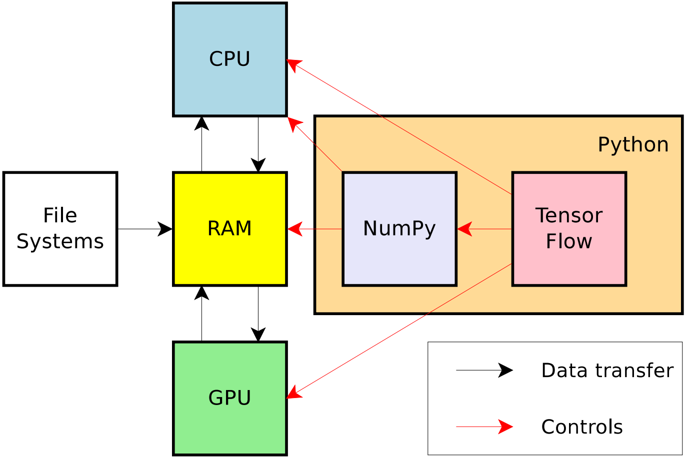
11-695: Competitive Engineering

`Python`, `NumPy` and Introduction to `TensorFlow`

Spring 2018

Roadmap


CPU
Python
File Systems
RAM
NumPy
Tensor Flow
GPU
Data transfer
Controls

Outline

CarnegieMellon

Python: The Hello World program

Carnegie Mellon

hello_world.py

```python
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

def main():
  print("Hello World")

if __name__ == "__main__":
  main()
```

In your Terminal

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

Python: The Hello World program

hello_world.py

```
from __future__ import absolute_import
from __future__ import division
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! They are the future.

Python: The Hello World program

hello_world.py

```python
def main():
  print("Hello World")

if __name__ == "__main__":
  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 this.
  - You're more than welcomed to come up with your standards,

Python: Arguments via TensorFlow

Carnegie Mellon

hello_world_with_args.py

```
import tensorflow as tf

flags = tf.app.flags
FLAGS = flags.FLAGS

flags.DEFINE_string(
  "user_name",                   # argument name
  None,                          # default value
  "We will greet this person"    # help message
)

def main(_args):
  print("Hello, {}!".format(FLAGS.user_name))

if __name__ == "__main__":
  tf.app.run()
```

In your Terminal

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

Python: Arguments via TensorFlow

Carnegie Mellon

hello_world_with_more_args.py

```python
# ...imports and others
flags.DEFINE_string("user_name", None, "We will greet this person")
flags.DEFINE_integer("num_prints", 5, "Number of times to print the message")

def main(_args):
  for i in range(FLAGS.num_prints):
    print("{}. Hello, {}!".format(i, FLAGS.user_name))

if __name__ == "__main__":
  tf.app.run()
```

In your Terminal

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

Python: Basic Types: `int`, `float`, `bool`

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

basic_types.py

```python
def main(_args):
  x = 5                                # integer
  print(type(x))                       # output: <type 'int'>
  print(x + 1, x - 2, x * 3, x ** 4)  # output: 6 3 15 625
  x += 6                               # now x = 11

  y = 2.5                      # float
  print(type(y))               # output: <type 'float'>
  print(x + y, type(x + y))  # output: 13.5 <type 'float'>

  z = 25                         # another integer
  print(z / x, type(z / x))      # float division. output: 2.27273 <type 'float'>
  print(z // x, type(z // x))  # integer division. output: 2 <type 'int'>

  t = True                                      # boolean: True, False
  print(t, type(t))                             # output: True <type 'bool'>
  u = (5 > 3)                                   # Comparisons return True or False
  v = (2 == 7)                                  # There are: >, <, >=, <=, ==, !=
  print(u and v, u or v, not u, u and not v)  # False True False True
```

Python: String

string_examples.py

```python
def main(_args):
  s = "Tensor"     # this is a string
  print(type(s))  # output: <type 'str'>
  print(s[1])     # output: e

  t = 'Flow'      # '...' and "..." are both ok, but do NOT mix them!
  print(len(t))   # output: 4
  w = s + t       # + means concatenation. w is "TensorFlow"
  print(w)        # output: TensorFlow

  m = "{} and {} are {} platforms".format(w, "PyTorch", 2)  # "{}".format(...)
  print(m)  # output: TensorFlow and PyTorch are 2 platforms.

  # A lot of built-in functions. Some examples:
  print(m.upper())              # output: TENSORFLOW AND PYTORCH ARE 2 PLATFORMS
  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!

Python: List

Carnegie Mellon

list_examples.py

```
# ...imports and others
def main(_args):
  a = [1, 2, 23, 2, 1, 27, 21] # this is a list

  print(a, type(a))    # [1, 2, 23, 2, 1, 27, 21] <type 'list'>
  print(a[0], a[3])    # index from 0. output: 1 2
  print(a[-1], a[-2])  # -1 means last element, -2 means next-to-last. output: 21 27

  a[0] = 7         # unlike string, list is mutable
  print(a)         # output: [7, 2, 23, 2, 1, 27, 21]

  a[1] = "hello"              # list can also contain different types
  a[4] = ["tensor", "flow"]   # even another list
  print(a)                    # output: [7, 'hello', 23, 2, ['tensor', 'flow'], 27, 21]

  b = [2.5, 6, 1.7]  #  this is another list
  c = a + b  # just like for strings, + means contatenation for lists
  print(c)   # output: [7, 'hello', 23, 2, ['tensor', 'flow'], 27, 21, 2.5, 6, 1.7]
```

- List are extremely flexible and important in Python.

Python: Dict

Carnegie Mellon

dict_examples.py

```python
# ...imports and others
def main(_args):
  d = {                    # creates a dictionary using {...}
    "hello": "world",      # pairs of key: value, separated by a comma
    1: "TensorFlow",       # key and value can be of any types
    6.0: [2, 2, 3]
  }

  print(d, type(d))
  # output: {1: 'TensorFlow', 'hello': 'world', 6.0: [2, 2, 3]} <type 'dict'>

  print(d["hello"], d[6.0])
  # output: "world" [2, 2, 3]

  d["PyTorch"] = {"author": "Facebook", "version": 2.0}
  # add an element, which is itself a dict

  print(d["PyTorch"])
  # output: {"author": "Facebook", "version": 2.0}
```

Python: Loops and Iterations

CarnegieMellon

loop_examples.py

```python
# ...imports and others
def main(_args):
  for i in range(10):  # basic for loop
    print(i)

  my_list = [3, 21, 4, 3.14, "numpy", 18, 281, "tensorflow"]
  for my_value in my_list:  # loop through a list
    print(my_value)
  for i in range(len(my_list)):  # this also works, but slower!
    print(my_list[i])
  for i, my_value in enumerate(my_list):  # do this if you want the index
    print("Element at {} is {}".format(i, my_value))

  a = [val for val in my_list if isinstance(val, float)]  # all floats
  print(a)  # output: [3.14]
  b = [val for i, val in enumerate(my_list) if i % 2 == 0]  # even-indexed
  print(b)  # output: [3, 4, "numpy", 281]

  # while loop
  x = 10
  while x <= 20:
    print("Now we have x = {}".format(x))
    x += 2
```

Outline

Carnegie Mellon

1. `Python`: a Quick Review

2. `NumPy`: Working with High-Dimensional Data

3. `TensorFlow`: A Computational Framework

NumPy: High-dimensional Arrays

Carnegie Mellon

- `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

```
import numpy as np

def main(_args):
  x = [1, 2, 4, 2, 56, 21, 12, 421]
  print(x)           # output: [1, 2, 4, 2, 56, 21, 12, 421]
  print(type(x))     # output: <type 'list'>

  x_np = np.array(x) # creates a numpy 1-dimensional array
  print(x_np)        # np-looking style: [  1   2   4   2  56  21  12 421]
  print(type(x_np))  # output: <type 'numpy.ndarray'>
  print(x_np.dtype)  # output: int64

  y_np = x_np.astype(np.int32)  # cast the dtype
  print(type(y_np))             # output: <type 'numpy.ndarray'>
  print(y_np.dtype)             # output: int32
```

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

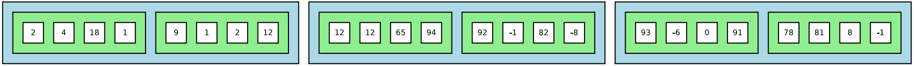
```python
def main(_args):
  x = [[[ 2,  4, 18,  1],
        [ 9,  1,  2, 12]],
       [[12, 12, 65, 94],
        [92, -1, 82, -8]],
       [[93, -6,  0, 91],
        [78, 81,  8, -1]]]
  x_np = np.array(x, dtype=np.int32)

  print(np.ndim(x_np))   # output: 3. It used to be np.rank(x), but was updated.
  print(np.shape(x_np))  # output: (3, 2, 4)
  print(np.size(x_np))   # output: 24
```

NumPy: Memory Layout

numpy_rank_shape_size.py

```python
def main(_args):
  x = [[[ 2,  4, 18,  1],
        [ 9,  1,  2, 12]],
       [[12, 12, 65, 94],
        [92, -1, 82, -8]],
       [[93, -6,  0, 91],
        [78, 81,  8, -1]]]
  x_np = np.array(x, dtype=np.int32)

  print(np.ndim(x_np))   # output: 3. It used to be np.rank(x), but was updated.
  print(np.shape(x_np))  # output: (3, 2, 4)
  print(np.size(x_np))   # output: 24
```

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

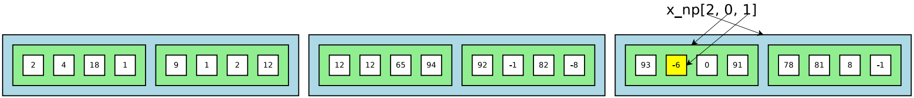


NumPy: Access a Single Element

numpy_memory_demonstration.py

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

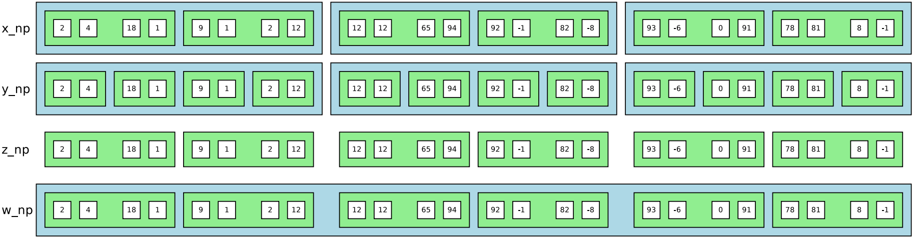
- NumPy arrays are *row major*
- An access to an element happens as follows



NumPy: Reshape and Memory Layout

numpy_reshape.py

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

Carnegie Mellon

numpy_reshape.py

```python
def main(_args):
  # create x_np as before
  print(np.shape(x_np))  # output: (3, 2, 4)

  y_np = np.reshape(x_np, [-1, 4, 2])
  print(np.shape(y_np))  # output: (3, 4, 2)

  z_np = np.reshape(x_np, [8, -1])
  print(np.shape(z_np))  # output: (8, 3)
```

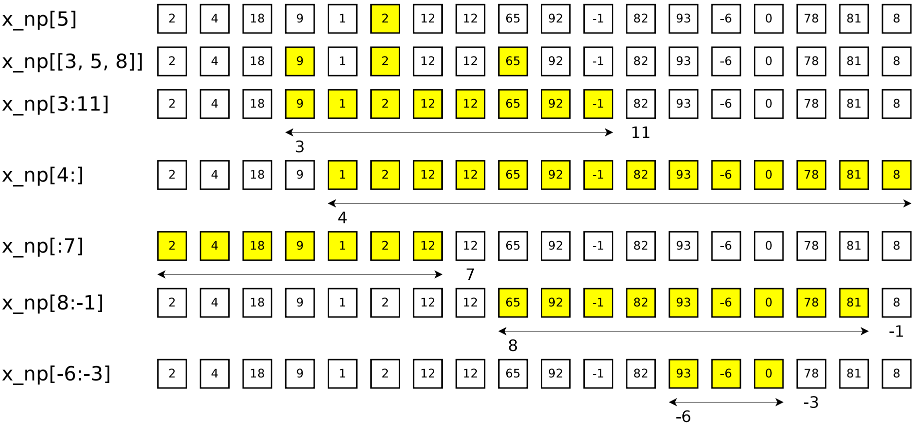
- If you know `x_np` and *all but one* reshaping dimensions
  - then you also know the remaining dimension
- `numpy` allows you not to worry about the misisng dimension
  - By using `-1` at *no more than one* dimension

NumPy: Indexing 1-dim Arrays

CarnegieMellon

numpy_indexing.py

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



NumPy: Indexing 1-dim Arrays

- 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]

`NumPy`: Indexing 1-dim Arrays

- 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`?

NumPy: Indexing 1-dim Arrays

CarnegieMellon

- 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:]`

NumPy: Indexing 1-dim Arrays

- 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

CarnegieMellon

- `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

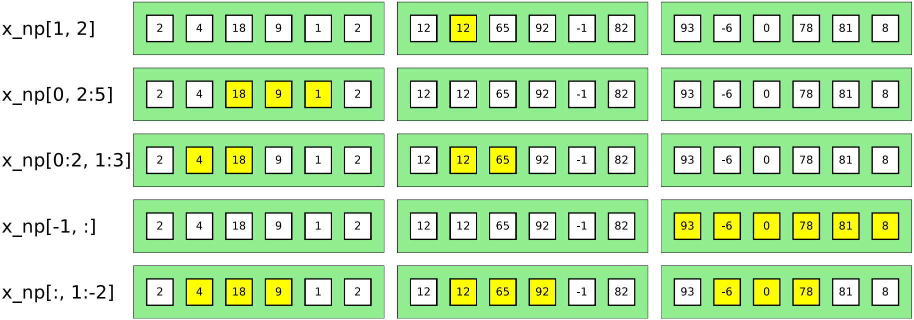
NumPy: Indexing 2-dims Arrays

Carnegie Mellon

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

numpy_indexing_2_dim.py

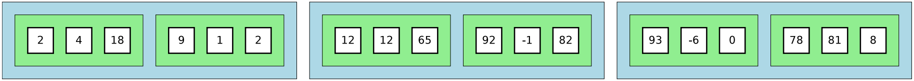
```
def main(_args):
  x = [[2, 4, 18, 9, 1, 2], [12, 12, 65, 92, -1, 82], [93, -6, 0, 78, 81, 8]]
  x_np = np.array(x, dtype=np.int32)
```



NumPy: Final Notes on Indexing

Carnegie Mellon

- 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 and see!
- Now try the following quizz:



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

NumPy: Transpose

Carnegie Mellon

numpy_transpose_2_dim.py

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

numpy_transpose_2_dim.py

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

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

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

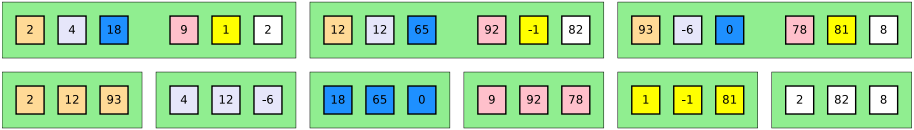
- What happens to the memory of `y_np`?

NumPy: Transpose

numpy_transpose_2_dim.py

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

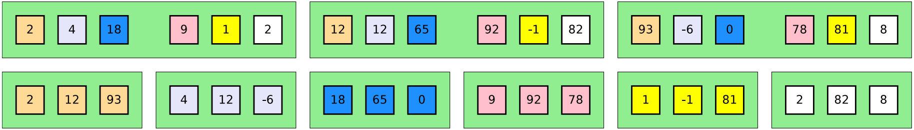
$$\begin{bmatrix} 2 & 4 & 18 & 9 & 1 & 2 \\ 12 & 12 & 65 & 92 & -1 & 82 \\ 93 & -6 & 0 & 78 & 81 & 8 \end{bmatrix} \longrightarrow \begin{bmatrix} 2 & 12 & 93 \\ 4 & 12 & -6 \\ 18 & 65 & 0 \\ 9 & 92 & 78 \\ 1 & -1 & 81 \\ 2 & 82 & 8 \end{bmatrix}$$



NumPy: Transpose

numpy_transpose_2_dim.py

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

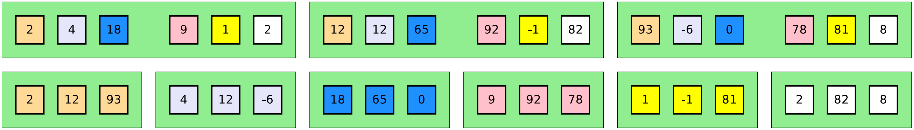


- `np.transpose` does *not* change the memory

NumPy: Transpose

numpy_transpose_2_dim.py

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

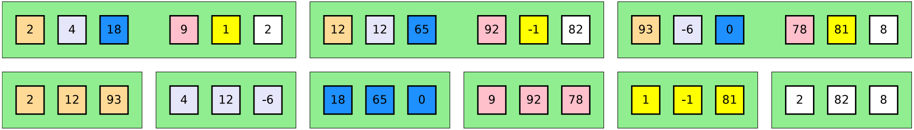


- `np.transpose` does *not* change the memory
  - but you should *think* that it does

NumPy: Transpose

numpy_transpose_2_dim.py

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

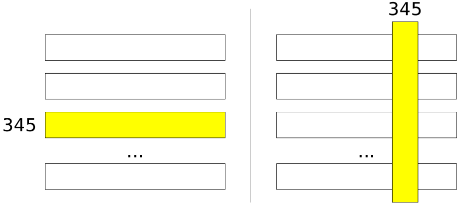


- `np.transpose` does *not* change the memory
  - but you should *think* that it does
- `tf.transpose` does ^_^
  - More on this later, but
  - Please try to remember it, so that you don't get confused

NumPy: Transpose and Indexing

numpy_transpose_and_index.py

```python
def main(_args):
  x = np.random.uniform(-1.0, 1.0, [1000, 1000])  # create a random array
  y = np.transpose(x)
  print(x[345, :])  # fast
  print(x[:, 345])  # slow
```



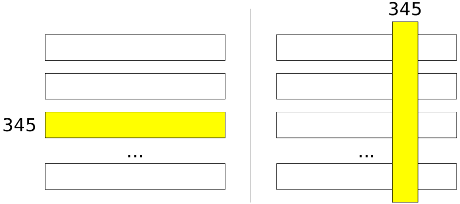
- How about `y[345, :]` and `y[:, 345]`?

NumPy: Transpose and Indexing

CarnegieMellon

numpy_transpose_and_index.py

```python
def main(_args):
  x = np.random.uniform(-1.0, 1.0, [1000, 1000])  # create a random array
  y = np.transpose(x)
  print(x[345, :])  # fast
  print(x[:, 345])  # slow
```

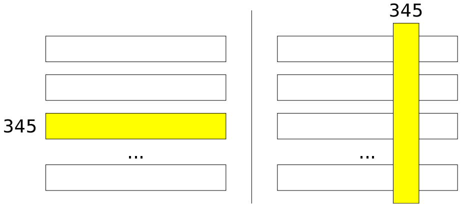


- How about `y[345, :]` and `y[:, 345]`?
- What if we transpose again at some points?

NumPy: Transpose and Indexing

numpy_transpose_and_index.py

```
def main(_args):
  x = np.random.uniform(-1.0, 1.0, [1000, 1000])  # create a random array
  y = np.transpose(x)
  print(x[345, :])  # fast
  print(x[:, 345])  # slow
```



- How about `y[345, :]` and `y[:, 345]`?
- What if we transpose again at some points?
  - Don't hurt yourself
  - Don't try to index after you tranpose in `numpy`

NumPy: Maths on Arrays

Carnegie Mellon

numpy_simple_maths.py

```
def main(_args):
  x = np.random.uniform(-1.0, 1.0, [1000, 1000])  # create a random array

  # for god's sake, please don't do this!!!
  for i in range(1000):
    for j in range(1000):
      x[i, j] += 1

  # this is the way to go
  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 on Arrays

Carnegie Mellon

numpy_maths_functions.py

```python
def main(_args):
  x = np.random.uniform(-1.0, 1.0, [1000, 1000])  # create a random array
  y = np.zeros_like(x)  # create an array with the same size of x, fille with 0

  # for god's sake, please don't do this!!!
  for i in range(1000):
    for j in range(1000):
      y[i, j] = np.exp(x[i, j])

  # this is the way to go
  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 on Arrays

Carnegie Mellon

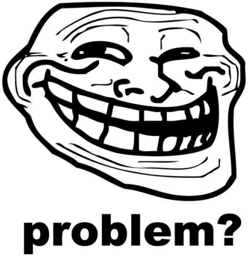
numpy_maths_norm.py

```python
def main(_args):
  x = np.random.uniform(-1.0, 1.0, [1000, 1000])  # create a random array

  # for god's sake, please don't do this!!!
  s = 0
  for i in range(1000):
    for j in range(1000):
      s += x[i, j] ** 2

  # this is the way to go
  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

`NumPy`: Summary of Maths on Arrays

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



Outline

TensorFlow: Getting Started

Carnegie Mellon

- Install: https://www.tensorflow.org/install/
- Usage:

tf_basic_program.py

```
import tensorflow as tf

def main(_args):
  # your programs

if __name__ == "__main__":
  tf.app.run()
```

Structure of a `tf` program

Carnegie Mellon

- A program in *tf* always consists of:
  - Building a *computational graph*
  - Execute the relevant parts in the built graph

tf_basic_program.py

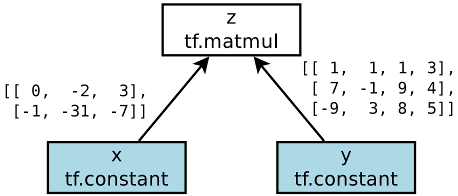
```python
import tensorflow as tf

def main(_args):
  g = tf.Graph()         # create a computational graph
  with g.as_default():  # everything you do with TF happens in the graph g
    build_tf_graph()    # define the operations in g

    with tf.Session() as sess:                      # TF boiler-plate code
      sess.run(tf.global_variables_initializer())  # TF boiler-plate code

      # execute the TF graph, e.g.:
      sess.run([train_op, compute_loss])

if __name__ == "__main__":
  tf.app.run()
```

Computational Graph

Carnegie Mellon

tf_graph_demonstration_1.py

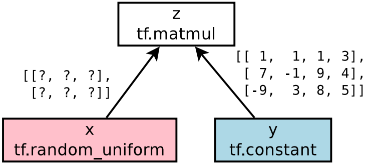
```python
import tensorflow as tf

def build_tf_graph():
  x = tf.constant([[0, -2, 3], [-1, -31, -7]], dtype=tf.int32)
  y = tf.constant([[1, 1, 1, 3], [7, -1, 9, 4], [-9, 3, 8, 5]], dtype=tf.int32)
  z = tf.matmul(x, y)
  return x, y, z

def main(_args):
  # other code...
  build_tf_graph()
  with tf.Session() as sess:
    output = sess.run([z])  # execute the operation z
    print(output)
```



Another Computational Graph

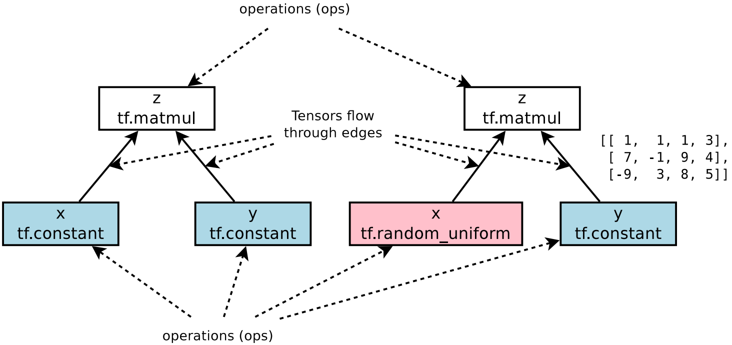
tf_graph_demonstration_2.py

```python
import tensorflow as tf

def build_tf_graph():
  x = tf.random_uniform([2, 3], minval=-5, maxval=5, dtype=tf.int32)
  y = tf.constant([[1, 1, 1, 3], [7, -1, 9, 4], [-9, 3, 8, 5]], dtype=tf.int32)
  z = tf.matmul(x, y)
  return x, y, z

def main(_args):
  # other code...
  x, y, z = build_tf_graph()
  with tf.Session() as sess:
    output = sess.run([z])  # execute the operation z
    print(output)
```



Computational Graph

Carnegie Mellon

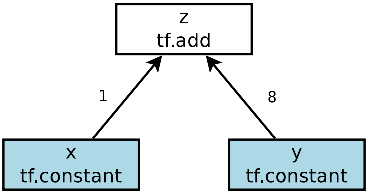


- Formally speaking
  - `tf` computational graph is a *directed acycic graph* (DAG)
  - Nodes are called *operations*, or *ops*
  - Ops produce *tensors*
  - Tensors flow around through edges

Yet Another Computational Graph

tf_execution_order.py

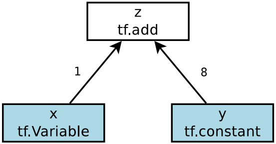
```python
import tensorflow as tf

def build_tf_graph():
  x = tf.constant(1, dtype=tf.int32)
  y = tf.constant(8, dtype=tf.int32)
  z = x + y
  return x, y, z

def main(_args):
  # other code...
  x, y, z = build_tf_graph()
  with tf.Session() as sess:
    output = sess.run([x, y, z])  # execute all 3 operations
    print(output)  # output: [1, 8, 9]
```



What about now?

tf_execution_order_2.py

```python
def build_tf_graph():
  x = tf.Variable(1, dtype=tf.int32, name="x")
  y = tf.constant(8, dtype=tf.int32)
  z = x + y
  return x, y, z

def main(_args):
  # other code...
  with tf.Session() as sess:
    output = sess.run([x, y, z])  # execute all 3 operations
    print(output)                 # output: [1, 8, 9]
```



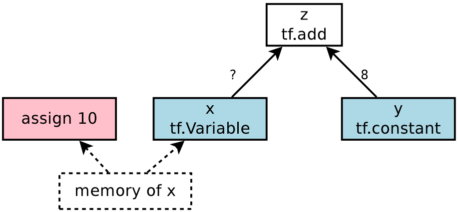
- Unlike `tf.constant`, `tf.Variable` can be changed

What about now?

Carnegie Mellon

tf_execution_order_3.py

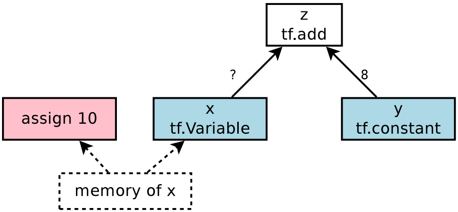
```python
def build_tf_graph():
  x = tf.Variable(1, dtype=tf.int32, name="x")
  assign_x = tf.assign(x, 10)
  y = tf.constant(8, dtype=tf.int32)
  z = x + y
  return x, y, z, assign_x

def main(_args):
  # other code...
  with tf.Session() as sess:
    output = sess.run([z, assign_x])  # execute all 3 operations
    print(output)                     # output: ?
```



What about now?

tf_execution_order_3.py

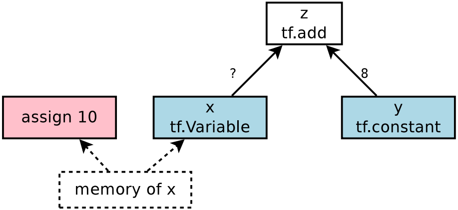
```python
def build_tf_graph():
  x = tf.Variable(1, dtype=tf.int32, name="x")
  assign_x = tf.assign(x, 10)
  y = tf.constant(8, dtype=tf.int32)
  z = x + y
  return x, y, z, assign_x

def main(_args):
  # other code...
  with tf.Session() as sess:
    output = sess.run([z, assign_x])  # execute all 3 operations
    print(output)                     # output: ?
```



We don't know!

What about now?

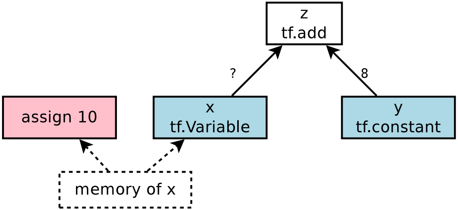
Carnegie Mellon



- Execution order follows the computational graph's topological order.

What about now?

Carnegie Mellon



- Execution order follows the computational graph's topological order.
- and nothing else!