
# Introduction to AI for postgraduate students

## Lecture Note 1: A *Brief* Guide on Python

Hyun Jong Yang (hyunyang@postech.ac.kr)

POSTECH

AiSLab

# References

**Python**

- The Python Tutorial — Python 3.9.7 documentation
- Python Tutorial (w3schools.com)

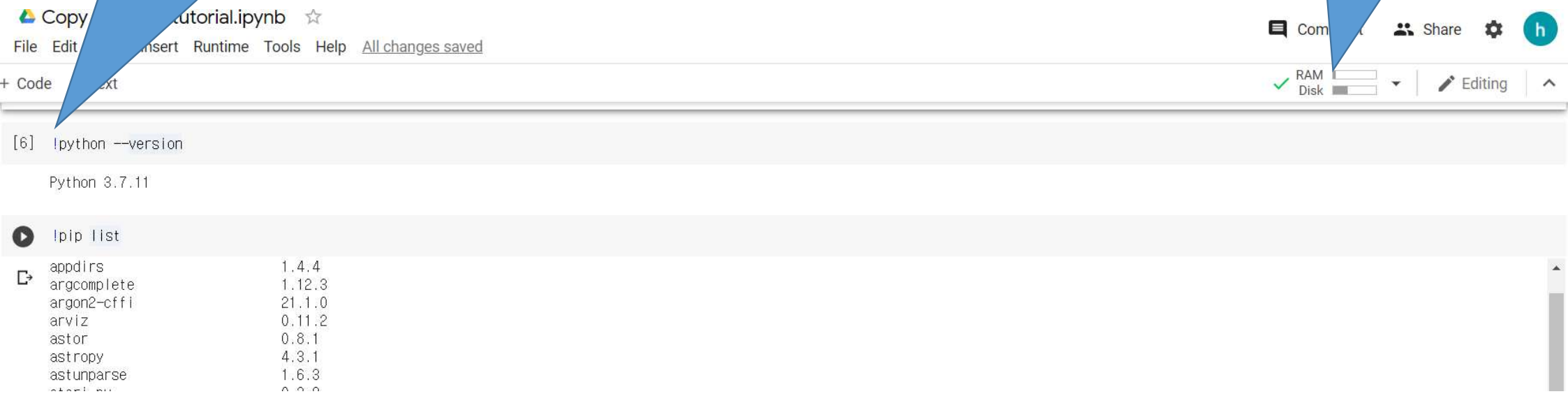
**Python with Colab**

- colab-tutorial.ipynb - Colaboratory (google.com)

# Check the version

"!" Mark: used to execute a Linux command

Your virtual machine assigned (free version: assigned only temporarily)



# Data Types: Numbers

```
[7]  x = 3
     print(x, type(x))
```

```
3 <class 'int'>
```

```
     print(x + 1)   # Addition
     print(x - 1)   # Subtraction
     print(x * 2)   # Multiplication
     print(x ** 2)  # Exponentiation
```

```
4
2
6
9
```

```
[9]  x += 1
     print(x)
     x *= 2
     print(x)
```

```
4
8
```

```
[10] y = 2.5
     print(type(y))
     print(y, y + 1, y * 2, y ** 2)
```

```
<class 'float'>
2.5 3.5 5.0 6.25
```

- Note that unlike many languages, Python does not have unary increment (x++) or decrement (x--) operators.
- Python also has builtin types for long integers and com plex numbers; you can find all of the details in https://docs.python.org/3.7/library/stdtypes.html#numeric-types-int-float-long-complex

# Data Type: Booleans

- **Python implements all of the usual operators for Boolean logic, but uses English words rather than symbols (&&, ||, etc.):**

```
[11] t, f = True, False
     print(type(t))
```

```
<class 'bool'>
```

Now we let's look at the operations:

```
[12] print(t and f) # Logical AND;
     print(t or f)  # Logical OR;
     print(not t)   # Logical NOT;
     print(t != f)  # Logical XOR;
```

```
False
True
False
True
```

# Data Type: Strings

```
hello = 'ok'   # String literals can use single quotes
world = "korea"   # or double quotes; it does not matter
print(hello, len(hello))
```

```
ok 2
```

```
[19] hw = hello + ' ' + world  # String concatenation
     print(hw)
```

```
ok korea
```

```
[20] hw12 = '{} {} {}'.format(hello, world, 12)  # string formatting
     print(hw12)
```

```
ok korea 12
```

```
[21] print(f"{hello} {world} 12")
```

```
ok korea 12
```

# Data Type: Strings

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

```
Hello
HELLO
  hello
 hello
he(ell)(ell)o
world
```

# Containers: Lists

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

```
xs = [3, 1, 2]   # Create a list
print(xs, xs[2])
print(xs[-1])     # Negative indices count from the end of the list; prints "2"
```

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

```
xs[2] = 'foo'    # Lists can contain elements of different types
print(xs)
```

```
[3, 1, 'foo']
```

```
xs.append('bar') # Add a new element to the end of the list
print(xs)
```

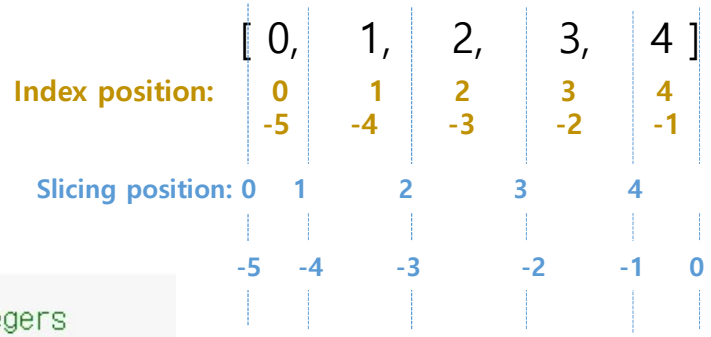
```
[3, 1, 'foo', 'bar']
```

```
x = xs.pop()     # Remove and return the last element of the list
print(x, xs)
```

```
bar [3, 1, 'foo']
```

| | [3, | 1 , | 2] |
|---|---|---|---|
| Index position: | 0<br>-3 | 1<br>-2 | 2<br>-1 |

# Containers: List Slicing




```
nums = list(range(5))    # range is a built-in function that creates a list of integers
print(nums)         # Prints "[0, 1, 2, 3, 4]"
print(nums[2:4])    # Get a slice from index 2 to 4 (exclusive); prints "[2, 3]"
print(nums[2:])     # Get a slice from index 2 to the end; prints "[2, 3, 4]"
print(nums[:2])     # Get a slice from the start to index 2 (exclusive); prints "[0, 1]"
print(nums[:])      # Get a slice of the whole list; prints ["0, 1, 2, 3, 4]"
print(nums[:-1])    # Slice indices can be negative; prints ["0, 1, 2, 3]"
nums[2:4] = [8, 9] # Assign a new sublist to a slice
print(nums)         # Prints "[0, 1, 8, 9, 4]"
```

```
[0, 1, 2, 3, 4]
[2, 3]
[2, 3, 4]
[0, 1]
[0, 1, 2, 3, 4]
[0, 1, 2, 3]
[0, 1, 8, 9, 4]
```

# Loops

```
[ ]  animals = ['cat', 'dog', 'monkey']
     for animal in animals:
         print(animal)
```

```
cat
dog
monkey
```

**0**, cat
**1**, dog
**2**, monkey

```
animals = ['cat', 'dog', 'monkey']
for idx, animal in enumerate(animals):
    print('#{}: {}'.format(idx + 1, animal))
```

```
#1: cat
#2: dog
#3: monkey
```

# List Comprehensions

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

```
[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)
```

```
[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)
```

```
[0, 4, 16]
```

# Container: 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; prints "True"
```

```
cute
True
```

```
[ ]  d['fish'] = 'wet'    # Set an entry in a dictionary
     print(d['fish'])      # Prints "wet"
```

```
wet
```

```
print(d['monkey'])  # KeyError: 'monkey' not a key of d
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-23-78fc9745d9cf> in <module>()
----> 1 print(d['monkey'])  # KeyError: 'monkey' not a key of d

NameError: name 'd' is not defined
```

SEARCH STACK OVERFLOW

# Container: Dictionaries

```
[ ]  print(d.get('monkey', 'N/A'))  # Get an element with a default; prints "N/A"
     print(d.get('fish', 'N/A'))    # Get an element with a default; prints "wet"
```

```
N/A
wet
```

```
[ ]  del d['fish']         # Remove an element from a dictionary
     print(d.get('fish', 'N/A')) # "fish" is no longer a key; prints "N/A"
```

```
N/A
```

# Container: Dictionaries

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

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

```
A person has 2 legs
A cat has 4 legs
A spider has 8 legs
```

Items(): return the list with all dictionary keys with values
"[('person', 2), ('cat', 4), ('spider', 8)]"

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)
```

```
{0: 0, 2: 4, 4: 16}
```

# Container: Sets

- **A set is an unordered collection of distinct elements.**

```
animals = {'cat', 'dog'}
print('cat' in animals)   # Check if an element is in a set; prints "True"
print('fish' in animals)  # prints "False"
```

```
True
False
```

```
animals.add('fish')      # Add an element to a set
print('fish' in animals)
print(len(animals))       # Number of elements in a set;
```

```
True
3
```

```
animals.add('cat')       # Adding an element that is already in the set does nothing
print(len(animals))
animals.remove('cat')    # Remove an element from a set
print(len(animals))
```

```
3
2
```

# Container: 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('#{}: {}'.format(idx + 1, animal))
```

```
#1: dog
#2: cat
#3: fish
```

# Container: Sets

- **Set comprehension**

```
[9]  from math import sqrt
     print([int(sqrt(x)) for x in range(30)])
     print({int(sqrt(x)) for x in range(30)})
```

```
[0, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5]
{0, 1, 2, 3, 4, 5}
```

# Container: Tuple

- **A tuple is an (immutable) ordered list of values. 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.**

```
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(d[t])
print(d[(1, 2)])
```

```
<class 'tuple'>
5
1
```

```
t[0] = 1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-43-c8aeb8cd20ae> in <module>()
----> 1 t[0] = 1

TypeError: 'tuple' object does not support item assignment

SEARCH STACK OVERFLOW
```

# Functions

Python functions are defined using the `def` keyword. For example:

```
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))
```

```
negative
zero
positive
```

We will often define functions to take optional keyword arguments, like this:

```
[ ] def hello(name, loud=False):
        if loud:
            print('HELLO, {}'.format(name.upper()))
        else:
            print('Hello, {}!'.format(name))

    hello('Bob')
    hello('Fred', loud=True)
```

```
Hello, Bob!
HELLO, FRED
```

# Classes

```
class Greeter:

    # Constructor
    def __init__(self, name):
        self.name = name  # Create an instance variable

    # Instance method
    def greet(self, loud=False):
        if loud:
          print('HELLO, {}'.format(self.name.upper()))
        else:
          print('Hello, {}!'.format(self.name))

g = Greeter('Fred')  # Construct an instance of the Greeter class
print(g.name)
g.greet()            # Call an instance method; prints "Hello, Fred"
g.greet(loud=True)   # Call an instance method; prints "HELLO, FRED!"
```

```
Fred
Hello, Fred!
HELLO, FRED
```

The **__init__() function** is called automatically every time the class is being used to create a new object.

Property

Method

# Classes

- **Properties**

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

John
36

# Class: Inheritance

- **Parent class**

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def description(self):
      print(self.firstname, self.lastname)

TT = Person('John', 'Doe')
TT.description()
```

```
John Doe
```

- **Child class**

```
class Student(Person):
  pass
```

# Class: Inheritance

- **Add the __init__() Function**
  - When you add the __init__() function, the child class will no longer inherit the parent's __init__() function.

```
class Student(Person):
  def __init__(self, fname, lname):
    self.firstname = "N/A"
    self.lastname = "N/A"

TT = Student('John', 'Doe')
TT.description()
```

```
N/A N/A
```

- **To keep the inheritance of the parent's __init__() function, add a call to the parent's __init__() function:**

```
class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self,fname, lname)

TT = Student('John', 'Doe')
TT.description()
```

```
John Doe
```

# Class: Inheritance

- **Alternative way with the addition of properties and methods**

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def welcome(self):
    print('Welcom',self.firstname, self.lastname, 'to the class of', self.graduationyear)

TT = Student('John', 'Doe', 2000)
TT.welcome()
```

```
Welcom John Doe to the class of 2000
```

Inheritance of the parent's __init__() function using "super"

Addition of method

Addition of properties