                    # output: [7, 'hello', 23, 2, ['tensor', 'flow'], 27, 21, 2.5, 6, 1.7]
```

- List are extremely flexible and important in Python.

list_vs_set.py

```
1 l = [1, 2, 3, 4, 5]
2 type(l)
3 for i in l:
4     i = i * 2
5 print(l) # result?
6
7 t = (1, 2, 3, 4, 5)
8 type(t)
9 print t[1]
10 t[1] = 10
11
12 s = set((1, 2, 3, 4, 5))
13 type(s)
14 print s[1]
15 s[1] = 10
16
17 x = [10 * i for i in range(100)]
18 type(x)
19
20 y = (10 * i for i in range(100))
21 type(y)
```

dict_examples.py

```
1 # ...imports and others
2 def main(_args):
3     d = {                # creates a dictionary using {...}
4         "hello": "world", # pairs of key: value, separated by a comma
5         1: "TensorFlow",  # key and value can be of any types
6         6.0: [2, 2, 3]
7     }
8
9     print(d, type(d))
10    # output: {1: 'TensorFlow', 'hello': 'world', 6.0: [2, 2, 3]} <type 'dict'>
11
12    print(d["hello"], d[6.0])
13    # output: "world" [2, 2, 3]
14
15    d["PyTorch"] = {"author": "Facebook", "version": 2.0}
16    # add an element, which is itself a dict
17
18    print(d["PyTorch"])
19    # output: {"author": "Facebook", "version": 2.0}
```

loop_examples.py

```
1 # ...imports and others
2 def main(_args):
3     for i in range(10): # basic for loop
4         print(i)
5
6     my_list = [3, 21, 4, 3.14, "numpy", 18, 281, "tensorflow"]
7     for my_value in my_list: # loop through a list
8         print(my_value)
9     for i in range(len(my_list)): # this also works, but slower!
10        print(my_list[i])
11    for i, my_value in enumerate(my_list): # do this if you want the index
12        print("Element at {} is {}".format(i, my_value))
13
14    a = [val for val in my_list if isinstance(val, float)] # all floats
15    print(a) # output: [3.14]
16    b = [val for i, val in enumerate(my_list) if i % 2 == 0] # even-indexed
17    print(b) # output: [3, 4, "numpy", 281]
18
19    # while loop
20    x = 10
21    while x <= 20:
22        print("Now we have x = {}".format(x))
23        x += 2
```

① Python: a Quick Review

② NumPy: Working with High-Dimensional Data

- `numpy.ndarray` is a **type** to store and manipulate high-dimensional data. It's *much faster* than `list`.
- Each `numpy.ndarray` has a `dtype`. You should think of `dtype` as the array's *data type*.

numpy_intro.py

```
1 import numpy as np
2
3 def main(_args):
4     x = [1, 2, 4, 2, 56, 21, 12, 421]
5     print(x)           # output: [1, 2, 4, 2, 56, 21, 12, 421]
6     print(type(x))     # output: <type 'list'>
7
8     x_np = np.array(x) # creates a numpy 1-dimensional array
9     print(x_np)        # np-looking style: [ 1  2  4  2 56 21 12 421]
10    print(type(x_np))   # output: <type 'numpy.ndarray'>
11    print(x_np.dtype)   # output: int64
12
13    y_np = x_np.astype(np.int32) # cast the dtype
14    print(type(y_np))          # output: <type 'numpy.ndarray'>
15    print(y_np.dtype)          # output: int32
```


- `numpy.ndarray` is a **type** to store and manipulate high-dimensional data. It's *much faster* than `list`.
- Each `numpy.ndarray` has a `dtype`. You should think of `dtype` as the array's *data type*.

`ndarray.py`

```
1 import numpy as np
2
3 x = [1, 2, 4, 2, 56, 21, 12, 421]
4
5 y = np.ndarray(x)
6 print(type(y))
7
8 z = np.ndarray((2, 3))
9 print(type(z))
```

NumPy: rank, shape, and size

- rank: number of dimensions
 - This is **not** the *matrix rank* in linear algebra
- shape: size in each dimension
- size: total number of elements

numpy_rank_shape_size.py

```
1 def main(_args):
2     x = [[[ 2,  4, 18,  1],
3           [ 9,  1,  2, 12]],
4          [[12, 12, 65, 94],
5           [92, -1, 82, -8]],
6          [[93, -6,  0, 91],
7           [78, 81,  8, -1]]]
8     x_np = np.array(x, dtype=np.int32)
9
10    print(np.ndim(x_np))    # output: 3. It used to be np.rank(x), but was updated.
11    print(np.shape(x_np))  # output: (3, 2, 4)
12    print(np.size(x_np))   # output: 24
```

numpy_rank_shape_size.py

```
1 def main(_args):
2     x = [[[ 2,  4, 18,  1],
3           [ 9,  1,  2, 12]],
4          [[12, 12, 65, 94],
5           [92, -1, 82, -8]],
6          [[93, -6,  0, 91],
7           [78, 81,  8, -1]]]
8     x_np = np.array(x, dtype=np.int32)
9
10    print(np.ndim(x_np))    # output: 3. It used to be np.rank(x), but was updated.
11    print(np.shape(x_np))   # output: (3, 2, 4)
12    print(np.size(x_np))    # output: 24
```

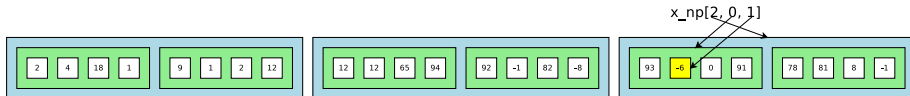
- NumPy arrays are *row major*
- The numbers are stored in your computer's memory as follows



numpy_memory_demonstration.py

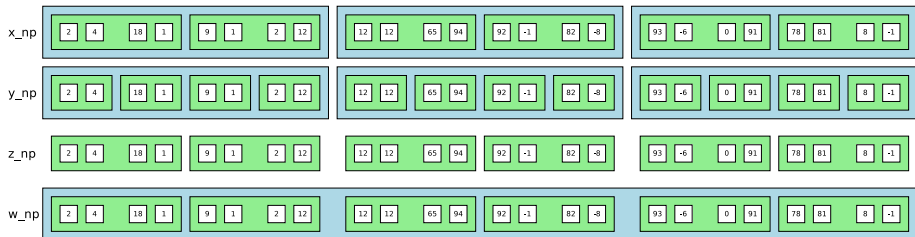
```
1 def main(_args):  
2     # create x_np as before  
3     print(x_np[2, 0, 1]) # output: -6
```

- NumPy arrays are *row major*
- An access to an element happens as follows



numpy_reshape.py

```
1 def main(_args):
2     # create x_np as before
3     y_np = np.reshape(x_np, [3, 4, 2])
4     z_np = np.reshape(y_np, [6, 4])
5     w_np = np.reshape(x_np, [1, 6, 4])
```



- When you call `np.reshape`, memory stays the same
- Only the memory layout changes
- `y_np`, `z_np`, and `w_np` points to *the same memory* with `x_np`

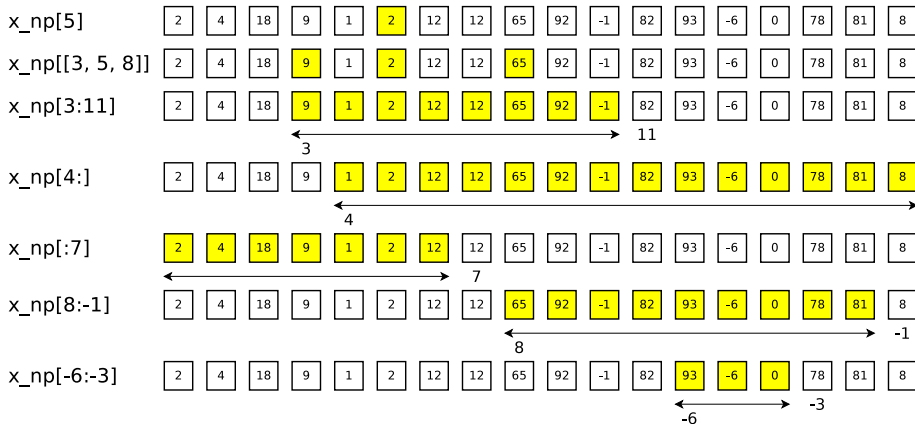
numpy_reshape.py

```
1 def main(_args):
2     # create x_np as before
3     print(np.shape(x_np)) # output: (3, 2, 4)
4
5     y_np = np.reshape(x_np, [-1, 4, 2])
6     print(np.shape(y_np)) # output: (3, 4, 2)
7
8     z_np = np.reshape(x_np, [8, -1])
9     print(np.shape(z_np)) # output: (8, 3)
```

- If you know `x_np` and *all but one* reshaping dimensions
 - then you also know the remaining dimension
- `numpy` allows you not to worry about the missing dimension
 - By using `-1` at *no more than one* dimension

numpy_indexing.py

```
1 def main(_args):
2     x = [2, 4, 18, 9, 1, 2, 12, 12, 65, 92, -1, 82, 93, -6, 0, 78, 81, 8]
3     x_np = np.array(x, dtype=np.int32)
```



- Quick quizz: what do the followings return?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
2	4	18	9	1	2	12	12	65	92	-1	82	93	-6	0	78	81	8

`x_np[[7, 4, 5]]` `x_np[-5:]` `x_np[:-7]` `x_np[[7, -1]]` `x_np[10:-3]`

- Quick quizz: what do the followings return?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
2	4	18	9	1	2	12	12	65	92	-1	82	93	-6	0	78	81	8

`x_np[[7, 4, 5]]` `x_np[-5:]` `x_np[:-7]` `x_np[[7, -1]]` `x_np[10:-3]`

- How do you select everything in `x_np`?

- Quick quizz: what do the followings return?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
2	4	18	9	1	2	12	12	65	92	-1	82	93	-6	0	78	81	8

`x_np[[7, 4, 5]]` `x_np[-5:]` `x_np[:-7]` `x_np[[7, -1]]` `x_np[10:-3]`

- How do you select everything in `x_np`?
 - `x_np[0:]`

- Quick quizz: what do the followings return?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
2	4	18	9	1	2	12	12	65	92	-1	82	93	-6	0	78	81	8

`x_np[[7, 4, 5]]` `x_np[-5:]` `x_np[:-7]` `x_np[[7, -1]]` `x_np[10:-3]`

- How do you select everything in `x_np`?
 - `x_np[0:]`
 - Here's a better way: `x_np[:]`

NumPy: Summary on Indexing 1-dim Arrays

- `x_np[5]`: one element
- `x_np[[7, 5, 2, 9]]`: indexing by a list of indices
- `x_np[3:7]`: indexing by range, right-hand side is exclusive
- `x_np[6:]`: indexing from an index (inclusive)
- `x_np[:8]`: indexing to an index (exclusive)
- `x_np[:-1]`: last element is indexed by -1
- `x_np[-2]`: next-to-last element is indexed -2
- `x_np[:]`: take everything

- Think of each dimension is a 1-dim array.

numpy_indexing_2_dim.py

```
1 def main(_args):
2     x = [[2, 4, 18, 9, 1, 2], [12, 12, 65, 92, -1, 82], [93, -6, 0, 78, 81, 8]]
3     x_np = np.array(x, dtype=np.int32)
```

<code>x_np[1, 2]</code>	2 4 18 9 1 2	12 12 65 92 -1 82	93 -6 0 78 81 8
<code>x_np[0, 2:5]</code>	2 4 18 9 1 2	12 12 65 92 -1 82	93 -6 0 78 81 8
<code>x_np[0:2, 1:3]</code>	2 4 18 9 1 2	12 12 65 92 -1 82	93 -6 0 78 81 8
<code>x_np[-1, :]</code>	2 4 18 9 1 2	12 12 65 92 -1 82	93 -6 0 78 81 8
<code>x_np[:, 1:2]</code>	2 4 18 9 1 2	12 12 65 92 -1 82	93 -6 0 78 81 8

- For higher-dim arrays, think of each dim as a 1-dim index.
- There are *other ways* to index **numpy** arrays
 - but avoid them if possible. They are extremely confusing.
 - When confused, try a toy one and validate!
- Now try the following quiz:



`x_np[:, :, 2]` `x_np[:, :-1, :-1]` `x_np[2, :, :]` `x_np[2, -2:, 1]` `x_np[2, -2:, 1:2]`

numpy_transpose_2_dim.py

```
1 def main(_args):
2     x = [[2, 4, 18, 9, 1, 2], [12, 12, 65, 92, -1, 82], [93, -6, 0, 78, 81, 8]]
3     x_np = np.array(x, dtype=np.int32)
4     y_np = np.transpose(x_np)
```

- Transpose of a 2-dim array is just like transpose of a matrix

$$\begin{bmatrix} 2 & 4 & 18 & 9 & 1 & 2 \\ 12 & 12 & 65 & 92 & -1 & 82 \\ 93 & -6 & 0 & 78 & 81 & 8 \end{bmatrix} \longrightarrow \begin{bmatrix} 2 & 12 & 93 \\ 4 & 12 & -6 \\ 18 & 65 & 0 \\ 9 & 92 & 78 \\ 1 & -1 & 81 \\ 2 & 82 & 8 \end{bmatrix}$$

numpy_transpose_2_dim.py

```
1 def main(_args):
2     x = [[2, 4, 18, 9, 1, 2], [12, 12, 65, 92, -1, 82], [93, -6, 0, 78, 81, 8]]
3     x_np = np.array(x, dtype=np.int32)
4     y_np = np.transpose(x_np)
```

- Transpose of a 2-dim array is just like transpose of a matrix

$$\begin{bmatrix} 2 & 4 & 18 & 9 & 1 & 2 \\ 12 & 12 & 65 & 92 & -1 & 82 \\ 93 & -6 & 0 & 78 & 81 & 8 \end{bmatrix} \longrightarrow \begin{bmatrix} 2 & 12 & 93 \\ 4 & 12 & -6 \\ 18 & 65 & 0 \\ 9 & 92 & 78 \\ 1 & -1 & 81 \\ 2 & 82 & 8 \end{bmatrix}$$

- What happens to the memory of `y_np`?

numpy_transpose_2_dim.py

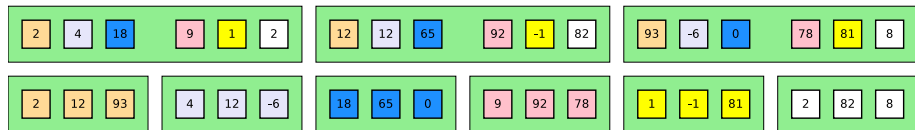
```
1 def main(_args):
2     x = [[2, 4, 18, 9, 1, 2], [12, 12, 65, 92, -1, 82], [93, -6, 0, 78, 81, 8]]
3     x_np = np.array(x, dtype=np.int32)
4     y_np = np.transpose(x_np)
```

$$\begin{bmatrix} 2 & 4 & 18 & 9 & 1 & 2 \\ 12 & 12 & 65 & 92 & -1 & 82 \\ 93 & -6 & 0 & 78 & 81 & 8 \end{bmatrix} \longrightarrow \begin{bmatrix} 2 & 12 & 93 \\ 4 & 12 & -6 \\ 18 & 65 & 0 \\ 9 & 92 & 78 \\ 1 & -1 & 81 \\ 2 & 82 & 8 \end{bmatrix}$$



numpy_transpose_2_dim.py

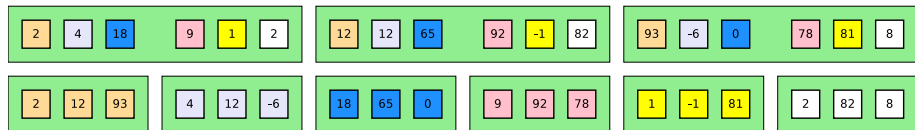
```
1 def main(_args):
2     x = [[2, 4, 18, 9, 1, 2], [12, 12, 65, 92, -1, 82], [93, -6, 0, 78, 81, 8]]
3     x_np = np.array(x, dtype=np.int32)
4     y_np = np.transpose(x_np)
```



- `np.transpose` does *not* change the memory

numpy_transpose_2_dim.py

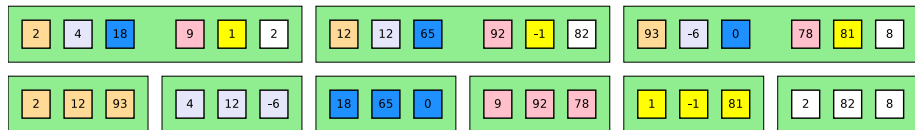
```
1 def main(_args):
2     x = [[2, 4, 18, 9, 1, 2], [12, 12, 65, 92, -1, 82], [93, -6, 0, 78, 81, 8]]
3     x_np = np.array(x, dtype=np.int32)
4     y_np = np.transpose(x_np)
```



- `np.transpose` does *not* change the memory
 - but you should *think* that it does

numpy_transpose_2_dim.py

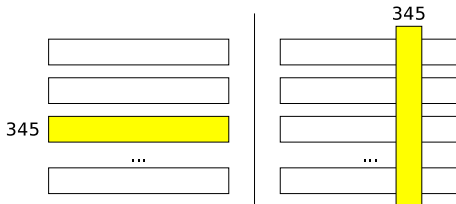
```
1 def main(_args):
2     x = [[2, 4, 18, 9, 1, 2], [12, 12, 65, 92, -1, 82], [93, -6, 0, 78, 81, 8]]
3     x_np = np.array(x, dtype=np.int32)
4     y_np = np.transpose(x_np)
```



- `np.transpose` does *not* change the memory
 - but you should *think* that it does
- `tf.transpose` does $\hat{_}$
 - More on this later, but
 - Please try to remember it, so that you don't get confused

numpy_transpose_and_index.py

```
1 def main(_args):
2     x = np.random.uniform(-1.0, 1.0, [1000, 1000]) # create a random array
3     y = np.transpose(x)
4     print(x[345, :])
5     print(x[:, 345])
```



numpy_simple_maths.py

```
1 def main(_args):
2     x = np.random.uniform(-1.0, 1.0, [1000, 1000]) # create a random array
3
4     # for god's sake, please don't do this!!!
5     for i in range(1000):
6         for j in range(1000):
7             x[i, j] += 1
8
9     # this is the way to go
10    x += 1
```

- numpy has a lot of built-in maths. **Always** use them if possible
 - $x + 10.0$: add 10.0 to all elements of x
 - $x ** 2$: compute $x_{i,j}^2$ for all i, j
 - $1.0 / (x + np.sqrt(2))$: compute $\frac{1}{x_{i,j} + \sqrt{2}}$ for all i, j

numpy_maths_functions.py

```
1 def main(_args):
2     x = np.random.uniform(-1.0, 1.0, [1000, 1000]) # create a random array
3     y = np.zeros_like(x) # create an array with the same size of x, filled with 0
4
5     # for god's sake, please don't do this!!!
6     for i in range(1000):
7         for j in range(1000):
8             y[i, j] = np.exp(x[i, j])
9
10    # this is the way to go
11    y = np.exp(x)
```

- numpy Even the functions
 - `np.exp(x)`: compute $e^{x_{i,j}}$ for all i, j
 - `np.sin(x)`: compute $\sin x_{i,j}$ for all i, j
 - `np.cos(x)`: ...
 - `np.tanh(x)`: ...
 - You can look them up on numpy's homepage

numpy_maths_norm.py

```
1 def main(_args):
2     x = np.random.uniform(-1.0, 1.0, [1000, 1000]) # create a random array
3
4     # for god's sake, please don't do this!!!
5     s = 0
6     for i in range(1000):
7         for j in range(1000):
8             s += x[i, j] ** 2
9
10    # this is the way to go
11    s = np.sum(x ** 2)
```

- numpy and these so-called *reducing operations*
 - `np.sum(x)`: compute the sum of all elements in x
 - `np.min(x)`: compute the minimum of all elements in x
 - `np.max(x)`: ...
 - You can look them up on **numpy's** homepage

- You should **hate** and **avoid** for loop
- You should **hate** and **avoid** while loop
- You should **hate** and **avoid** whatever loops