



COMP 2211 Exploring Artificial Intelligence
Python Fundamentals for Artificial Intelligence
Dr. Desmond Tsoi

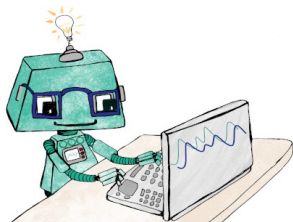
Department of Computer Science & Engineering
The Hong Kong University of Science and Technology, Hong Kong SAR, China



Why More Python?

You have taken COMP 1021/COMP 1029P. So you can program in Python, right?

- Think about this: You have been learned **Python (mostly Turtle)** in **COMP1021**, but can you write AI programs?
- You basically have learned the basics of Python in COMP1021/COMP1029P with a brief introduction to Python classes, and you can write small Python programs.
- In this topic, we will give you a crash course on both the **Python programming language** and **other essentials for writing AI programs**.



Part I

Crash Course on Python Programming Language



Python

- Python is a high-level, dynamically typed multiparadigm programming language.
- Python code is often said to be almost like pseudocode, since it allows you to express very powerful ideas in very few lines of code while being very readable.




A Python Program that Prime Factorizes a Number

```
# File: prime_factorize.py
```

```
def prime_check(n):  
    for i in range(2,n):  
        if n%i == 0: return False  
    return True
```

```
n = int(input("Enter the Number: "))  
print(str(n)+" = ",end='')
```

15 16 17 18 19 20 21
24 26 27 28 29 30
33 36 37 38 39
42 44 45 46 47 48



```
for i in range (2,n+1):  
    c = 0  
    if prime_check(i) == True:  
        while True:  
            if(n%i == 0):  
                n /= i  
                c+= 1  
            else: break  
        if c == 1:  
            print(str(i)+" x ",end='')  
        elif c !=0:  
            print(str(i)+"^"+str(c)+" x ",end='')  
  
print(" \b\b\b ")
```

Python Versions

- As of January 1, 2020, Python has officially dropped support for python2.
- For this course, all code will use [Python 3.7](#).
- The latest version is Python 3.10.2.
- You can double-check your Python version at the command line after activating your environment by running

```
import sys
print(sys.version)
```



Data Types and Arithmetic Operations

- Like most languages, Python has a number of **basic types** including **integers**, **floats**, **booleans**, **strings**, and **containers** including **lists**, **dictionaries**, **set**, **tuples**.
- The basic types behave in ways that like the other programming languages.

Name	Type	Description
Integers	int	Whole numbers, such as: 3, 300, 200
Floating point	float	Numbers with a decimal point: 2.3, 4.6, 100.0
Booleans	bool	Logical value indicating True or False
Strings	str	Ordered sequence of characters: "Hello", 'Desmond', "2000"
Lists	list	Ordered sequence of objects: [10, "Hello", 200.3]
Dictionaries	dict	Unordered Key:Value pairs: {"mykey" : "value", "name" : "Desmond"}
Sets	set	Unordered collection of unique objects: {"a", "b"}
Tuples	tup	Ordered immutable sequence of objects: ("a", "b")

type() method **returns class type** of the argument(object) passed as parameter. **type()** function is mostly used for debugging purposes.

Syntax

```
type(<object>)
```

Integers and Floats

- Python supports **integers** and **floating-point numbers**.
- It also implements all **arithmetic operators**, i.e.,
 $+$, $-$, $*$, $/$, $//$, $\%$, $**$, $()$, $+=$, $-=$, $*=$, $/=$

Note:

Unlike many languages, Python does not have increment ($x++$, $++x$) or decrement ($x--$, $--x$) operators

```
x = 3
print(type(x))           # Print "<class 'int'>"
print(x)                 # Print "3"
print(x + 1)             # Addition; print "4"
print(x - 1)             # Subtraction; print "2"
print(x * 2)             # Multiplication; print "6"
print(x / 2)             # Division, print "1.5"
print(x // 2)            # Integer division, print "1"
print(x % 2)             # Modulus; print "1"
print(x ** 2)            # Exponentiation; print "9"
x += 1; print(x)         # Print "4"
x *= 2; print(x)         # Print "8"
y = 2.5; print(type(y))  # Print "<class 'float'>"
print(y, y + 1, y * 2, y ** 2) # Print "2.5 3.5 5.0 6.25"
```


Booleans

- In Python, **booleans** represent one of two values: **True** or **False**.
- Python implements all the **relational operators** (comparison operators), i.e., `==`, `!=`, `>`, `>=`, `<`, `<=`, which returns **True** or **False**.
- Python also implements all of the usual operators for Boolean logic, but **uses English (and/or)** rather than symbols (`&&`, `||`, etc.).

```
print(1 + 1 != 3)  # Print "True"
print(4 + 6 == 10) # Print "True"
x = 3; y = 5
print(x + y > 7)   # Print "True"
print(x * y <= 7)  # Print "False"
```

```
t = True; f = False
print(type(t)) # Print "<class 'bool'>"
print(t and f) # Logical AND; print "False"
print(t or f)  # Logical OR; print "True"
print(not t)   # Logical NOT; print "False"
print(t != f)  # Logical XOR; print "True"
```

Strings

- A **string** is a **sequence of characters**.
- Strings in Python are **surrounded by either single quotation marks, or double quotation marks**.
- Python has **great support** for **strings**.

```
hello = 'hello'           # String literals can use single quotes
world = "world"           # or double quotes; it does not matter
print(hello)              # Print "hello"
print(len(hello))         # String length; print "5"
hw = hello + ' ' + world  # String concatenation
print(hw)                 # Print "hello world"
hw12 = '%s %s %d' % (hello, world, 12) # sprintf style string formatting
print(hw12)               # Print "hello world 12"
print(type(hw12))         # Print "<class 'str'>"
print(hello * 3)          # Print "hellohellohello",
                          # i.e., "hello" is duplicated by 3 times
```

Strings

- We can add a prefix in front of strings.

Prefix	Meaning	Example
u or U	Unicode string (default in Python 3)	<code>u"Desmond"</code>
b or B	Byte (ASCII) string Bytes are machine-readable (can be directly stored on the disk, need decoding to become human-readable) and string is human-readable (need encoding before they can be stored on disk)	<code>b"Desmond"</code>
r or R	Raw string, i.e., a character following a backslash is included in the string without change. This is useful when we want to have a string that contains backslash and do not want it to be treated as an escape character.	<code>r"\nDesmond\n"</code>
f or F	Formatted string, i.e., contains expressions inside braces	<code>a = 1</code> <code>b = 2</code> <code>print(f'a+b={a+b}')</code>

Examples

```
unicode_string = u'COMP2211'  
print(unicode_string)      # Print "COMP2211"
```

```
byte_string = b'COMP2211'  
print(type(byte_string))   # Print "<class 'bytes'>"  
decode_string = byte_string.decode('utf-8')  
print(type(decode_string)) # Print "<class 'str'>"  
encode_string = decode_string.encode('utf-8')  
print(type(encode_string)) # Print "<class 'bytes'>"
```

```
raw_string = r'Hi\nHello'  
print(raw_string)        # Print "Hi\nHello"
```

```
a = 1  
b = 2  
print(f'a + b = {a+b}')
```

```
# Print "a + b = 3"
```



String Objects

- **Strings** are actually **objects** in Python.
- **String objects** have a bunch of **useful methods**; for example:

```
s = "hello"
print(s.capitalize())    # Capitalize a string; prints "Hello"
print(s.upper())         # Convert a string to uppercase; prints "HELLO"
print(s.rjust(7))        # Right-justify a string, padding with spaces;
                        # Print "  hello"
print(s.center(7))       # Center a string, padding with spaces;
                        # Print " hello "
print(s.replace('l', '(ell)')) # Replace all instances of one substring
                        # with another; print "he(ell)(ell)o"
print('  world '.strip()) # Strip leading and trailing whitespace;
                        # print "world"
```

Full List of all String Methods

<https://docs.python.org/3.5/library/stdtypes.html#string-methods>

Type Conversion

- We can convert the data type of an object to required data type using the predefined functions like `int()`, `float()`, `bool()`, `str()`, etc to perform **explicit type conversion**.

Syntax

`<required-datatype>(<expression>)`

- Parameters:
 - required datatype: int, float, bool, str, etc.
 - expression: The expression to be type-converted

```
print(int('3') + 5)    # Print "8"
print('3' + str(5))    # Print "35"
print(float(3) + 3)     # Print "6.0"
print(int(3.14) + 3)    # Print "6"
```

Output Data to the Standard Output Device

- `print()` function is used to output data to the standard output device (screen).

Syntax

```
print(*<objects>, <sep>=' ', <end>='\n', <file>=sys.stdout, <flush>=False)
```

- Parameters:
 - objects: The value(s) to be printed
 - sep: The separator is used between the values. It defaults into a space character.
 - end: After all values are printed, end is printed. It defaults into a new line.
 - file: The object where the values are printed and its default value is `sys.stdout` (screen).
 - flush: To ensure that we get output as soon as `print()` is called, we need to set `flush` to `True`.

Output formatting can be done by using `str.format()` method. `{}` are used as placeholders, and we can specify the order in which they are printed by using numbers. Also, we can even use keyword arguments to format the string.

Output Data to Standard Output Device

```
print('COMP2211 is the best COMP course :D') # Print a string
```

```
age = 18
```

```
print('I am', age, 'years old')           # Print 'I am 18 years old'
```

```
print(2, 2, 1, 1)                         # Print 2 2 1 1
```

```
print(2, 2, 1, 1, sep='@')               # Print 2@2@1@1
```

```
print(2, 2, 1, 1, sep='~', end='*')      # Print 2~2~1~1*
```

```
print()                                   # Print '\n', i.e., move to the next line
```

```
x = 'A+'; y = 'A'
```

```
# Print 'Desmond will get A+ and John will get A'
```

```
print('Desmond will get {} and John will get {}'.format(x,y))
```

```
# Print 'Desmond will get A+ and John will get A'
```

```
print('Desmond will get {0} and John will get {1}'.format(x,y))
```

```
# Print 'Desmond will get A and John will get A+'
```

```
print('Desmond will get {1} and John will get {0}'.format(x,y))
```

```
# Print 'Desmond will get A+ and John will get A'
```

```
print('Dsemond will get {a} and John will get {b}'.format(a = 'A+', b = 'A'))
```


Input Data

- We can take the input from the user using `input()` function.

Syntax

`input(<prompt>)`

- Parameter:
 - prompt: The string we wish to display on the screen. It is optional.

Note: The entered data is a string.

```
age = input('Enter your age: ')
print(age)           # Assume the input is 18. It prints "18"
print(type(age))     # Print "<class 'str'>"
print(type(int(age))) # Print "<class 'int'>"
print(type(float(age))) # Print "<class 'float'>"
age = int(age) + 1    # Convert age to int and increase it by 1
print('Now you are', age, 'years old') # Print "Now you are 19 years old"
```

Containers

- Python includes several **built-in container types**
 - Lists
 - Dictionaries
 - Sets
 - Tuples



Data Structure	Ordered?	Duplicate?	Indexing/ Slicing?	Changeable/ Mutable?	Constructor	Example
List	Yes	Yes	Yes	Yes	[] or list()	[5.7, 4, 'yes', 5.7]
Dictionary	No	No	Yes	Yes	{ } or dict()	{'Jun':75, 'Jul':89}
Set	No	No	No	Yes	{ } or set()	{5.7, 4, 'yes'}
Tuple	Yes	Yes	Yes	No	() or tuple()	(5.7, 4, 'yes', 5.7)

Lists

- A **list** is the Python **equivalent of an array**, but is **resizeable** and can **contain elements of different types**.

Syntax

```
<list-name> = [<value1>, <value2>, <value3>, ...]
```

- Parameters:
 - list-name: A variable name of a list
 - value1, value2, value3, ...: List values

Note: List literals are written within square brackets [].

- The **list items** can be **accessed using index operator** ([]) – not to be confused with an empty list). The expression inside the brackets specifies the index.
 - The **first element has index 0**, and the **last element has index (# of elements in the list - 1)**.
 - **Negative index** values will locate items **from the right** instead of from the left.

Lists

- The `append()` method adds an item to the end of the list.

Syntax

```
<list-name>.append(<item>)
```

- Parameters:
 - list-name: The list that we want to append the element to
 - item: An item (number, string, list, etc.) to be added at the end of the list
- The `+=` operator adds a list of elements to the list

Syntax

```
<list-name> += <new-list-name>
```

- Parameters:
 - list-name: The list that we want to append the new list to
 - new-list-name: The new list of elements to add

Lists

- The `pop()` method removes and returns the last value from the list or the given index value.

Syntax

```
<list-name>.pop(<index>)
```

- Parameters:
 - list-name: The variable name of list that we want to remove an element
 - index: The value at index is popped out and removed. If the index is not given, then the last element is popped out and removed
- The `remove()` method removes the specified value from the list

Syntax

```
<list-name>.remove(<element>)
```

- Parameters:
 - list-name: The variable name of list that we want to remove an element.
 - element: The element that we want to remove. If the element does not exist, it throws an exception.

Lists

```
xs = [3, 1, 2]      # Create a list
print(xs, xs[2])    # Print "[3, 1, 2] 2"
print(xs[-1])       # Negative indices count from the end of the list; print "2"
xs[2] = 'foo'       # Lists can contain elements of different types
print(xs)           # Print "[3, 1, 'foo']"
xs.append('bar')     # Add a new element to the end of the list
print(xs)           # Print "[3, 1, 'foo', 'bar']"
x = xs.pop()        # Remove and return the last element of the list
print(x, xs)        # Print "bar [3, 1, 'foo']"
ys = ['a', 3, [4.5, 'c', True]] # Create another list
print(ys)           # Print ['a', 3, [4.5, 'c', True]]
print(ys[2][2])     # Print "True"
ys += [4, 5, 6]     # Add 3 elements, 4, 5, 6 to ys
print(ys)           # Print ['a', 3, [4.5, 'c', True], 4, 5, 6]
ys.remove(5)        # Remove 4.5 from the list
print(ys)           # Print ['a', 3, [4.5, 'c', True], 4, 6]
```

More Details About Lists

<https://docs.python.org/3.5/tutorial/datastructures.html#more-on-lists>

Slicing

- In addition to accessing list elements one at a time, Python provides concise syntax to **access sublists**; this is known as **slicing**.

Syntax

```
<new-list-name> = <list-name> [<start>:<end(exclusive)>:<jump>]
```

- Parameters:
 - list-name: The name of list to be sliced
 - new-list-name: The name of the extracted sub-list
 - start: The index of the start element
 - end: The index of the element after the last element
 - jump: The index step jump
- It returns the portion of the list from index “start” to index “end”(exclusive), at a step size “jump”.

Note: If **parameters are omitted**, the default value of **start**, **end**, and **jump** are **0**, (**#elements in the list**), **1**, respectively.

Slicing

```
nums = list(range(5))  
  
print(nums)  
print(nums[2:4])  
  
print(nums[2:])  
  
print(nums[:2])  
  
print(nums[:])  
  
print(nums[:-1])  
  
print(nums[0:4:2])  
nums[2:4] = [8, 9]  
print(nums)
```

*# range is a built-in function that creates a
list of integers, [0, 1, 2, 3, 4]
Print "[0, 1, 2, 3, 4]"
Get a slice from index 2 to 4 (exclusive);
print "[2, 3]"
Get a slice from index 2 to the end;
print "[2, 3, 4]"
Get a slice from the start to index 2 (exclusive);
print "[0, 1]"
Get a slice of the whole list;
print "[0, 1, 2, 3, 4]"
Slice indices can be negative;
print "[0, 1, 2, 3]"
Print "[0, 2]"
Assign a new sublist to a slice
Print "[0, 1, 8, 9, 4]"*

Loops

- You can **loop over the elements of a list** like this:

```
animals = ['cat', 'dog', 'monkey']  
for animal in animals:  
    print(animal)  # Prints "cat", "dog", "monkey", each on its own line.
```

```
# Alternative  
# for index in range(3):  
#     print(animals[index])  
# Prints "cat", "dog", "monkey", each on its own line.
```

```
i = 0  
while i < len(animals):  
    print(animals[i])  
    i = i + 1      # Prints "cat", "dog", "monkey", each on its own line.
```

List Comprehensions

- List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

Syntax

```
<new-list-name> = [ <expression> for <element> in <list-name> if <condition> ]
```

- List comprehensions start and end with opening and closing square brackets.
- Parameters:
 - expression: Operation we perform on each value inside the original list
 - element: A temporary variable we use to store for each item in the original list
 - list-name: The name of the list that we want to go through
 - condition: If condition evaluates to True, add the processed element to the new list
 - new-list-name: New values created which are saved

Note: There can be an optional if statement and additional for clause.

List Comprehensions

- As a simple example, consider the following code that computes square numbers:

```
nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x ** 2)
print(squares)    # Prints [0, 1, 4, 9, 16]
```

- You can make this code simpler using a **list comprehension**:

```
nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print(squares)    # Prints [0, 1, 4, 9, 16]
```

- List comprehensions** can also contain conditions:

```
nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]
print(even_squares)    # Prints "[0, 4, 16]"
```

Dictionaries

- A **dictionary** stores **(key, value) pairs**, similar to a Map in Java or an object in Javascript.

Syntax

```
<dict-name> = { <key1>:<value1>, <key2>:<value2>, <key3>:<value3>, ... }
```

- Parameters:
 - dict-name: The name of a dictionary
 - key1, key2, key3, ...: The keys
 - value1, value2, value3, ...: The values

Note: Dictionary literals are written within curly brackets { }.

- Items of a dictionary can be **accessed by referring to its key name, inside square brackets**.

Dictionaries

- Adding an item to the dictionary is done by using a new index key and assigning a value

Syntax

```
<dict-name>[<new-key>] = <new-value>
```

- Parameters:
 - dict-name: The name of a dictionary
 - new-key: The key name of the item you want to add
 - new-value: The corresponding value of the item you want to add

- The `get()` method returns the value of the item with the specified key.

Syntax

```
<dict-name>.get(<keyname>,<value>)
```

- Parameters:
 - dict-name: The name of a dictionary
 - keyname: The key name of the item you want to return the value from
 - value: Optional. A value to return if the specified key does not exist. Default value None

Dictionaries

- The `pop()` method removes the specified item from the dictionary. The value of the removed item is the return value of the `pop()` method.

Syntax

```
<dict-name>.pop(<keyname>, <default-value>)
```

- Parameters:
 - `dict-name`: The name of a dictionary
 - `keyname`: The key name of the item you want to remove
 - `default-value`: A value to return if the specified key do not exist. If this parameter is not specified, and the item with the specified key is not found, an error is raised

Dictionaries

```
• d = {'cat': 'cute', 'dog': 'furry'} # Create a new dictionary with some data
print(d['cat'])                       # Get an entry from a dictionary; prints "cute"
print('cat' in d)                     # Check if a dictionary has a given key; print "True"
d['fish'] = 'wet'                     # Set an entry in a dictionary
print(d['fish'])                       # Print "wet"
# print(d['monkey'])                  # KeyError: 'monkey' not a key of d
print(d.get('monkey', 'N/A'))          # Get an element with a default; print "N/A"
print(d.get('fish', 'N/A'))            # Get an element with a default; print "wet"
del d['fish']                          # Remove an element from a dictionary
print(d.get('fish', 'N/A'))            # "fish" is no longer a key; print "N/A"
item = d.pop('cat')                    # Remove the item with key 'cat' from the dictionary
print(item)                           # Print "cute"
print(d)                              # Print {'dog': 'furry'}
```

More Details About Dictionaries

<https://docs.python.org/3.5/library/stdtypes.html#dict>

Dictionaries

- It is easy to iterate over the keys in a dictionary:

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal in d:
    legs = d[animal]
    print('A %s has %d legs' % (animal, legs))
# Print "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

- If you want to access keys and their corresponding values, use the `items` method:

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    print('A %s has %d legs' % (animal, legs))
# Print "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

- **Dictionary comprehensions:** These are similar to list comprehensions, but allow you to easily construct dictionaries. For example:

```
nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
print(even_num_to_square) # Print "{0: 0, 2: 4, 4: 16}"
```


Sets

- A **set** is an **unordered collection of distinct elements** (i.e., no duplicates).

Syntax

```
<set-name> = { <value1>, <value2>, <value3>, ... }
```

- Parameters:
 - set-name: The name of a set
 - value1, value2, value3, ...: Set values

Note: Set literals are written within curly brackets { }.

- A **set** is **unchangeable** (i.e., cannot change the items after the set has been created), and unindexed. But we **can add new items and remove items**.

Sets

- The `add()` method adds a given element to a set if the element is not present in the set.

Syntax

```
<set-name>.add(<element>)
```

- Parameters:
 - set-name: The name of a set that we want to add an element to
 - element: The element that we want to add

- The `remove()` method removes the specified element from the set.

Syntax

```
<set-name>.remove(<element>)
```

- Parameters:
 - set-name: The name of a set that we want to remove an element from
 - element: The element that we want to remove

Sets

- As a simple example, consider the following:

```
animals = {'cat', 'dog'}
print('cat' in animals)    # Check if an element is in a set; prints "True"
print('fish' in animals)   # Print "False"
animals.add('fish')         # Add an element to a set
print('fish' in animals)   # Print "True"
print(len(animals))        # Number of elements in a set; prints "3"
animals.add('cat')          # Adding an element that is already in the set
                             # does nothing
print(len(animals))        # Print "3"
animals.remove('cat')       # Remove an element from a set
print(len(animals))        # Print "2"
```

More Details About Sets

<https://docs.python.org/3.5/library/stdtypes.html#set>

Sets

- Loops: Iterating over a set has the same syntax as iterating over a list; however since sets are unordered, you **cannot make assumptions about the order in which you visit the elements of the set**:

```
animals = {'cat', 'dog', 'fish'}
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
# Prints
# #1: cat
# #2: dog
# #3: fish
```

- Set comprehensions**: Like lists and dictionaries, we can easily construct sets using set comprehensions:

```
from math import sqrt
nums = {int(sqrt(x)) for x in range(30)}
print(nums) # Print "{0, 1, 2, 3, 4, 5}"
```

Tuples

- A **tuple** in an (immutable/unchangeable) ordered list of values.

Syntax

```
<tuple-name> = (<value1>, <value2>, <value3>, ...)
```

- Parameters:

- tuple-name: The variable name of a tuple that we want to add element(s) to
- value1, value2, value3, ...: The list of values that we want to add to the tuple

Note: Tuple literals are placed inside parentheses (), separated by commas.

- The **tuple items** can be **accessed using index operator** ([] – not to be confused with an empty list). The expression inside the brackets specifies the index.
 - The **first element has index 0**, and the **last element has index (#elements in the tuple - 1)**.
 - **Negative index** values will **locate items from the right** instead of from the left.

Tuples

- A **tuple** is in many ways **similar to a list**; one of the **most important differences** is that **tuples can be used as keys in dictionaries and as elements of sets**, while **lists cannot**.
- Here is a trivial example:

```
d = {(x, x + 1): x for x in range(10)}  # Create a dictionary with tuple keys
t = (5, 6)                             # Create a tuple
print(type(t))                         # Print "<class 'tuple'>"
print(d[t])                           # Print "5"
print(d[(1, 2)])                       # Print "1"
```

More Details About Tuples

<https://docs.python.org/3.5/tutorial/datastructures.html#tuples-and-sequences>

Parallel Iteration

- The `zip()` method takes iterable or containers and returns a single iterator object, having mapped values from all the containers.
- If the passed iterable or containers have different lengths, the one with the least items decides the length of the new iterator.

Syntax

```
zip(<iterator1>, <iterator2>, <iterator3>...)
```

- Parameters:
 - `iterator1`, `iterator2`, `iterator3`, ...: Iterator objects that will be joined together

Parallel Iteration

```
fruits = ['apple', 'peach', 'banana', 'guava', 'papaya']  
colors = ['red', 'pink', 'yellow', 'green', 'orange']
```

```
for name, color in zip(fruits, colors):  
    print(name, 'is', color)
```

```
# It prints  
# apple is red  
# peach is pink  
# banana is yellow  
# guava is green  
# papaya is orange
```


Functions

- Python **functions** are **defined using def keyword**.
- The **return keyword** is to **exit a function and return a value**.

```
def sign(x):  
    if x > 0:  
        return 'positive'  
    elif x < 0:  
        return 'negative'  
    else:  
        return 'zero'
```

```
for x in [-1, 0, 1]:  
    print(sign(x))
```

It prints
negative
zero
positive



Functions

- Normally, when we create a **variable inside a function**, that variable is **local**, and **can only be used inside that function**.
- To create a **global variable** inside a function, we can use the **global keyword**.

```
def assign_word():  
    global word  
    word = "cool"  
  
assign_word()  
  
# Print "Desmond is really cool"  
print("Desmond is really " + word)
```



Functions

- If you **do not know how many arguments** that will be passed into your function, **add an asterisk (i.e., *) before the parameter name** in the function definition. This way the function will receive a **tuple of arguments**, and can access the items accordingly:

```
def print_kids(*kids):  
    print("The youngest child is " + kids[2])
```

```
# Print "The youngest child is John"  
print_kids("Desmond", "Peter", "John")
```



Functions

- If you **do not know how many keyword arguments** that will be passed into your function, **add two asterisk: **** before the **parameter name** in the function definition. This way the function will receive a **dictionary of arguments**, and can access the items accordingly:

```
def print_kid(**kid):  
    print("His first name is " + kid["fname"] + ", last name is " + kid["lname"])  
  
# Print "His first name is Desmond, last name is Tsoi"  
print_kid(fname = "Desmond", lname = "Tsoi")
```



Functions

- Function can be defined with arguments that have default values.

```
def hello(name, loud=False):  
    if loud:  
        print('HELLO, %s!' % name.upper())  
    else:  
        print('Hello, %s' % name)  
  
hello('Bob')           # Print "Hello, Bob"  
hello('Fred', loud=True) # Print "HELLO, FRED!"
```

More Details About Functions

<https://docs.python.org/3.5/tutorial/controlflow.html#defining-functions>

Classes

- A **class** is a **user-defined data type** from which objects are created. It is created by **keyword class**.
- Class provides a means of **bundling data (i.e., instance variables) and functionality (i.e., methods) together**.
- Instance variables:
 - They are the variables that belong to an object.
 - They are **always public by default** and can **accessed using the dot (.) operator**.
- Methods:
 - They have an extra **first parameter, self**, in the method definition.
 - We **do not give a value for this parameter** when we call the method, Python provides it.
- The instance variables and methods are accessed by using the object.
- **`__init__`** method is similar to constructors in Java/C++. **Constructors** are used to **initializing the instance variables of objects**.

Classes

- Define a class with instance variables and methods (and a constructor).

Syntax

```
class <class-name>:
    def __init__(self, <arguments0>):
        self.<instance-variable1> = <value1>
        self.<instance-variable2> = <value2>
        ...
    def <method1-name>(self, <arguments1>):
        <statement1>
        <statement2>
        ...
    def <method2-name>(self, <arguments2>):
        <statement1>
        <statement2>
        ...
```

Parameters:

- class-name: The name of the class.
- arguments0, arguments1, arguments2, ...: Method parameters (i.e., variables)
- instance-variable1, instance-variable2, ...: Instance variables
- value1, value2, ...: Value assigned to the instance variables
- statement1, statement2, ...: Python statements

Classes

- Create an object, call a method, and modify an instance variable

Syntax

```
<object-name> = <class-name>(<arguments>)  
<object-name>.<method-name>(<arguments>)  
<object-name>.<instance-variable-name> = <value>
```

- Parameters:
 - object-name: The name of an object
 - class-name: The name of a class
 - arguments: The values that we pass to the constructor/method
 - method-name: The name of the method
 - instance-variable-name: The name of an instance variable
 - value: The value assigned to the instance variable

Classes: Public, Private and Protected Members

- As mentioned, all members in a Python class are public by default, i.e., any member can be accessed from outside the class. To restrict the access to the members, we can make them protected/private.
- **Protected members** of a class are **accessible from within the class and also available to its sub-classes** (Will not be covered in this course). **By convention**, Python makes an/a instance variable/method protected by **adding a prefix `_`** (single underscore) to it.
- **Private members** of a class are **accessible from within the class only**. **By convention**, Python makes an instance variable/method private by **adding a prefix `__`** (double underscore) to it.

Note

“By convention” means the responsible programmer would refrain from accessing and modifying instance variables prefixed with `_` or `__` from outside its class. But if they do so, still works. :(

Example

```
class Person(object):
    def __init__(self, name='Tom',
                  age=18,
                  gender='M'):
        self.__name = name
        self.__age = age
        self.__gender = gender

    def get_name(self):
        return self.__name

    def get_age(self):
        return self.__age

    def get_gender(self):
        return self.__gender

    def set_name(self, name):
        self.__name = name
```

```
    def set_age(self, age):
        self.__age = age

    def set_gender(self, gender):
        self.__gender = gender

    def print(self):
        print('--- Print Person ---')
        print('Name: ' + self.__name + ' ')
        print('Age: ' + str(self.__age) + ' ')
        print('Gender: ' + self.__gender)

desmond = Person('Haha', 18, 'M')
print('Name: ' + desmond.get_name())
print('Age: ' + str(desmond.get_age()))
print('Gender: ' + desmond.get_gender())
desmond.set_name('Desmond')
desmond.set_age(19)
desmond.print()
```

Part II

Other Essentials for Writing AI Programs



Modules, Packages, Libraries

Modules

A **modules** in Python is a bunch of **related code** (functions and values) saved in a file with the extension `.py`.

Packages

A **package** is basically a **directory of a collection of modules**.

Libraries

A **library** is a **collection of packages**.



Imports

- The `import` keyword is the most common way to **access external modules**.
- **Modules** are **files (.py)** that **contain functions and values** that you can reference in your program without having to write them yourself.
- This is how you get access to machine learning libraries, like `numpy`, `keras`, `sklearn`, `tensorflow`, `pandas`, `pytorch`, `matplotlib`, etc.
- `import` is similar to the `#include <something.h>` in C++.
- However, you will **usually only need to import a few functions** from each of those libraries.
- In that case, we can use the `from` keyword to **import only the exact functions** we are using in the code.

Syntax

```
from [module] import [function/value]
```

Imports

- For instance, if we only want to import the `sqrt` function from the `math` library, we just need to do:

```
from math import sqrt # Import only sqrt function from math module  
x = 100  
# Now, it's imported, we can directly use sqrt to call the math.sqrt()  
print(sqrt(x)) # It print 10
```

- There is also the “`from [module] import *`” syntax which imports all functions. So the following code works as well.

```
from math import * # Import ALL functions from math module  
x = 100  
print(sqrt(x)) # It print 10
```

Imports

- However, we should generally **avoid importing all functions** since we do not know if `sqrt` function is from the `math` library. If we use the `import all` syntax for multiple libraries, we may end up with clashes or ambiguity, which decreases the readability of our code and makes it harder to spot errors.
- One such example:

```
# Import ALL functions from math module, including math.sqrt()  
from math import *  
# Import ALL functions from cmath, including cmath.sqrt()  
from cmath import *  
  
x = 100  
# The following will use cmath.sqrt(), which is a different implementation to  
# math.sqrt()  
print(sqrt(x)) # It print 10
```

Imports

- One of the most frequently used machine learning libraries is sklearn.
- For instance, you may use the K-Means algorithm (we will cover this later in class) to separate your data into groups.
- Let's look at the scikit-learn documentation
<https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html#sklearn.cluster.KMeans>
and try to understand it.

```
class sklearn.cluster.KMeans( n_clusters=8, *, init='k-means++', n_init=10,  
                               max_iter=300, tol=0.0001, verbose=0,  
                               random_state=None, copy_x=True, algorithm='auto')
```

Each “.” refers to accessing something inside a package.

Imports

- For instance, the following is a simplified layout of sklearn:

```
sklearn/  
  __init__.py  
  discriminant_analysis.py  
  cluster/  
    KMeans.py  
    SpectralClustering.py  
    affinity_propagation.py  
    ...  
  linear_model/  
    bayes.py  
    logistic.py  
  neural_network/  
    multilayer_perceptron.py
```



Imports

- `import sklearn.cluster` means we import all the modules inside the package `sklearn.cluster`, including but not limited to (`KMeans`, `SpectralClustering`, `affinity_propagation`.)
- `import sklearn.cluster.KMeans` means we only import the module `KMeans` inside the package `sklearn.cluster`.
- If we want to call K-Means on our data, we need to call `sklean.cluster.KMeans`. But that's quite a long function call, so we typically declare the everything but the name of the actual module in the import section, like so:

```
# Import KMeans module from package sklearn.cluster  
from sklearn.cluster import KMeans  
kmeans = KMeans(n_clusters=2, random_state=600)
```

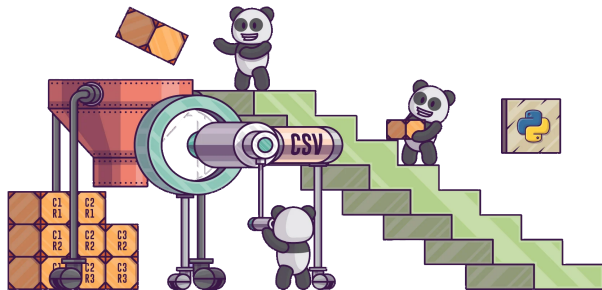
Notice that we can just use the function name directly in the 2nd line as `KMeans`.

- If our function is particularly long and we do not want to type it out every time, we can shorten it by using the `as` keyword to create an alias (a “nickname”) to use in place of the original (and long) function name.

```
from sklearn.cluster import KMeans as km  
kmeans = km(n_clusters=2, random_state=600)
```

Importing and Exporting Data

- Your data can come in many forms, which are too exhaustive to list here.
- We will introduce how to deal with **data in csv (comma-separated-values)** form, since that is fairly common.
- In Python, most data manipulation is done with the pandas library.



Real Python

Importing Data

- To import data from Google Drive, we need to “mount” the directory.
- It will ask you to visit a link to allow Colab to access your drive.

```
from google.colab import drive  
drive.mount('/content/drive')
```

- Copy the alphanumeric code and paste it into your notebook.
- Afterwards, Drive files will be mounted, which you can view in the sidebar to the left.
- To **access file** training_data.csv in the root “My Drive” folder, for instance:

```
df = pd.read_csv('/content/drive/My Drive/training_data.csv', index=False)
```

Exporting Data

- To **save a file** in the root “My Drive” folder, we do

```
import pandas as pd
# Create a pandas.DataFrame object with two columns:
# 'CustomerId' and 'CanRepayLoan'
output = pd.DataFrame({'CustomerId': np.array([1, 2, 3]),
                       'CanRepayLoan': np.array([0, 0, 1])})
# Convert the pandas.DataFrame object into the csv "loans .csv" and
# Export it to the root folder of My Drive
output.to_csv('/content/drive/My Drive/loans.csv', index=False)
```

We will cover NumPy array very soon! :)

Numpy

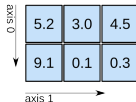
- NumPy is a short form for Numerical Python and is the core library for numeric and scientific computing in Python.
- It provides high-performance multidimensional array object, and tools for working with these arrays.

1D array



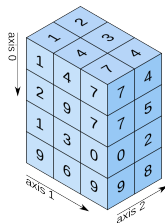
shape: (4,)

2D array



shape: (2, 3)

3D array



shape: (4, 3, 2)



Arrays

- A **numpy array** is a **grid of values**, all of the **same type**, and is **indexed by a tuple of non-negative integers**.
- The **number of dimensions** is the **rank of the array**.
- The **shape of an array** is a **tuple of integers giving the size** of the array along each dimensions.
- We can initialize numpy arrays from nested Python lists, and access elements using square brackets.

```
import numpy as np
```

```
a = np.array([1, 2, 3])           # Create a rank 1 array
print(type(a))                   # Print "<class 'numpy.ndarray'>"
print(a.shape)                   # Print "(3,)"
print(a[0], a[1], a[2])          # Print "1 2 3"
a[0] = 5                         # Change an element of the array
print(a)                         # Print "[5, 2, 3]"
b = np.array([[1,2,3],[4,5,6]])  # Create a rank 2 array
print(b.shape)                   # Print "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0]) # Print "1 2 4"
```

Arrays

- Numpy also provides many functions to create arrays.

```
import numpy as np
```

```
a = np.zeros((2,2))          # Create an array of all zeros
print(a)                    # Print "[[ 0.  0.]
                             #          [ 0.  0.]]"

b = np.ones((1,2))          # Create an array of all ones
print(b)                    # Print "[[ 1.  1.]]"

c = np.full((2,2), 7)       # Create a constant array
print(c)                    # Print "[[ 7.  7.]
                             #          [ 7.  7.]]"

d = np.eye(2)               # Create a 2x2 identity matrix
print(d)                    # Print "[[ 1.  0.]
                             #          [ 0.  1.]]"

e = np.random.random((2,2)) # Create an array filled with random values
print(e)                    # Might print "[[ 0.91940167  0.08143941]
                             #          [ 0.68744134  0.87236687]]"
```


Array Indexing

Expression	Description
<code>a[i]</code>	Select element at index <code>i</code> , where <code>i</code> is an integer (start counting from 0).
<code>a[-i]</code>	Select the <code>i</code> th element from the end of the list, where <code>n</code> is an integer. The last element in the list is addressed as <code>-1</code> , the second to last element as <code>-2</code> , and so on.
<code>a[i:j]</code>	Select elements with indexing starting at <code>i</code> and ending at <code>j-1</code> .
<code>a[:]</code> or <code>a[0:-1]</code>	Select all elements in the given axis.
<code>a[:i]</code>	Select elements starting with index 0 and going up to index <code>i - 1</code> .
<code>a[i:]</code> or <code>a[i:-1]</code>	Select elements starting with index <code>i</code> and going up to the last element in the array.
<code>a[i:j:n]</code>	Select elements with index <code>i</code> through <code>j</code> (exclusive), with increment <code>n</code> .
<code>a[::-1]</code>	Select all the elements, in reverse order.

Array Indexing

- Similar to Python lists, **numpy arrays can be sliced**. Since arrays may be multidimensional, you must specify a slice for each dimension of the array.

```
import numpy as np
```

```
# Create the following rank 2 array with shape (3, 4)
# [[1  2  3  4]
#  [5  6  7  8]
#  [9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]
# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])    # Prints "2"
b[0, 0] = 77      # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])    # Prints "77"
```

Array Indexing

- You can also **mix integer indexing with slice indexing**.
- However, doing so will **yield an array of lower rank** than the original array.

`import numpy as np`

```
# Create the following rank 2 array with shape (3, 4)
```

```
# [[ 1  2  3  4]
```

```
#  [ 5  6  7  8]
```

```
#  [ 9 10 11 12]]
```

```
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
```

```
# Mixing integer indexing with slices yields an array of lower rank,
```

```
# while using only slices yields an array of same rank as the original array:
```

```
row_r1 = a[1, :]      # Rank 1 view of the second row of a
```

```
row_r2 = a[1:2, :]   # Rank 2 view of the second row of a
```

```
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
```

```
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"
```

```
# We can make the same distinction when accessing columns of an array:
```

```
col_r1 = a[:, 1]
```

```
col_r2 = a[:, 1:2]
```

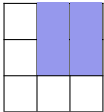
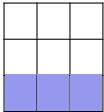
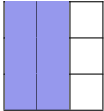
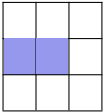
```
print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"
```

```
print(col_r2, col_r2.shape)  # Prints "[[ 2]
```

```
#      [ 6]
```

```
#      [10]] (3, 1)"
```

Arrays

	Expression	Shape
	<code>arr[:2, 1:]</code>	<code>(2, 2)</code>
	<code>arr[2]</code> <code>arr[2, :]</code> <code>arr[2:, :]</code>	<code>(3,)</code> <code>(1, 3)</code>
	<code>arr[:, :2]</code>	<code>(3, 2)</code>
	<code>arr[1, :2]</code> <code>arr[1:2, :2]</code>	<code>(2,)</code> <code>(1, 2)</code>

Integer Array Indexing

- Integer array indexing allows you to construct arbitrary arrays using the data from another array.

```
import numpy as np
```

```
a = np.array([[1,2], [3, 4], [5, 6]])
```

```
# An example of integer array indexing.
```

```
# The returned array will have shape (3,) and
```

```
print(a[[0, 1, 2], [0, 1, 0]]) # Prints "[1 4 5]"
```

```
# The above example of integer array indexing is equivalent to this:
```

```
print(np.array([a[0, 0], a[1, 1], a[2, 0]])) # Prints "[1 4 5]"
```

```
# When using integer array indexing, you can reuse the same
```

```
# element from the source array:
```

```
print(a[[0, 0], [1, 1]]) # Prints "[2 2]"
```

```
# Equivalent to the previous integer array indexing example
```

```
print(np.array([a[0, 1], a[0, 1]])) # Prints "[2 2]"
```

Integer Array Indexing

- One useful trick with **integer indexing** is **selecting or mutating one element from each row of a matrix**.

```
import numpy as np
```

```
# Create a new array from which we will select elements  
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
```

```
print(a) # prints "array([[ 1,  2,  3],  
#           [ 4,  5,  6],  
#           [ 7,  8,  9],  
#           [10, 11, 12]])"
```

```
# Create an array of indices
```

```
b = np.array([0, 2, 0, 1])
```

```
# Select one element from each row of a using the indices in b
```

```
print(a[np.arange(4), b]) # Prints "[ 1  6  7 11]"
```

```
# Mutate one element from each row of a using the indices in b
```

```
a[np.arange(4), b] += 10
```

```
print(a) # prints "array([[11,  2,  3],  
#           [ 4,  5, 16],  
#           [17,  8,  9],  
#           [10, 21, 12]])"
```

Boolean Array Indexing

- Boolean array indexing lets you pick out arbitrary elements of an array.
- Frequently this type of indexing is used to select the elements of an array that satisfy some condition.

```
import numpy as np
```

```
a = np.array([[1,2], [3, 4], [5, 6]])
```

```
bool_idx = (a > 2)    # Find the elements of a that are bigger than 2; this returns a numpy  
                     # array of Booleans of the same shape as a, where each slot of bool_idx tells  
                     # whether that element of a is > 2.
```

```
print(bool_idx)       # Prints "[False False]  
                     #           [ True  True]  
                     #           [ True  True]]"
```

```
# We use boolean array indexing to construct a rank 1 array consisting of the elements of a  
# corresponding to the True values of bool_idx
```

```
print(a[bool_idx])    # Prints "[3 4 5 6]"
```

```
# We can do all of the above in a single concise statement:
```

```
print(a[a > 2])       # Prints "[3 4 5 6]"
```

Datatypes

- Every numpy array is a grid of elements of the same type.
- Numpy provides a large set of numeric datatypes that you can use to construct arrays.
- Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an **optional argument to explicitly specify the datatype**.

```
import numpy as np
```

```
x = np.array([1, 2])    # Let numpy choose the datatype  
print(x.dtype)         # Prints "int64"
```

```
x = np.array([1.0, 2.0]) # Let numpy choose the datatype  
print(x.dtype)         # Prints "float64"
```

```
x = np.array([1, 2], dtype=np.int64) # Force a particular datatype  
print(x.dtype)         # Prints "int64"
```


Array Math

- Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module.

```
import numpy as np
```

```
x = np.array([[1,2],[3,4]], dtype=np.float64)
```

```
y = np.array([[5,6],[7,8]], dtype=np.float64)
```

```
# Elementwise sum; both produce the array
```

```
# [[ 6.0  8.0]
```

```
# [10.0 12.0]]
```

```
print(x + y)
```

```
print(np.add(x, y))
```

```
# Elementwise difference; both produce the array
```

```
# [[-4.0 -4.0]
```

```
# [-4.0 -4.0]]
```

```
print(x - y)
```

```
print(np.subtract(x, y))
```

```
# Elementwise product; both produce the array
```

```
# [[ 5.0 12.0]
```

```
# [21.0 32.0]]
```

```
print(x * y)
```

```
print(np.multiply(x, y))
```

```
# Elementwise division; both produce the array
```

```
# [[ 0.2          0.33333333]
```

```
# [ 0.42857143  0.5         ]]
```

```
print(x / y)
```

```
print(np.divide(x, y))
```

```
# Elementwise square root; produces the array
```

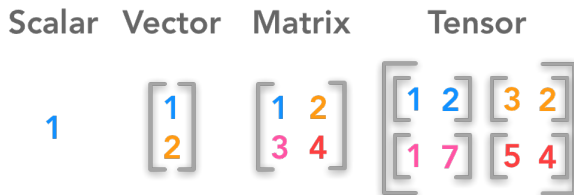
```
# [[ 1.          1.41421356]
```

```
# [ 1.73205081  2.         ]]
```

```
print(np.sqrt(x))
```

Vector, Matrix and Tensor

- **Vectors** are 1-dimensional arrays of numbers.
- **Matrices** are 2-dimensional arrays of numbers.
- **Tensors** are multi-dimensional arrays of numbers.
- They are useful in expressing numerical information, and are extremely useful in expressing different operations.
- So, they are important in statistics, machine learning and computer science.



Matrix Multiplications

- **Matrix multiplication** is one of the very important operation for artificial intelligence.
- Let's see how to perform matrix multiplication.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix} = \begin{bmatrix} (1)(7)+(2)(8)+(3)(9) \\ (4)(7)+(5)(8)+(6)(9) \end{bmatrix} = \begin{bmatrix} 7+16+27 \\ 28+40+54 \end{bmatrix} = \begin{bmatrix} 50 \\ 122 \end{bmatrix}$$

2 x 3 3 x 1 2 x 1 2 x 1 2 x 1

Array Math

- **Matrix multiplication** can be **performed using dot function** in Python.
- dot is available both as a function in the numpy module and as an instance method of array objects.

```
import numpy as np
```

```
x = np.array([[1,2],[3,4]])
```

```
y = np.array([[5,6],[7,8]])
```

```
v = np.array([9,10])
```

```
w = np.array([11, 12])
```

```
# Inner product of vectors; both produce 219
```

```
print(v.dot(w))
```

```
print(np.dot(v, w))
```

```
# Matrix / vector product; both produce the rank 1 array [29 67]
```

```
print(x.dot(v))
```

```
print(np.dot(x, v))
```

```
# Matrix / matrix product; both produce the rank 2 array
```

```
# [[19 22]
```

```
#  [43 50]]
```

```
print(x.dot(y))
```

```
print(np.dot(x, y))
```

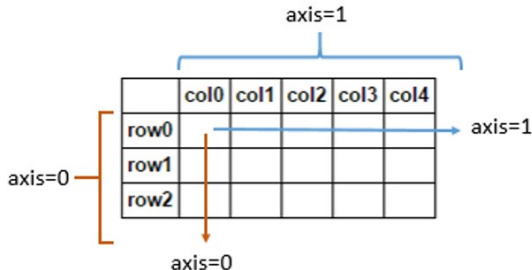
Numpy Functions

- Numpy provides many useful functions for performing computations on arrays, one of the most useful is sum.

```
import numpy as np
```

```
x = np.array([[1,2],[3,4]])
```

```
print(np.sum(x))           # Compute sum of all elements; prints "10"  
print(np.sum(x, axis=0))   # Compute sum of each column; prints "[4 6]"  
print(np.sum(x, axis=1))   # Compute sum of each row; prints "[3 7]"
```



Transpose

- Apart from computing mathematical functions using arrays, we frequently need to reshape or otherwise manipulate in arrays.
- The simplest example of this type of operation is **transposing a matrix (or two-dimensional array)**, to transpose a matrix, simply use the **T attribute of an array object**.

```
import numpy as np
```

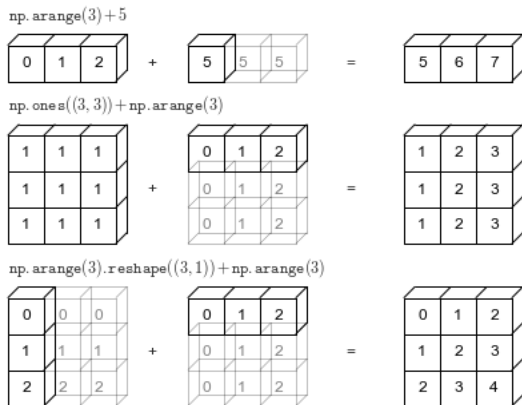
```
x = np.array([[1,2], [3,4]])  
print(x)      # Print "[[1 2]  
              #          [3 4]]"  
print(x.T)    # Print "[[1 3]  
              #          [2 4]]"
```

Note that taking the transpose of a rank 1 array does nothing:

```
v = np.array([1,2,3])  
print(v)      # Print "[1 2 3]"  
print(v.T)    # Print "[1 2 3]"
```

Broadcasting

- **Boardcasting** is a powerful mechanism that **allows numpy to work with arrays of different shapes** when performing arithmetic operations.
- Frequently, we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.



Broadcasting - Motivation

```
import numpy as np

# We will add the vector v to each row of the matrix x, storing the result in y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)    # Create an empty matrix with the same shape as x

# Add the vector v to each row of the matrix x with an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v

print(y)    # It prints "[[2 2 4]
            #           [5 5 7]
            #           [8 8 10]
            #           [11 11 13]]"
```

The above works, however when the matrix x is very large, computing an explicit loop could be **slow**.

Broadcasting - Motivation

- Note that adding the vector v to each row of the matrix x is equivalent to forming a matrix vv by stacking multiple copies of v vertically, then performing elementwise summation of x and vv . We could implement this as follows:

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1))    # Stack 4 copies of v on top of each other
print(vv)                  # Prints "[[1 0 1]
                            #          [1 0 1]
                            #          [1 0 1]
                            #          [1 0 1]]"

y = x + vv                  # Add x and vv elementwise

print(y)                    # Prints "[[2 2 4]
                            #          [5 5 7]
                            #          [8 8 10]
                            #          [11 11 13]]"
```

Broadcasting

- Numpy broadcasting allows us to perform this computation without actually creating multiple copies of v .
- Consider the following version, using broadcasting:

```
import numpy as np
```

```
# We will add the vector v to each row of the matrix x,  
# storing the result in the matrix y
```

```
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
```

```
v = np.array([1, 0, 1])
```

```
y = x + v # Add v to each row of x using broadcasting
```

```
print(y) # Prints "[[ 2  2  4]  
#           [ 5  5  7]  
#           [ 8  8 10]  
#           [11 11 13]]"
```

The line `y = x + v` works even though `x` has shape $(4, 3)$ and `v` has shape $(3,)$ due to broadcasting; this line works as if `v` actually had shape $(4, 3)$, where each row was a copy of `v`, and the sum was performed elementwise.

Broadcasting Rules

Broadcasting two arrays together follows these rules:

1. If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
2. The two arrays are said to be compatible in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
3. The arrays can be broadcast together if they are compatible in all dimensions.
4. After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
5. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension.

Understanding Broadcasting Rules

- Given two arrays `A = np.array([1, 2, 3])`, and `B = np.array([2])`. Can we perform `A * B`?

- Do they have the same rank? Yes, rank of A is 1, rank of B is 1.
- Are they compatible in all dimensions? Yes. Array B has size 1 in that dimension.
- So, they are compatible.

```
A = np.array([1, 2, 3])
print(A.ndim)      # Print 1
B = np.array([2])
print(B.ndim)      # Print 1
print(A * B)       # Print "[2, 4, 6]"
```

Understanding Broadcasting Rules

- Given two arrays $A = \text{np.array}([1, 2, 3])$, and $B = \text{np.array}([[4, 4, 4], [3, 3, 3]])$. Can we perform $A * B$?

- Do they have the same rank? No, they do not have the same rank, rank of A is 1, rank of B is 2. But we can prepend the shape of B (the lower rank array) with 1s until they have the same length.
- Are they compatible in all dimensions? Yes. Array A has size 1 in that dimension.
- So, they are compatible.

```
A = np.array([1, 2, 3])
print(A.ndim)    # Print 1
B = np.array([[4, 4, 4], [3, 3, 3]])
print(B.ndim)    # Print 2

print(A*B)       # Print [[4 8 12]
                  #         [3 6 9]]
```

Understanding Broadcasting Rules

How about the following?

1. Pair 1

- A: Dimensions - 5×4
- B: Dimension - 1

2. Pair 2

- A: Dimensions - 5×4
- B: Dimension - 4

3. Pair 3

- A: Dimensions - $15 \times 3 \times 5$
- B: Dimensions - $15 \times 1 \times 5$

4. Pair 4

- A: Dimensions - $15 \times 3 \times 5$
- B: Dimensions - 3×5

5. Pair 5

- A: Dimensions - $15 \times 3 \times 5$
- B: Dimensions - 3×1

1. Yes. Result - 5×4
2. Yes. Result - 5×4
3. Yes. Result - $15 \times 3 \times 5$
4. Yes. Result - $15 \times 3 \times 5$
5. Yes. Result - $15 \times 3 \times 5$

Why Numpy Arrays are Better?

- Numpy arrays **consume less memory** than Python List
- Numpy arrays are **fast** as compared to Python List
- Numpy arrays are **more convenient** to use



Memory Consumption Between Numpy Arrays and Lists

```
import numpy as np      # Import numpy package
import sys              # Import system module

L = range(1000)         # Declare a list of 1000 elements

# Print size of each element of the list
print("Size of each element of list in bytes: ", sys.getsizeof(L))  # Print 48
# Print size of the whole list
print("Size of the whole list in bytes: ", sys.getsizeof(L)*len(L)) # Print 48000

A = np.arange(1000)     # Declare a Numpy array of 1000 elements

# Print size of each element of the Numpy array
print("Size of each element of the Numpy array in bytes: ", A.itemsize) # Print 8
# Print size of the whole Numpy array
print("Size of the whole Numpy array in bytes: ", A.size*A.itemsize) # Print 8000
```


Time Comparison Between Numpy Arrays and Python Lists

```
import numpy as np          # Import required packages
import time as t

size = 1000000              # Size of arrays and lists

list1 = range(size)        # Declare lists
list2 = range(size)
array1 = np.arange(size)   # Declare arrays
array2 = np.arange(size)

# Capture time before the multiplication of Python lists
initialTime = t.time()
# Multiply elements of both the lists and stored in another list
resultList = [(a * b) for a, b in zip(list1, list2)]
# Calculate execution time, it prints "Time taken by Lists: 0.13024258613586426 s"
print("Time taken by Lists:", (t.time() - initialTime), "s")

# Capture time before the multiplication of Numpy arrays
initialTime = t.time()
# Multiply elements of both the Numpy arrays and stored in another Numpy array
resultArray = array1 * array2
# Calculate execution time, it prints "Time taken by NumPy Arrays: 0.006006956100463867 s"
print("Time taken by NumPy Arrays:", (t.time() - initialTime), "s")
```

Effect of Operations on Numpy Arrays and Python Lists

```
import numpy as np # Import Numpy package

ls = [1, 2, 3]      # Declare a list
arr = np.array(ls) # Convert the list into a Numpy array

try:
    ls = ls + 4      # Add 4 to each element of list
except(TypeError):
    print("Lists don't support list + int")

# Now on array
try:
    arr = arr + 4    # Add 4 to each element of Numpy array
    print("Modified Numpy array: ",arr) # Print the Numpy array
except(TypeError):
    print("Numpy arrays don't support list + int")
```

Output:

Lists don't support list + int

Modified Numpy array: [5 6 7]

Practice Problems (Print the result)

1. How to create array A of size 15, with all zeros?
2. How to find memory size of array A?
3. How to create array B with values ranging from 20 to 60?
4. How to create array C of reversed array of B?
5. How to create 4×4 array D with values from 0 to 15 (from top to bottom, left to right)?
6. How to find the dimensions of array E `[[3, 4, 5], [6, 7, 8]]`?
7. How to find indices for non-zero elements from array F `[0, 3, 0, 0, 4, 0]`?
8. How to create $3 \times 3 \times 3$ array G with random values?
9. How to find maximum values in array H `[1, 13, 0, 56, 71, 22]`?
10. How to find minimum values in array H?
11. How to find mean values of array H?
12. How to find standard deviation of array H?
13. How to find median in array H?
14. How to transpose array D?

Practice Problems (Print the results)

15. How to append array [4, 5, 6] to array I [1, 2, 3]?
16. How to memberwise add, subtract, multiply and divide two arrays J [1, 2, 3] and K [4, 5, 6]?
17. How to find the total sum of elements of array I?
18. How to find natural log of array I?
19. How to use build an array L with [8, 8, 8, 8, 8] using full/repeat function?
20. How to sort array M [2, 5, 7, 3, 6]?
21. How to find the indices of the maximum values in array M?
22. How to find the indices of the minimum values in array M?
23. How to find the indices of elements in array M that will be sorted?
24. How to find the inverse of array N = [[6, 1, 1], [4, -2, 5], [2, 8, 7]] in numpy?
25. How to find absolute value of array N?
26. How to extract the third column (from all rows) of the array O [[11, 22, 33], [44, 55, 66], [77, 88, 99]]?
27. How to extract the sub-array consisting of the odd rows and even columns of P [[3, 6, 9, 12], [15, 18, 21, 24], [27, 30, 33, 36], [39, 42, 45, 48], [51, 54, 57, 60]]?

Practice Problems (Answers)

1. `A = np.zeros(15); print(A)`
2. `print(A.size * A.itemsize)`
3. `B = np.arange(20, 61); print(B)`
4. `C = B[::-1]; print(C)`
5. `D = np.arange(16).reshape(4, 4); print(D)`
6. `E = np.array([[3, 4, 5], [6, 7, 8]]); print(E.shape)`
7. `F = np.array([0, 3, 0, 0, 4, 0]); print(F.nonzero())`
8. `G = np.random.random((3, 3, 3)); print(G)`
9. `H = np.array([1, 13, 0, 56, 71, 22]); print(H.max())`
10. `print(H.min())`
11. `print(H.mean())`
12. `print(H.std())`
13. `print(np.median(H))`
14. `print(np.transpose(D))`
15. `I = np.array([1, 2, 3]); I = np.append(I, [4, 5, 6]); print(I)`

Practice Problems (Answers)

16. `J = np.array([1, 2, 3]); K = np.array([4, 5, 6]);
print(J + K); print(J - K); print(J * K); print(J / K)`
17. `print(np.sum(I))`
18. `print(np.log(I))`
19. `L = np.full(5, 8); print(L)
L = np.repeat(8, 5); print(L)`
20. `M = np.array([2, 5, 7, 3, 6]); print(np.sort(M))`
21. `print(M.argmax())`
22. `print(M.argmin())`
23. `print(M.argsort())`
24. `N = np.array([[6, 1, 1], [4, -2, 5], [2, 8, 7]]); print(np.linalg.inv(N))`
25. `print(np.abs(N))`
26. `O = np.array([[11, 22, 33], [44, 55, 66], [77, 88, 99]])
print(O[:,2])`
27. `P = np.array([[3, 6, 9, 12], [15, 18, 21, 24], [27, 30, 33, 36], [51, 54, 57, 60]])
print(P[:,2, 1::2])`

That's all!

Any questions?

