

week_1

February 28, 2019

1 Environment and installation

- Python
Python is an interpreted, high-level, general-purpose programming language.
download link www.python.org/downloads/
Latest version 3.7.2
- Anaconda
Anaconda is a free and open-source distribution of the python and R programming languages for scientific computing (data science, machine learning application, large-scale data processing, predictive analytics, etc.), that aims to simplify package management and deployment. Package version are managed by the package management system conda.
download link www.anaconda.com/download/
Latest version python3.7 version
- Pytorch
Pytorch is an open-source machine learning library for Python, based on Torch, used for application such as natural language processing.
download link pytorch.org
Latest version 0.4.1

2 Usage of gpu server

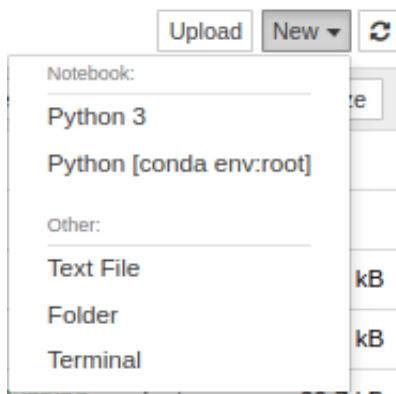
- Server address: <http://222.200.180.185:25999>
- Register an account with the student number as the username
- Please keep your account and password carefully.




3 Usage of jupyter

- Create a new folder as the root directory of jupyter
- Open your terminal, then enter command `jupyter notebook`, then enter the home directory
- Manage your files in Files , and manage your running code or running terminal in Running



- Create a python file by clicking New button and clicking Python 3



- Create a cell by clicking 
- Delete a cell by clicking 
- ctrl+enter to run your code or click 
- Esc+Y to enter code mode
- Esc+M to enter markdown mode

4 Learn python in a short time

In order to learn pytorch, we should have a general understanding of python

4.1 Properties

Python is strongly typed (i.e. types are enforced), dynamically, implicitly typed (i.e. you don't have to declare variables), case sensitive (i.e. var and VAR are different variables) and object-oriented (i.e. everything is an object).

4.2 Getting help

Help in Python is always available right in the interpreter. If you want to know how an object works, all you need to do is to call `help(<object>)`! Also `dir()` are useful, which shows you all the object's methods and `<object>.doc`.

```
In [ ]: # show the usage of the object 'int'
        help(int)
        #help(1)
```

```
In [ ]: # show the methods of int
        dir(int)
```

```
In [ ]: # show the documents of int
        int.__doc__
```

4.3 Syntax

Comments start with the pound (#) sign and are single-line. Values are assigned (in fact, objects are bound to names) with the equals sign ("="), and equality testing is done using two equals signs ("=="). You can increment/decrement values using the += and -= operators respectively by the right-hand amount. This works on many datatypes, strings included. You can also use multiple variables on one line. For example:

```
In [4]: # assign a value to variable
        x = 1

In [ ]: # x = x + 2
        x += 2
        print(x)

In [ ]: # x = x - 1
        x -= 1
        print(x)

In [ ]: # concatenate two string by '+'
        y = 'Hello '
        y1 = 'world'
        print(y + y1)

In [ ]: # Assign two values to two variable respectively
        z1, z2 = 1, 'str'
        print(z1, z2)

In [ ]: '''
        swap z1 and z2
        temp = z1
        z1 = z2
        z2 = temp
        '''

        z1, z2 = z2, z1
        print(z1, z2)
```

4.4 Data types

The data structures available in python are lists, tuples and dictionaries. The index of the first item in all array types is 0. Negative numbers count from the end towards the beginning, -1 is the last item. Variables can point to functions. The usage is as follows:

```
In [ ]: # list is a bit like array we have learned in c or c++
        list_sample = [1,2]
        print(list_sample)
        list_sample1 = ['hello', 'world']
        print(list_sample1)
```

```

In [ ]: # tuple is a bit like list, but its element can't be changed
tuple_sample = (1, 2, 3)
print(tuple_sample)
tuple_sample1 = ('hello', 'world')
print(tuple_sample1)

#try to change the element of tuple
# tuple_sample1[0] = 'hey'
# print(tuple_sample1)

In [ ]: # dictionary = {key:value}
dict_sample = {'key1': 'value1', 2:3, 'tuple': (1,2)}
# get value by key
print(dict_sample['key1'])
print(dict_sample['tuple'][0])

In [ ]: # a list can contain any type of data
mix_sample = [5, 'hello world', [1, 'alex', (1,2,3)], ('my', 'tuple')]
print(mix_sample[0])
#index -1 refers the last element
print(mix_sample[-1])

In [14]: '''
variables point to functions
type() is a built-in function
help(type)
'''

function_sample = type

In [ ]: # show the type of list_sample
print(function_sample(list_sample))
# equivalent to the above line
print(type(list_sample))

In [16]: list_sample2 = [1, 'a', 2, 'b', 3, 'c']

In [ ]: # only access the first element
first, *_ = list_sample2
print(first)

In [ ]: # only access the last element
*_, last = list_sample2
print(last)

In [ ]: #only access the second element
_, second, *_ = list_sample2
print(second)

```

You can access array ranges using a colon (:). Leaving the start index empty assumes the first item, leaving the end index assumes the last item. Negative indexes count from the last item backwards (thus -1 is the last item) like so:

```
In [ ]: # access values using index
list1 = [1, 2, 3, 4, 5, 6]
# array[start:end] access elements from start to end but not contain end
# access all elements
print(list1[:])
# access elements from zeroth to fifth
print(list1[0: 6])
print(list1[0: 2])
print(list1[-3:-1])
```

4.5 Strings

Python's strings can use either single or double quotation marks, and you can have quotation marks of one kind inside a string that uses the other kind (i.e. "He said 'hello'." is valid). Multiline strings are enclosed in triple double (or single) quotes ("""). To fill a string with values, you use the % (modulo) operator and a tuple. Each %s gets replaced with an item from the tuple, left to right, and you can also use dictionary substitutions, like so:

```
In [ ]: # %s will be replaced by variable in %()
print('Name: %s' % ('alex'))

# another usage
print('Name: %(name)s' % {'name': 'alex'})

# {} will be replaced by parameters in .format()
print('Name: {}'.format('alex'))

In [ ]: #print a multi-line string
multi_string = '''this is a
multiline
string'''
print(multi_string)

In [ ]: print('Name: %s\nNumber: %d\nString: %s\n' % ('alex', 1, 4 * '-'))

print('this %(value1)s a %(value2)s\n' % {'value2': 'test', 'value1': 'is'})

message = 'let\'s learn {} in a {}'.format('python', 'short time')
print(message)
```

4.6 Flow control statements

Flow control statements are if, for, and while. Python has no mandatory statement termination characters and blocks are specified by indentation. Indent to begin a block, cancel indentation to end one. Statements that expect an indentation level end in a colon (:). These statements' syntax is thus:

```
In [ ]: # generate a list which has elements from zero to nine
range_list = range(10)
```

```

'''
it will be written if we use c or c++
for(i=0; i <10; i++){
    range_list[i] = i;
}
'''
print(range_list)

for i in range_list:
    print(i,end=' ')

```

```

In [25]: # format of the for loop
# for variable_name in iterable_object:
for number in range_list:
    # Check if number is one of
    # the numbers in the tuple.
    if number in (3, 4, 7, 9):
        # "Break" terminates a for without
        # executing the "else" clause.
        break
    else:
        # "Continue" starts the next iteration
        # of the loop. It's rather useless here,
        # as it's the last statement of the loop.
        continue
else:
    # The "else" clause is optional and is
    # executed only if the loop didn't "break".
    pass # Do nothing

```

```

In [ ]: if range_list[1] == 2:
        print("The second item (lists are 0-based) is 2")
elif range_list[1] == 3:
    print("The second item (lists are 0-based) is 3")
else:
    print("\nDunno")

```

```

In [27]: # Format of the while loop
while range_list[0] == 1:
    pass

```

```

In [ ]: # add odd and even respectively
odd = 0
even = 0
for number in range(100):
    if number % 2 == 0:
        even += number
    else:

```

```

        odd += number
    print('the total of odd:',odd)
    print('the total of even',even)

```

4.7 Functions

Functions are declared with the `def` keyword. Optional arguments are set in the function declaration after the mandatory arguments by being assigned a default value. For named arguments, the name of the argument is assigned a value. Functions can return a tuple (and using tuple unpacking you can effectively return multiple values). Lambda functions are ad hoc functions that are comprised of a single statement. Parameters are passed by reference, but immutable types (tuples, ints, strings, etc) **cannot be changed**. This is because only the memory location of the item is passed, and binding another object to a variable discards the old one, so immutable types are replaced. For example:

```

In [29]: '''
           Comment the function

           def function_name(args):
               \'''
               coments
               \'''
               pass
           '''

           def passing_example(a_list, an_int = 2, a_string = " A default string"):
               '''a_list: ****
                  an_int: ****
                  a_string: ****
               '''
               a_list.append("A new item")
               an_int = 4
               return a_list, an_int, a_string

In [ ]: list1 = [0,1,2]
        int1 = 10
        print(passing_example(list1,int1))

In [ ]: #show method usage
        help(passing_example)

In [ ]: # list1 will be changed
        print(list1)
        # int1 will not be changed
        print(int1)

In [33]: #swap function
        a, b =1, 2

        def swap(a,b):

```

```

t = a
a = b
b = t
return a, b

```

```

In [ ]: a, b = swap(a,b)
        print(a,b)

```

```

In [35]: #keyword parameter, its order is not important
        def print_info(name, age=35):
            print('name:', name, 'age:', age)

```

```

In [ ]: print_info(age=50, name='miko')

```

```

In [37]: #indefinite length parameter
        #you may need a function that can handle more parameters than when you declared it.
        def print_info(arg1, *args):
            print(arg1, end=' ')
            for x in args:
                print(x, end=' ')
            print('')

```

```

In [ ]: # pass one parameter
        print_info(10)
        # pass two parameter
        print_info(10,20)
        # pass three parameter
        print_info(10,20,30)

```

```

In [39]: #global variable and local variable
        total = 0

```

```

        def sum1(arg1, arg2):
            total = arg1 + arg2
            print(total)

        def sum2(arg1, arg2):
            # refer global variable
            global total
            total = arg1 + arg2
            print(total)

```

```

In [ ]: sum1(10,20)
        # total will not be changed
        print(total)

```

```

In [ ]: sum2(10,20)
        # total will be changed
        print(total)

```



```
In [ ]: #anonymous function
        # the same as def function(x): return x+1
        functionvar = lambda x: x + 1          #lambda [arg1 [,arg2,...,argn]]:expression
        print(functionvar(1))
```

4.8 Classes

Python supports a limited form of multiple inheritance in classes. Private variables and methods can be declared (by convention, this is not enforced by the language) by adding at least two leading underscores and at most one trailing one (e.g. “__spam”). We can also bind arbitrary names to class instances. An example follows:

```
In [43]: # Declaring a simple class
```

```
class People:
    """
    Base class. It has two attributes: name and age,
    and two methods: get_name and get_age to obtain attribute value.
    When declaring classes, you must rewrite __init__ method to initialize class for
    """

    def __init__(self, name, age):
        """initialization"""
        # initialization for class's attribute, use self.xx
        self.name = name
        # initialization for class's attribute, use self.xx
        self.age = age

    def get_age(self):
        # return self.age's value
        return self.age

    def get_name(self):
        # return self.name's value
        return self.name
```

```
In [ ]: # Obtain a People object and use its method
```

```
people = People(name='xiaoming', age=21) # obtain a People object
print(people)

# using class's method to get its attribute value
print('age :', people.get_age())
print('name :', people.get_name())

# directly get its value by access its attribute
print('age :', people.age)
print('name :', people.name)
```

```
In [45]: # Declaring a class to inherit People class
```

```
class Student(People):
    """
    Student inherit People class.
    Thus, Student also have attribute like name and methods like get_age and get_name
    In student class, we add one more attribute idx for student class, as well as a n
    """

    def __init__(self, name, age, idx):
        # initialize name and age using People's initialization's method
        super(Student, self).__init__(name, age)
        # initialization for class's attribute
        self.idx = idx

    def get_idx(self):
        """
        add a new method for Student to obtain idx
        """
        return self.idx
```

```
In [ ]: # Obtain a Student object and use its method
```

```
student = Student('xiaoming', 21, 12)

print(student)

# using class's method to get its attribute value
print('age :', student.get_age())
print('name :', student.get_name())
# new method only owned by Student class
print('idx :', student.get_idx())

# directly get its value by access its attribute
print('age :', student.age)
print('name :', student.name)
print('idx :', student.idx)
```

4.9 File I/O

We could use open to open a file and do some complex operation like, opening for reading only, opening with text mode. You can see more about open in [python open's method document](#). Here, we just introduce open about its writing mode and reading mode.

```
In [47]: # Writing a file
```

```
# 'w' => writing mode
# if there is no file named 'example.txt', it would be created and we could write som
```

```

# if there is a file called 'example.txt', its content would be erased and we could w

# open(rootpath, mode), see more from document
with open("./example.txt", 'w') as f:
    # write 'hello world!' to files first line
    f.writelines('hello world!')
    # write 'Bye bye!' to files second line
    f.writelines('Bye bye!')

In [ ]: # Reading a file, must exists first

# 'r' => reading only mode
with open('./example.txt', 'r') as f:
    print(f.readlines()) # read all content at one time

```

4.10 Miscellaneous

```

In [49]: list1 = [1, 2, 3]
        list2 = [3, 4, 5]

In [ ]: # Calculate the product of any elements from list1 and list2

# [1*3, 1*4, 1*5, 2*3, 2*4, 2*5, 3*3, 3*4, 3*5]
print([x * y for x in list1 for y in list2])

In [ ]: # Select elements > 1 and < 4 for list1

# [2, 3]
print([x for x in list1 if 1 < x < 4])

In [ ]: # if there exists a element that could be divided exactly by 3, return true

# True
print(any(i % 3 for i in range(10)))

In [ ]: # account how many 4 in a list

print(sum(1 for i in [2, 2, 3, 3, 4, 4] if i == 4))

```

5 Classwork:

5.1 Programming:

5.1.1 Question1:

- realize several of activation functions(Sigmoid, tanh, ReLU, Leaky ReLU, ELU, **request: those functions must be contained in a class**, point: you can import math module, then use `math.exp()` function)