18-661: Introduction to ML for Engineers

Python

Spring 2022

Tianshu Huang, ECE – Carnegie Mellon University

Why Python?

Python is not:

- The first mover: **R** is an implementation of S created circa 1976, while python was released 1991. R is still very popular in statistics.

Why Python?

Python is not:

- The first mover: **R** is an implementation of S created circa 1976, while python was released 1991. R is still very popular in statistics.
- The ideal implementation: **Julia** is currently being developed, and is essentially Python but better in every way technically.

Why Python?

Python is not:

- The first mover: **R** is an implementation of S created circa 1976, while python was released 1991. R is still very popular in statistics.
- The ideal implementation: **Julia** is currently being developed, and is essentially Python but better in every way technically.
- The most popular: compared to other dynamically typed scripting languages, **JavaScript** is significantly more popular (67.7%) than Python (44.1%) according to the 2021 StackOverflow developer survey.

Why Python?

Python:

- is fast to develop and very readable
- is dynamically typed and easy to debug
- can be extended using other languages (usually C/C++)
- is reasonably performant (when using libraries)
- can run in development as well as deployed environments

Most importantly, Python has strong community support and extensive tooling for Machine Learning ...

Why Python?

Python:

- is fast to develop and very readable
- is dynamically typed and easy to debug
- can be extended using other languages (usually C/C++)
- is reasonably performant (when using libraries)
- can run in development as well as deployed environments

Most importantly, Python has strong community support and extensive tooling for Machine Learning ...

Python is *the* standard for Machine Learning.

Outline

Disclaimer: while this lecture will be helpful when working on the homework, it will not be directly included on your homework or exams.

1. Python Review: Basics and Tools

2. Better Python: Writing "Pythonic" Code

3. Faster Python: Understanding Python Performance

A jupyter notebook including code used to generate examples shown in the slides can be found here (download or choose "Open with Google Colaboratory").

Python Review: Basics and Tools

Data Types: Primitives

Numerical Primitives:

- float: standard IEEE 64-bit floating point number
- complex: 64-bit real and imaginary components
- int: bignum, with starting size 32 bits (usually); cannot overflow

Array-like Primitives:

- str: fixed-size char array
- bytes: fixed-size byte array (immutable)

```
type(1.)
```

```
float
```

```
type(1j)
```

```
complex
```

```
x = 10**20 + 1
print(x)
print(2**64)
type(x)
```

```
100000000000000000001
18446744073709551616

int
```

```
type("1")
```

```
str
```

```
type(b"1")
```

```
bytes
```

Data Types: Booleans

Special Primitives: None, True, False

- These "special objects" are reused: there is only one "True" and "False" object in all of cPython, and all booleans are just pointers to these canonical boolean objects.
- We can check this using the "is" keyword, which tells you whether two variables point to the same object in memory.

```
x = 123456
y = 123456
print(x is y)
```

```
False
```

```
x = True
y = True
x is y
```

```
True
```

Data Types: Booleans

Special Primitives: None, True, False

- These "special objects" are reused: there is only one "True" and "False" object in all of cPython, and all booleans are just pointers to these canonical boolean objects.
- We can check this using the "is" keyword, which tells you whether two variables point to the same object in memory.

Note: this behavior also applies to some other python objects such as small integers.

```python
x = 123456
y = 123456
print(x is y)
```

```
False
```

```python
x = True
y = True
x is y
```

```
True
```

```python
x = 1
y = 1
x is y
```

```
True
```

```python
x = 1.0
y = 1.0
x is y
```

```
False
```

Data Types: Arrays

Array-like:

- list: exponentially over-allocated arrays with growth factor $\frac{9}{8}n + 6$, rounded down to the nearest 8. Elements can be any type.
- bytearray: resizable and mutable bytes
- tuple: fixed-length, immutable array

```
arr = (1, 2, 3)
# Do:
x, y, z = arr
arr = (x, 1, z)
# Don't:
arr[1] = 0
```

```
TypeError: 'tuple' object does not support item assignment
```

```
arr = [1, 1.0, "1"]
print(arr)
```

```
[1, 1.0, '1']
```

```
arr[2] = "a"
print(arr[2], arr[-1])
```

```
a a
```

```
arr.append("b")
print(arr)
```

```
[1, 1.0, 'a', 'b']
```

```
print(len(arr), arr)
arr = arr + ['c', 2.0]
print(len(arr), arr)
```

```
4 [1, 1.0, 'a', 'b']
6 [1, 1.0, 'a', 'b', 'c', 2.0]
```

Array Slicing

Array-like data types as well as array-like primitives can be “sliced” using

`array[start:end:stride]`

where start defaults to 0, end defaults to -1, and stride defaults to 1.

```
arr = list(range(10))
print(arr)
print(arr[:2], arr[4:6], arr[8:])
print(arr[::2], arr[1:7:2])
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1] [4, 5] [8, 9]
[0, 2, 4, 6, 8] [1, 3, 5]
```

```
s = "abcdefghijklmnopqrstuvwxyz"
s[::2]
```

```
'acegikmoqsuwy'
```

Data Types: Maps

Map-like:

- dict: standard hash table.
- set: a key-only dict for checking membership

Keys can have any immutable data type (numerical, string, bytes, tuple).

```
states = {
    "Pittsburgh": "PA",
    "Austin": "TX",
    "San Francisco": "CA"
}
print(states["Pittsburgh"], states.get("Pittsburgh"))
print("Miami" in states, states.get("Miami"))
```

```
PA PA
False None
```

```
states["Miami"] = "FL"
print(states)
```

```
{'Pittsburgh': 'PA', 'Austin': 'TX', 'San Francisco': 'CA', 'Miami': 'FL'}
```

Warning: Elements are References

Python data structures only contain references to their elements!

```
inner = [1, 2, 3]
outer = [inner, [0], [0]]
nested = {"key": outer}
print(inner)
nested["key"][0][0] = -1
print(inner)
```

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

```
inner = [1, 2, 3]
outer = [inner, [0], [0]]
nested = {"key": outer.copy()}
print(inner)
nested["key"][0][0] = -1
print(inner)
```

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

```
inner = [1, 2, 3]
outer = [inner.copy(), [0], [0]]
nested = {"key": outer.copy()}
print(inner)
nested["key"][0][0] = -1
print(inner)
```

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

Iterators

Loops in python do not use C-like

```
for(init, condition, update) {}
```

loops. Instead, loops use *iterators* that encapsulate this logic:

- for i in range(start, stop, step)
- for element in example_list
- for character in example_string
- for key in example_dict
- for key, value in example_dict.items()

```python
# range(end): 0...end
print(list(range(7)))
# range(start, end): start...end
print(list(range(2, 7)))
# range(start, end, skip):
print(list(range(2, 7, 2)))
```

```
[0, 1, 2, 3, 4, 5, 6]
[2, 3, 4, 5, 6]
[2, 4, 6]
```

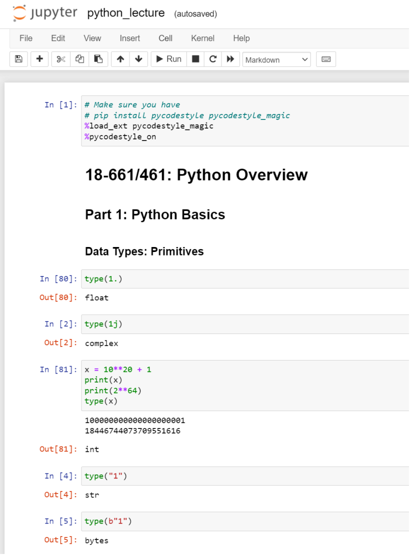
```python
example_list = [1, 2, "a", "b"]
for element in example_list:
    print(element)
```

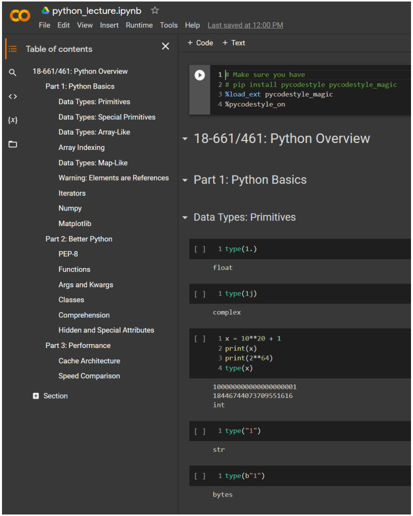
```
1
2
a
b
```

```python
example_dict = {"a": 1, "b": 2}
for key in example_dict:
    print(key)
for key, value in example_dict.items():
    print(key, ":", value)
```

```
a
b
a : 1
b : 2
```

Tools: Jupyter Notebook, Google Colab





Tools: Libraries

The "Scipy Stack":

- **numpy**: vectors, matrices, arrays, and operations involving them
- **pandas**: dealing with tabular data, usually CSVs
- **matplotlib**: plotting; originally started as a matlab clone
- and others!

Tools: Libraries

The "Scipy Stack":

- **numpy**: vectors, matrices, arrays, and operations involving them
- **pandas**: dealing with tabular data, usually CSVs
- **matplotlib**: plotting; originally started as a matlab clone
- and others!

Machine Learning Libraries:

- **sklearn**: numpy-based classical machine learning methods
- **pytorch** (also tensorflow, JAX): will be covered in week 11
- **statsmodels**: statistical models (equivalent of many R packages); not used in this course

Numpy

Numpy adds Numpy Arrays, a new primitive which contains an n-dimensional arrays where each element has the same data type. These data types correspond to primitives supported by hardware.

```
import numpy as np

src = [[1., 2., 3.], [4., 5., 6.]]
X1 = np.array(src)
print(X1, "(Rows, Columns):", x1.shape, "Type:", x1.dtype)
```

```
[[1. 2. 3.]
 [4. 5. 6.]] (Rows, Columns): (2, 3) Type: float64
```

```
X2 = np.array([[2., 2., 2.], [1., 1., 1.,]])
print(X1 * 2 + 1)
print(X1 * X2)
```

```
[[ 3.  5.  7.]
 [ 9. 11. 13.]]
[[2. 4. 6.]
 [4. 5. 6.]]
```

```
vec = np.array([1., 2., 3., 4., 5., 6.], dtype=np.float16)
print(vec, "Length:", vec.shape, "Type:", vec.dtype)
```

```
[1. 2. 3. 4. 5. 6.] Length: (6,) Type: float16
```

Numpy

Common operations:

np.random.random	Random array with elements Unif(0, 1)
np.random.unif	Random array from uniform distribution
np.eye	Identity matrix
np.zeros	Zero matrix
np.ones	Ones matrix
np.arange	Equivalent to np.array(range(.))
np.linspace	Linear space; evenly spaced points
np.load	Load array from file
np.save	Save single array to file
np.savez	Save multiple arrays (keyword args)
np.dot	Dot product
np.matmul	Matrix multiplication

Please refer to the Numpy Documentation.

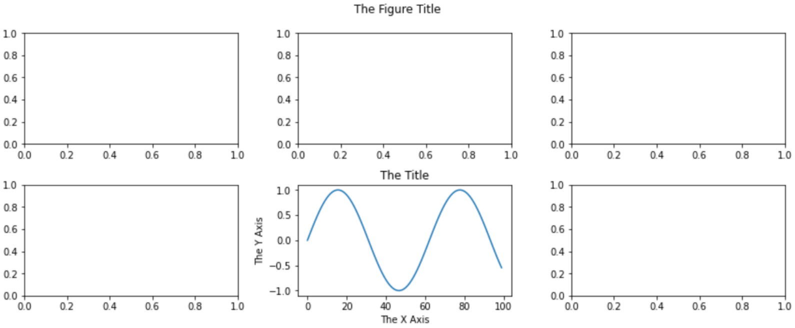
Matplotlib

```
from matplotlib import pyplot as plt

fig, axs = plt.subplots(2, 3, figsize=(12, 5))

x = np.linspace(0, 10, 100)
y = np.sin(x)
axs[1][1].plot(y)
axs[1][1].set_xlabel("The X Axis")
axs[1][1].set_ylabel("The Y Axis")
axs[1][1].set_title("The Title")

fig.suptitle("The Figure Title")
fig.tight_layout()
```



Better Python: Writing "Pythonic" Code

PEP-8

Install a Linter.

- VSCode: the official Python extension includes everything.
- Jupyter Notebook: make sure you have pycodestyle and pycodestyle_magic, then:

  ```
  %load_ext pycodestyle_magic
  %pycodestyle_on
  ```

As a bonus, also enable pydocstyle!

```
1  arr = [1,2, 3]
2  
3  try:
4      arr[0] = arr[0] / 0
5  except:
6      pass
7  
8  very_long_array = [123456789, 123456789,
9          123456789, 123456789, 123456789]
10 other_long_array = [
11     123456789, 123456789,
12         123456789, 123456789, 123456789]
13 
14 
15 
16 bad_tuple = ( 1, 2)
```

```
1:9: E231 missing whitespace after ','
5:1: E722 do not use bare 'except'
7:1: W293 blank line contains whitespace
9:9: E128 continuation line under-indented for visual indent
12:9: E131 continuation line unaligned for hanging indent
16:1: E303 too many blank lines (3)
16:14: E201 whitespace after '('
```

Functions

Functions can take positional and keyword arguments.

- Extra positional arguments are used to fill keyword arguments in order
- Keyword arguments that are not provided use default values.

Functions (as well as methods and objects) can include a "docstring."

```
def example_function(pos, scale=2.0):
    """This is an example function.

    The docstring here should describe what the function does.

    Parameters
    ----------
    pos : float
       Providing the type is useful since it is difficult to
       determine the expected type just by looking at code.

    Keyword Args
    ------------
    scale : float
        Another argument.

    Returns
    -------
    float
        A description of what the return value does.
    """
    # Extremely sophisticated math goes here
    return pos * scale


print(example_function(1.5, scale=1.5))
print(example_function(1.5, 1.5))
print(example_function(1.5))

help(example_function)
```

```
2.25
2.25
3.0
Help on function example_function in module __main__:

example_function(pos, scale=2.0)
    This is an example function.

    The docstring here should describe what the function does.
```

*args and **kwargs

Instead of having to write out every single argument, we can pass them programmatically!

- *args: collapses a list of positional args.
- **kwargs: collapses a dict of keyword args.

Example: argument passthrough

```
def div_zero_is_inf(func, *args, **kwargs):
    """If a function tries to divide by zero, returns -inf."""
    try:
        func(*args, **kwargs)
    except ZeroDivisionError:
        return float('inf')


def test_func(x, y, eps=0.0):
    return x / (y + eps)


div_zero_is_inf(test_func, 1, 0, 0.0)
```

```
inf
```

Classes

```python
class BaseClass:
    """Initializer docstring goes here.

    Parameters
    ----------
    name : str
        Name of this object.
    """
    def __init__(self, name):
        self.name = name


class SubClass(BaseClass):
    """Example subclass."""

    example_attribute = []

    def append(self, elem):
        self.example_attribute.append(elem)
        print(self.example_attribute)
```

Classes

```python
class BaseClass:
    """Initializer docstring goes here.

    Parameters
    ----------
    name : str
        Name of this object.
    """
    def __init__(self, name):
        self.name = name


class SubClass(BaseClass):
    """Example subclass."""

    example_attribute = []

    def append(self, elem):
        self.example_attribute.append(elem)
        print(self.example_attribute)
```

```python
obj1 = SubClass("Example1")
obj2 = SubClass("Example2")
obj1.append(1)
obj2.append(2)
```

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

Classes

```python
class BaseClass:
    """Initializer docstring goes here.

    Parameters
    ----------
    name : str
        Name of this object.
    """
    def __init__(self, name):
        self.name = name


class SubClass(BaseClass):
    """Example subclass."""

    example_attribute = []

    def append(self, elem):
        self.example_attribute.append(elem)
        print(self.example_attribute)
```

```python
obj1 = SubClass("Example1")
obj2 = SubClass("Example2")
obj1.append(1)
obj2.append(2)
```

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

```python
class SubClass(BaseClass):
    """Example subclass."""

    def __init__(self, name):
        super().__init__(name)
        self.example_attribute = []

    def append(self, elem):
        self.example_attribute.append(elem)
        print(self.example_attribute)
```

```python
obj1 = SubClass("Example1")
obj2 = SubClass("Example2")
obj1.append(1)
obj2.append(2)
```

```
[1]
[2]
```

Hidden Attributes

```
class ExampleClass:
    """Example Attributes."""

    def _private_method(self, x):
        return x**2

    def __actually_private_method(self, x):
        return x**2

    def exposed_private_method(self, x):
        return self.__actually_private_method(x + 1)


example = ExampleClass()
```

```
example._private_method(5)
```

```
25
```

```
example.__actually_private_method(5)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
C:\Users\TIANSH~1\AppData\Local\Temp/ipykernel_11692/4284769622.py in <module>
----> 1 example.__actually_private_method(5)

AttributeError: 'ExampleClass' object has no attribute '__actually_private_method'
```

```
example.exposed_private_method(5)
```

```
36
```

Comprehension

- List:

```
list_comprehension = [
    x for x in iterator
    if condition
]
```

- Dict:

```
dict_comprehension = {
    k: v
    for k, v in
    iterator
}
```

```python
arr_in = [1, 2, 3, 4, 5]

# Don't:
arr_out = []
for x in arr_in:
    arr_out.append(x**2 + 5 * x + 13)
print(arr_out)

# Do:
arr_out = [x**2 + 5 * x + 13 for x in arr_in]
print(arr_out)
```

```
[19, 27, 37, 49, 63]
[19, 27, 37, 49, 63]
```

```python
arr_out = [
    x**2 + 5 * x + 13
    for x in arr_in if x % 2 == 0]
print(arr_out)
```

```
[27, 49]
```

```python
dict_out = {
    x: x**2 + 5 * x + 13 for x in arr_in}
print(dict_out)
```

```
{1: 19, 2: 27, 3: 37, 4: 49, 5: 63}
```

Modules: Files

Python files can be imported as modules by scripts (or jupyter notebooks!) which have that file in the same directory.

```python
"""Modules should include a module-level docstring."""

import numpy as np


CONSTANT_ATTRIBUTE = "Constants are usually ALL_CAPS"

# This code will be run when the file is imported.
print("Imported example_file")


def __private_func():
    """Function is private to this file."""
    print("This function is private!")


def example_func(x):
    """Publicly accessible at the .example_func attribute."""
    print("The input was:", x)
    __private_func()
```

```python
import example_file

print(example_file.example_func)
```

```
<function example_func at 0x00000155B1A88040>
```

```python
from example_file import example_func

print(example_func)
```

```
<function example_func at 0x00000155B1A88040>
```

Modules: Directories

Directories can be also made into modules containing other files by including a "__init__.py" file.

```python
"""Modules can have docstrings too."""

# This allows us to do
#   from example_module import example_func
# or
#   import example_module
#   ...
#   example_module.example_func(...)
# Instead of
#   example_module.example_file.example_func
from example_file import example_func
```

General guidelines:

- If scrolling up and down gets annoying, you should probably split the file up.
- Modules can go multiple levels down (useful for larger projects).
- If using Juypter Notebook, put "core" code into a module, and import this into the notebook to run your experiments.

Faster Python: Understanding Python Performance

CPython

Everything in CPython is a PyObject, which contains a reference count, type, and contents.

```
/** Definition has been simplified for clarity. */
typedef struct {
    /** Reference count */
    Py_ssize_t ob_refcnt;
    /** Object type */
    struct _typeobject *ob_type;
    /** Actual data extends below */
    void *buf;
} PyObject;
```

Source: https://svn.python.org/projects/python/trunk/Include/object.h

Reference Counting

```c
// Note: these definitions are actually macros
// Implementations have been simplified for clarity
void Py_INCREF(PyObject *op) {
    op->ob_refcnt++;
}

void Py_DECREF(PyObject *op) {
    if (--op->ob_refcnt != 0) {
        // Raise error if ob_refcnt < 0
    } else {
        _Py_Dealloc(op);
    }
}
```

Interpreter Overhead

```
// Pseudocode for c = a + b.
// Assume a has type int
allocate c
c.refcount = 1
if b.type == int:
    c.type = int
    c.value = a.value + b.value
else:
    c.type = float
    a_float = (float) a.value
    c.value = a_float + b.value
a.refcount -= 1
b.refcount -= 1
return c
```

Assume that *a* and *b* are provided dynamically (cannot determine types while parsing).

Interpreter Overhead

```
// Pseudocode for c = a + b.
// Assume a has type int
allocate c
c.refcount = 1
if b.type == int:
    c.type = int
    c.value = a.value + b.value
else:
    c.type = float
    a_float = (float) a.value
    c.value = a_float + b.value
a.refcount -= 1
b.refcount -= 1
return c
```

Assume that *a* and *b* are provided dynamically (cannot determine types while parsing).

Rough cost, not including intermediate pointers:

- 3-4 arithmetic operations: add, possibly cast, update refcount
- 6 values read: refcount, type, and value for a and b
- 5 values written: refcount for a, b, and c, value and type of c

Global Interpreter Lock (GIL)

Because reference counting is not atomic, CPython has a global interpreter lock which can only be released by Python C API functions that promise not to modify any reference counts.

Global Interpreter Lock (GIL)

Because reference counting is not atomic, CPython has a global interpreter lock which can only be released by Python C API functions that promise not to modify any reference counts.

- Since reference counts are manipulated so often (accounts for 5-10% of execution time!), the GIL is held by default and must be explicitly released by Python C API functions. Source: https://www.caichinger.com/blog/2015/05/23/python_atomic_refcounting_slowdown/

Global Interpreter Lock (GIL)

Because reference counting is not atomic, CPython has a global interpreter lock which can only be released by Python C API functions that promise not to modify any reference counts.

- Since reference counts are manipulated so often (accounts for 5-10% of execution time!), the GIL is held by default and must be explicitly released by Python C API functions. Source: https://www.caichinger.com/blog/2015/05/23/python_atomic_refcounting_slowdown/
- It's not worth it to release the GIL for small scalar operations.

Global Interpreter Lock (GIL)

Because reference counting is not atomic, CPython has a global interpreter lock which can only be released by Python C API functions that promise not to modify any reference counts.

- Since reference counts are manipulated so often (accounts for 5-10% of execution time!), the GIL is held by default and must be explicitly released by Python C API functions. Source: https://www.caichinger.com/blog/2015/05/23/python_atomic_refcounting_slowdown/
- It's not worth it to release the GIL for small scalar operations.

This forces code seeking to leverage multicore computers to create processes (multiprocessing) instead of threads (multithreading)!

Memory Access

Many common operations such as matrix/vector multiplication are dominated by memory access time.

- Memory is "opened" one row at a time (512B - 2KB)
- Memory is loaded into cache one "line" at a time (64 bytes)

If we access memory randomly instead of sequentially, we have to throw away a lot of work!

Vectorization: Numpy

What is a numpy array?

- ndarray.dtype: data type, i.e. float32, int64
- ndarray.data: data buffer; usually row-major, i.e.
  `data[i, j] = *(data + i * num_columns + j);`
- ndarray.shape: array dimensions
- ndarray.strides: how many elements to skip to get to the next:
  `data[i] = *(data + i * stride);`

Operations on numpy arrays are **vectorized**: operations are applied to the whole array instead of individual elements.

Why Vectorize?

Multithreading:

- Parallelize without paying process overhead
- Large vectors spend more time in the C API function, which makes releasing the GIL worth it

Why Vectorize?

Multithreading:

- Parallelize without paying process overhead
- Large vectors spend more time in the C API function, which makes releasing the GIL worth it

Python interpreter:

- Reduce interpreter overhead: the entire vector shares the same PyObject, so memory management and type checking only has to happen once

Why Vectorize?

Multithreading:

- Parallelize without paying process overhead
- Large vectors spend more time in the C API function, which makes releasing the GIL worth it

Python interpreter:

- Reduce interpreter overhead: the entire vector shares the same PyObject, so memory management and type checking only has to happen once

Architecture:

- Reduce instruction decode overhead with SIMD vector instructions (AVX512)
- Make caching more efficient: vectors usually* occupy continuous memory

Vectorization Example

```
x = np.random.uniform(size=65536)
y = np.random.randint(low=0, high=len(x), size=4096)
print(x)
print(y)
```

```
[0.3058059  0.86948619 0.43254264 ... 0.60967155 0.27625591 0.26483979]
[54151 61653  6018 ...  9395 23801 15794]
```

Vectorization Example

```
x = np.random.uniform(size=65536)
y = np.random.randint(low=0, high=len(x), size=4096)
print(x)
print(y)
```

```
[0.3058059  0.86948619 0.43254264 ... 0.60967155 0.27625591 0.26483979]
[54151 61653  6018 ...  9395 23801 15794]
```

```
def naive_approach():
    arr = []
    for idx in y:
        arr.append(x[idx])
    return sum(arr)

timeit(naive_approach, number=1000)
```

```
0.9709060000000136
```

Vectorization Example

```
x = np.random.uniform(size=65536)
y = np.random.randint(low=0, high=len(x), size=4096)
print(x)
print(y)
```

```
[0.3058059  0.86948619 0.43254264 ... 0.60967155 0.27625591 0.26483979]
[54151 61653  6018 ...  9395 23801 15794]
```

```
def naive_approach():
    arr = []
    for idx in y:
        arr.append(x[idx])
    return sum(arr)

timeit(naive_approach, number=1000)
```

```
0.9709060000000136
```

```
def better_approach():
    return sum(x[idx] for idx in y)

timeit(better_approach, number=1000)
```

```
0.7448531000000003
```

Vectorization Example

```
x = np.random.uniform(size=65536)
y = np.random.randint(low=0, high=len(x), size=4096)
print(x)
print(y)
```

```
[0.3058059  0.86948619 0.43254264 ... 0.60967155 0.27625591 0.26483979]
[54151 61653  6018 ...  9395 23801 15794]
```

```
def naive_approach():
    arr = []
    for idx in y:
        arr.append(x[idx])
    return sum(arr)

timeit(naive_approach, number=1000)
```

```
0.9709060000000136
```

```
def better_approach():
    return sum(x[idx] for idx in y)

timeit(better_approach, number=1000)
```

```
0.7448531000000003
```

```
def best_approach():
    return np.sum(x[y])

timeit(best_approach, number=1000)
```

```
0.025412899999992078
```

*: Cache Architecture

Physical caches do not operate like software least recently used (LRU) caches; they are **set associative**.

- Searching for matches across an entire cache is very expensive, so instead each memory address is restricted to a “set” of 8-16 (i.e. Zen 2 L1: 8 lines) cache locations that it can occupy.

*: Cache Architecture

Physical caches do not operate like software least recently used (LRU) caches; they are **set associative**.

- Searching for matches across an entire cache is very expensive, so instead each memory address is restricted to a "set" of 8-16 (i.e. Zen 2 L1: 8 lines) cache locations that it can occupy.
- If the cache size is 64KB (i.e. Zen 2 L1), this means every

$$\frac{\text{64KB cache}}{\text{8 way}} = \text{8192 bytes} = \text{2048 floats}$$

  will correspond to the same set.

If we read just a few bytes over and over again from different lines in the same set, we will "clobber" the cache!

*: Cache Architecture

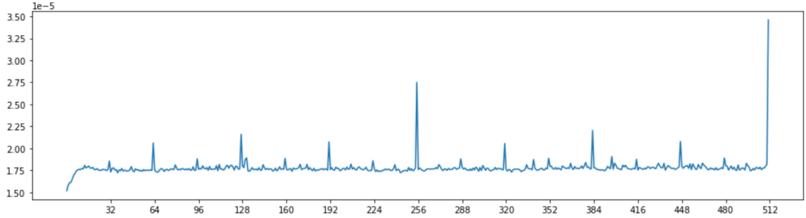
Numpy strided array access is not continuous memory.

```
def test(stride, n=1000):
    x = np.random.uniform(size=(2048, stride))
    return timeit(lambda: np.var(x[:, 0]), number=n) / n
```

```
res = [test(s, n=10000) for s in tqdm(np.arange(512) + 1)]
```

```
100%|██████████████████████████████████████████████████| 512/512 [01:32<00:00,  5.51it/s]
```

```
fig, axs = plt.subplots(1, 1, figsize=(16, 4))
axs.plot(res)
_ = axs.set_xticks(np.arange(0, 512, 32) + 32)
```



Example

```
x = np.random.uniform(size=(2048, 2048))
K = 100000
```

```
timeit(lambda: np.var(x[0]), number=K) / K * 10**6
```

```
15.442851000000248
```

```
timeit(lambda: np.var(x[:, 0]), number=K) / K * 10**6
```

```
37.20540700000129
```

```
K = 10**3
def _tmp1():
    _mean = sum(v for v in x[0]) / 2048
    return sum((v - _mean)**2 for v in x[0]) / 2048
timeit(_tmp1, number=K) / K * 10**6
```

```
752.6543999999831
```

```
def _tmp2():
    _mean = 0
    for i in range(2048):
        _mean += x[0][i]
    _mean = _mean / 2048

    _var = 0
    for i in range(2048):
        _var += (x[0][i] - _mean)**2
    return _var / 2048
timeit(_tmp2, number=K) / K * 10**6
```

```
1233.996599999955
```

Runtime Comparison:

Native $\approx$ Numpy dense (15us)

$<$ Numpy strided (37us $\approx$ 2.5x)

$<$ Python iterator (753us $\approx$ 50x)

$<$ Python naive (1234us $\approx$ 80x)

Example

```
x = np.random.uniform(size=(2048, 2048))
K = 100000
```

```
timeit(lambda: np.var(x[0]), number=K) / K * 10**6
```

```
15.442851000000248
```

```
timeit(lambda: np.var(x[:, 0]), number=K) / K * 10**6
```

```
37.20540700000129
```

```
K = 10**3
def _tmp1():
    _mean = sum(v for v in x[0]) / 2048
    return sum((v - _mean)**2 for v in x[0]) / 2048
timeit(_tmp1, number=K) / K * 10**6
```

```
752.6543999999831
```

```
def _tmp2():
    _mean = 0
    for i in range(2048):
        _mean += x[0][i]
    _mean = _mean / 2048

    _var = 0
    for i in range(2048):
        _var += (x[0][i] - _mean)**2
    return _var / 2048
timeit(_tmp2, number=K) / K * 10**6
```

```
1233.996599999955
```

Runtime Comparison:

Native $\approx$ Numpy dense (15us)

$<$ Numpy strided (37us $\approx 2.5x$)

$<$ Python iterator (753us $\approx 50x$)

$<$ Python naive (1234us $\approx 80x$)

Vectorize, vectorize, vectorize!

- Use numpy and other library functions whenever possible
- Use dense array representations
- Avoid python iterators
- Avoid indexing numpy arrays