UIC
BUSINESS

Business Data Mining

IDS 472 (Spring 2024)

Instructor: Wenxin Zhou

Getting Started with Python

UIC BUSINESS

- Simplicity & Readability (along with many 3rd-party packages) have made Python the "go-to" programming language for ML
- Step 1: Install Anaconda
  - Anaconda is a data science platform that comes with many things that are required throughout the class
  - It comes with a Python distribution, a package manager known as conda, and many DS packages and libraries (NumPy, pandas, SciPy, scikit-learn, and Jupyter Lab/Notebook)
  - To install Anaconda, refer to Python_Setup.pdf on Blackboard

Jupyter Notebook/Lab

UIC
BUSINESS

- Python is a programming language that is interpreted rather than complied [ChatGPT]
  - We can run codes line-by-line
  - The IDE (integrated development environment) of choice in this class is Jupyter lab or Jupyter notebook
  - Using Jupyter Lab allows us to write and run codes, and combine them with text and graphics
  - Once Anaconda is installed, we can launch Jupyter lab either from Anaconda Navigator panel or from terminal
  - All materials for this class are developed in Jupyter lab

Variables

UIC BUSINESS

- In a Jupyter notebook, everything is part of *cells*
- Code and markdown (text) cells are the most common cells

```
2 + 5
```

```
7
```

```
x = 2.2     # this is a comment (use "#" for comments)
y = 2
x * y
```

```
4.4
```

```
x = 1         # an integer
x = 0.3       # a floating-point
x = 'what a nice day!' # a string
x = True     # a boolean variable (True or False)
x = None     # None type (the absence of any value)
```

Variables

UIC BUSINESS

- Python variables are references!
  - An assignment statement such as `x = 1` creates a reference `x` to a memory location storing object 1
  - Types are attached to the objects on the right, not to the variable name on the left
  - We can see the type of a variable by `type()` method

```
x = 3.3
display(type(x))
x = True
display(type(x))
x = None
display(type(x))
```

Strings

UIC BUSINESS

- A string is a sequence of characters

```
string1 = 'This is a string'
print(string1)
string2 = "Well, this is 'string' too!"
print(string2)
string3 = 'Johnny said: "How are you?"'
print(string3)
```

- Concatenating strings can be done using (+) operator

```
string3 = string1 + ". " + string2
print(string3)
```

```
This is a string. Well, this is 'string' too!
```

- Use `\t` and `\n` to add tab and newline characters to a string

```
print("Here we use a newline\nto go to the next\t line")
```

```
Here we use a newline
to go to the next	 line
```

Some Important Operators

UIC BUSINESS

- The following expressions include arithmetic operators in Python:

```
x = 5
y = 2
print(x + y)   # addition
print(x - y)   # subtraction
print(x * y)   # multiplication
print(x / y)   # dividion
print(x // y)  # floor division (removing fractional parts)
print(x % y)   # modulus (integer remanider of division)
print(x ** y)  # x to the power of y
```

- The following expressions include relational and logical operators:

```
print(x < 5)   # less than (> is greater than)
print(x <= 5)  # less than or equal to (>= is greater than or equal to)
print(x == 5)  # equal to
print(x != 5)  # not equal to
print((x > 4) and (y < 3)) # "and" keyword is used for logical and (it␣
↪is highlighted to be distinguished from other texts)
print((x < 4) or (y > 3))  # "or" keyword is used for logical or
print(not (x > 4))         # "not" keyword is used for logical not
```

Membership Operators

UIC BUSINESS

- Membership operator is used to check whether an element is present within a collection of data item
- By collection, we refer to ordered or unordered data structures such as string, lists, sets, tuples & dictionaries

```
print('Hello' in 'HelloWorlds!') # 'HelloWorlds!' is a string and 'Hello'␣
↪is part of that
print('Hello' in 'HellOWorlds!')
print(320 in ['Hi', 320, False, 'Hello'])
```

```
True
False
True
```

Lists

UIC BUSINESS

- Python has a number of built-in data structures that are used to store multiple data items as separate entries
- The most basic collection is a *list*, which is used to store a *sequence* of objects (ordered)
- It is created by a sequence of comma-separated objects within [ ]

```
x = [5, 3.0, 10, 200.2]
x[0]  # the index starts from 0
```

```
x = ['JupytherNB', 75, None, True, [34, False], 2, 75] # observe that␣
↪this list contains another list
x[4]
```

Lists

- Lists are *mutable*: they can be modified after they are created

```
x[4] = 52 # here we change one element of list x
x
```

```
['JupytherNB', 75, None, True, 52, 2, 75]
```

- We can use a number of functions and methods with a list:

```
len(x) # here we use a built-in function to return the length of a list
```

```
y = [9, 0, 4, 2]
print(x + y) # to concatenate two lists, + operator is used
print(y * 3) # to concatenate multiple copies of the same list, *␣
↪operator is used
```

```
z = [y, x] # to nest lists to create another list
z
```

Lists

UIC BUSINESS

- We can use *indexing* to access an element within a list

```
x[3]
```

```
True
```

- To access the elements of nested lists (list of lists), we need to separate indices with square brackets

```
z[1][0] # this way we access the second element within z and within
 ↪that we access the first element
```

```
'JupytherNB'
```

- A negative index has a meaning

```
x[-1] # index -1 returns the last item in the list; -2 returns the
 ↪second item from the end, and so forth
```

Lists

UIC BUSINESS

- *Slicing* is used to access multiple elements in the form of a sub-list: use a colon (:) to specify the start point (inclusive) and end point (non-inclusive)

```
x[0:4] # the last element seen in the output is at index 3
```

```
x[:4] # equivalent to x[0:4]
```

```
['JupytherNB', 75, None, True]
```

```
x[4:]
```

```
[52, 2, 75]
```

```
print(x)
x[-2:]
```

```
['JupytherNB', 75, None, True, 52, 2, 75]
```

```
[2, 75]
```

Lists

UIC BUSINESS

- Another useful type of slicing is using [start:stop:stride] syntax, where the stride is the step size

```
x[0:4:2] # steps of 2
```

```
['JupytherNB', None]
```

```
x[4::-2] # a negative step returns items in reverse (it works backward
 ↪so here we start from the element at index 4 and go backward to the
 ↪beginning with steps of 2)
```

```
[52, None, 'JupytherNB']
```

- Modifying elements in a list

```
x.append(-23) # to append a value to the end of the list
x
```

```
['JupytherNB', 75, None, True, 52, 2, 75, -23]
```

Lists

UIC BUSINESS

```
x.remove(75) # to remove the first matching element
x
```

```
['JupytherNB', None, True, 52, 2, 75, -23]
```

```
y.sort() # to sort the element of y
y
```

```
[0, 2, 4, 9]
```

```
x.insert(2, 10) # insert(pos, elmnt) method inserts the specified elmnt␣
↪at the specified position (pos) and shift the rest to the right
x
```

```
['JupytherNB', None, 10, True, 52, 2, 75, -23]
```

```
print(x.pop(3)) # pop(pos) method removes (and returns) the element at␣
↪the specified position (pos)
x
```

```
True
['JupytherNB', None, 10, 52, 2, 75, -23]
```

Lists

UIC BUSINESS

```
del x[1] # del statement can also be used to delete an element from a␣
↪list by its index
x
```

```
['JupytherNB', 10, 52, 2, 75, -23]
```

```
x.pop() # by default the position is -1, which means that it removes␣
↪the last element
x
```

```
['JupytherNB', 10, 52, 2, 75]
```

- Copying a list: it is often desired to make a copy of a list and work with it without affecting the original list
- If we simply use the assignment operator, we end up changing the original list!

Lists

UIC BUSINESS

```python
list1 = ['A+', 'A', 'B', 'C+']
```

```python
list2 = list1
list2
```

```
['A+', 'A', 'B', 'C+']
```

```python
list2.append('D')
print(list2)
print(list1)
```

```
['A+', 'A', 'B', 'C+', 'D']
['A+', 'A', 'B', 'C+', 'D']
```

- When we write `list2 = list1`, what happens internally is that variable `list2` will point to the same container as `list1`

Lists

UIC BUSINESS

- There are three simple ways to properly copy the elements of a list
  1) *slicing*
  2) `copy()` method
  3) `list()` constructor

- They all create *shallow* copies of a list (in contrast with *deep* copies)

A shallow copy of a compound object such as list creates a new compound object and then adds references (to the objects found in the original object) into it. A deep copy of a compound object creates a new compound object and then adds *copies* of the objects found in the original object.

Lists

UIC BUSINESS

```
list3 = list1[:] # the use of slicing; that is, using [:] we make a
 ↪shallow copy of the entire list1
list3.append('E')
print(list3)
print(list1)
```

```
list4 = list1.copy() # the use of copy() method
list4.append('E')
print(list4)
print(list1)
```

```
list5 = list(list1) #the use of list() constructor
list5.append('E')
print(list5)
print(list1)
```

Tuples

UIC BUSINESS

- Tuple is another data-structure that can hold other arbitrary data types
- A tuple is *immutable*: once it's created, its size and contents cannot be changed

```
tuple1 = ('Machine', 'Learning', 'with', 'Python', '1.0.0')
tuple1
```

```
('Machine', 'Learning', 'with', 'Python', '1.0.0')
```

```
tuple1[0]
```

```
'Machine'
```

```
tuple1[::2]
```

```
('Machine', 'with', '1.0.0')
```

Tuples

UIC BUSINESS

```
len(tuple1) # the use of len() to return the length of tuple
```

- An error is raised if we try to change the content of a tuple

```
tuple1[0] = 'Jupyter' # Python does not permit changing the value
```

```
TypeError                                 Traceback (most recent call␣
 ↪last)
/var/folders/vy/894wbsn11db_lqf17ys9fvdm0000gn/T/ipykernel_51384/
 ↪877039090.py in <module>
----> 1 tuple1[0] = 'Jupyter' # Python does not allow us to change the␣
 ↪value

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

- There is no `append` or `remove` method for tuples

Tuples

UIC BUSINESS

- We could redefine the entire tuple

```
tuple1 = ('Jupyter', 'NoteBook') # redefine tuple1
tuple1
```

```
('Jupyter', 'NoteBook')
```

- We can concatenate them to create new tuples

```
tuple2 = tuple1 + ('Good', 'Morning')
tuple2
```

```
('Jupyter', 'NoteBook', 'Good', 'Morning')
```

- A common use of tuples is in functions that return multiple values
  - `modf()` function from math module returns a two-item tuple including the fractional and integer parts of its input

Tuples

UIC BUSINESS

```
from math import modf   # more on "import" later. For now just read this as "from math module, import modf function" so that modf function is available in our program

a = 56.5
modf(a) # the function is returning a two-element tuple
```

```
(0.5, 56.0)
```

- Sequence unpacking:

```
x, y = modf(a)
print("x = " + str(x) + "\n" + "y = " + str(y))
```

```
x = 0.5
y = 56.0
```

Tuples

UIC BUSINESS

- A sequence of comma separated objects without paratheses is packed into a tuple

```
tuple1 = 'Machine', 'Learning', 'with', 'Python', '1.0.0' # sequence␣
 ↪packing
tuple1
```

```
('Machine', 'Learning', 'with', 'Python', '1.0.0')
```

```
x, y, z, v, w = tuple1 # the use of sequence unpacking
print(x, y, z, v, w)
```

```
Machine Learning with Python 1.0.0
```

Tuples

UIC BUSINESS

- Multiple assignment and unpacking with lists

```
x, y, z, v, w = 'Machine', 'Learning', 'with', 'Python', '1.0.0'
print(x, y, z, v, w)
```

```
Machine Learning with Python 1.0.0
```

```
list6 = ['Machine', 'Learning', 'with', 'Python', '1.0.0']
x, y, z, v, w = list6
print(x, y, z, v, w)
```

```
Machine Learning with Python 1.0.0
```

- To create a one-element tuple, the comma is required

```
tuple3 = 'Machine', # remove the comma and see what would be the type␣
 ↪here
type(tuple3)
```

```
tuple
```

Dictionaries

UIC BUSINESS

- A dictionary is a useful data structure that contains a set of *values*, where each value is labeled by a unique *key*
- Dictionaries are created using a collection of key:value pairs wrapped within { } and are non-ordered

```
dict1 = {1:'value for key 1', 'key for value 2':2, (1,0):True, False:
 ↪[100,50], 2.5:'Hello'}
dict1
```

```
{1: 'value for key 1',
 'key for value 2': 2,
 (1, 0): True,
 False: [100, 50],
 2.5: 'Hello'}
```

```
dict1['key for value 2']
```

```
2
```

Dictionaries

UIC BUSINESS

```
dict1['key for value 2'] = 30 # change an element
dict1
```

```
{1: 'value for key 1',
 'key for value 2': 30,
 (1, 0): True,
 False: [100, 50],
 2.5: 'Hello'}
```

```
dict1[10] = 'Bye'
dict1
```

```
{1: 'value for key 1',
 'key for value 2': 30,
 (1, 0): True,
 False: [100, 50],
 2.5: 'Hello',
 10: 'Bye'}
```

Dictionaries

UIC BUSINESS

- `del` statement can be used to remove a key:value pair

```
del dict1['key for value 2']
dict1
```

```
{1: 'value for key 1',
 (1, 0): True,
 False: [100, 50],
 2.5: 'Hello',
 10: 'Bye'}
```

- A key can not be a mutable object such as list

```
dict1[['1','(1,0)']] = 100 # list is not allowed as the key
```

```
TypeError                                 Traceback (most recent call␣
 ↪last)
/var/folders/vy/894wbsn11db_lqf17ys9fvdm0000gn/T/ipykernel_71178/
 ↪1077749807.py in <module>
----> 1 dict1[['1','(1,0)']] = 100 # list is not allowed as the key

TypeError: unhashable type: 'list'
```

Dictionaries

- To check the membership among keys, we use the `keys()` method to return a `dict_keys` object

```
(1,0) in dict1.keys()
```

```
True
```

```
(1,0) in dict1 # equivalent to: in dict1.keys()
```

- To check the membership among values, use the `values()` method to return a `dict_values` object

```
"Hello" in dict1.values()
```

```
True
```

Dictionaries

UIC BUSINESS

- Another common way to create a dictionary is to use the `dict()` constructor

```
dict2 = dict([('Police', 102), ('Fire', 101), ('Gas', 104)])
dict2
```

```
{'Police': 102, 'Fire': 101, 'Gas': 104}
```

```
dict3 = dict(Country='USA', phone_numbers=dict2, population_million=18.
↪7) # the use of keywords arguments = object
dict3
```

```
{'Country': 'USA',
 'phone_numbers': {'Police': 102, 'Fire': 101, 'Gas': 104},
 'population_million': 18.7}
```

Sets

UIC BUSINESS

- Sets are collections of non-ordered unique and immutable objects

```
set1 = {'a', 'b', 'c', 'd', 'e'}
set1
```

```
set2 = {'b', 'b', 'c', 'f', 'g'}
set2 # observe that the duplicate entery is removed
```

- They support union, intersection, difference, and symmetric difference

```
set1 | set2 # union using an operator. Equivalently, this could be done␣
↪by set1.union(set2)
```

```
{'a', 'b', 'c', 'd', 'e', 'f', 'g'}
```

```
set1 & set2 # intersection using an operator. Equivalently, this could␣
↪be done by set1.intersection(set2)
```

```
{'b', 'c'}
```

Sets

UIC BUSINESS

```
set1 ^ set2 # symmetric difference: elements only in one set, not in
 ↪both. Equivalently, this could be done by set1.
 ↪symmetric_difference(set2)
```

```
{'a', 'd', 'e', 'f', 'g'}
```

```
'b' in set1 # check membership
```

```
True
```

- We can use `help()` to see a list of all available set operations

```
help(set1) # output not shown
```

Sequence Unpacking

UIC BUSINESS

- In the following example, variable `y` becomes a list of ‘Learning’ and ‘with’

```
x, *y, v, w = ['Machine', 'Learning', 'with', 'Python', '1.0.0']
print(x, y, v, w)
```

```
Machine ['Learning', 'with'] Python 1.0.0
```

- Here * is working as an operator of implement *extended iterable unpacking*
- Any list or tuple is an iterable object
- We may use * right before an iterable in which case the iterable is expanded into a sequence of items

Sequence Unpacking

UIC BUSINESS

- Here is an example in which * operates on an iterable (a list), but at the site of unpacking, we create a tuple

```
*[1,2,3], 5
```

```
(1, 2, 3, 5)
```

- Here is a similar example: the 2nd iterable is a list

```
[*[1,2,3], 5]
```

```
[1, 2, 3, 5]
```

- Create a set

```
{*[1,2,3], 5}
```

```
{1, 2, 3, 5}
```

Sequence Unpacking

UIC BUSINESS

- The following example raises an error

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

```
  File "/var/folders/vy/894wbsn11db_lqf17ys9fvdm0000gn/T/ipykernel_7117
↪386627056.py", line 1
    *[1,2,3]
    ^
SyntaxError: can't use starred expression here
```

- Iterable unpacking can be only used in certain places
- It can be used inside a list, tuple, or set
- It can be also used in *list comprehension* (discussed later) and inside function definitions and calls

Extended Iterable Unpacking

UIC BUSINESS

- * is a "catch-all" operator

```
x, *y, v, w = ('Machine', 'Learning', 'with', 'Python', '1.0.0')
print(x, y, v, w)
```

```
Machine ['Learning', 'with'] Python 1.0.0
```

- To create an output as before, use again * before `y` in the print function

```
print(x, *y, v, w)
```

```
Machine Learning with Python 1.0.0
```

for Loops

UIC BUSINESS

- `for` loop statement allows us to loop over any *iterable* object
  - an iterable is any object capable of returning its members one at a time
  - list, string, tuple, sets, dictionaries are iterable objects
- For example, to iterate over a list:

```
for x in list1:
    print(x)
```

```
A+
A
B
C+
D
```

for Loops

- Iterate over a string:

```
string = "Hi There"
for x in string:
    print(x, end = "") # to print on one line one after another
```

```
Hi There
```

- Iterate over a dictionary:

```
dict2 = {1:"machine", 2:"learning", 3:"with python"}
for key in dict2: # looping through keys in a dictionary
    val = dict2[key]
    print('key =', key)
    print('value =', val)
    print()
```

for Loops

UIC BUSINESS

- Equivalently, we can replace `dict2` with `dict2.keys()` and achieve the same result:

```
dict2 = {1:"machine", 2:"learning", 3:"with python"}
for key in dict2.keys(): # looping through keys in a dictionary
    val = dict2[key]
    print('key =', key)
    print('value =', val)
    print()
```

```
key = 1
value = machine

key = 2
value = learning

key = 3
value = with python
```

for Loops

UIC BUSINESS

- When looping through a dictionary, we can use the `items()` method to fetch the keys and values at the same time

```
for key, val in dict2.items():
    print('key =', key)
    print('value =', val)
    print()
```

- Iterate over a sequence of numbers:

```
for i in range(3,8): # the sequence from 3 to 7
    print('i =', i)
```

```
i = 3
i = 4
i = 5
i = 6
i = 7
```

for Loops

UIC BUSINESS

- When looping through a sequence, we can use `enumerate(iterable, start=0)` to fetch the indices and their corresponding values at the same time

```
for i, v in enumerate(list6):
    print(i, v)
```

```
0 Machine
1 Learning
2 with
3 Python
4 1.0.0
```

- Start the count from 1

```
for i, v in enumerate(list6, start=1):
    print(i, v)
```

for Loops

UIC BUSINESS

- The `zip()` function creates an iterator that aggregates two or more iterables, and then loops over this iterator

```
list_a = [1,2,3,4]
list_b = ['a','b','c','d']
for item in zip(list_a,list_b):
    print(item)
```

```
(1, 'a')
(2, 'b')
(3, 'c')
(4, 'd')
```

- We can now use `dict()` and `zip` to create a dictionary where keys are names and values are numbers

```
name_list = ['John', 'James', 'Jane']
phone_list = [979, 797, 897]
dict3 = dict(zip(name_list, phone_list)) # it works because here we use␣
↪zip on two lists; therefore, each element of the iterable has two␣
↪objects
dict3
```

```
{'John': 979, 'James': 797, 'Jane': 897}
```

List Comprehension

UIC BUSINESS

- Once we have an iterable, it's often required to perform three operations:
  - select some elements that meet some conditions
  - perform some operations on every element
  - perform some operations on some elements that meet some conditions
- Python has an idiomatic way of doing these, known as *list comprehension* (*listcomps*)
- Create a list containing square of odd numbers between 1 to 20:

```
list_odd = [] # start from an empty list
for i in range(1, 21):
    if i%2 !=0:
        list_odd.append(i**2)

list_odd
```

```
[1, 9, 25, 49, 81, 121, 169, 225, 289, 361]
```

List Comprehension

UIC BUSINESS

- List comprehension allows us to combine all this code in one line by combining the list creation, appending, the for loop, and the condition:

```
list_odd_lc = [i**2 for i in range(1, 21) if i%2 !=0]
list_odd_lc
```

```
[1, 9, 25, 49, 81, 121, 169, 225, 289, 361]
```

- Use the listcomps to generate a list of two-element tuples of non-equal integers between 0 and 3:

```
list_non_equal_tuples = [(x, y) for x in range(3) for y in range(3) if␣
↪x != y]
list_non_equal_tuples
```

```
[(0, 1),
 (0, 2),
 (1, 0),
 (1, 2),
 (2, 0),
 (2, 1)]
```

if-elif-else

UIC BUSINESS

- Conditional statements can be implemented by `if-elif-else` statement:

```
list4 = ["Machine", "Learning", "with", "Python"]
if "java" in list4:
    print("There is java too!")
elif "C++" in list4:
    print("There is C++ too!")
else:
    print("Well, just Python there.")
```

```
Well, just Python there.
```

Functions

UIC BUSINESS

- Functions are blocks of code that are named and do a specific job
- We can define a function using `def` keyword

```
def subtract_three_numbers(num1, num2, num3):
    result = num1 - num2 - num3
    return result
```

```
x = subtract_three_numbers(10, 3.0, 1)
print(x)
```

```
6.0
```

```
x = subtract_three_numbers(num3 = 1, num1 = 10, num2 = 3.0)
print(x)
```

```
6.0
```

Functions

UIC BUSINESS

- In Python functions, we can return any data type such as lists, tuples, dictionaries, etc
- We can also `return` multiple values, packed into one tuple

```
def string_func(string):
    return len(string), string.upper(), string.title()

string_func('coolFunctions') # observe the tuple
```

```
(13, 'COOLFUNCTIONS', 'Coolfunctions')
```

```
x, y, z = string_func('coolFunctions') # unpacking
print(x, y, z)
```

```
13 COOLFUNCTIONS Coolfunctions
```

Functions

UIC BUSINESS

- If we pass an object to a function and within the function the object is modified, the changes will be permanent

```
def list_mod(inp):
    inp.insert(1, 'AB')

list7 = [100, 'ML', 200]
list_mod(list7)
list7 # observe that the changes within the function appears outside␣
↪the function
```

```
[100, 'AB', 'ML', 200]
```

- Sometimes we do not know in advance how many positional or keyword arguments should be passed to the function
- Use * or ** before a `parameter_name`

Functions

UIC BUSINESS

- Define a function that receives the amount of money we can spend for grocery, and the name of items we need to buy
- The function prints the amount of money with a message as well as a capitalized acronym made out of items in the grocery list
- As we do not know in advance how many items we need to buy, the function should work with an arbitrary number of items in the grocery list
- Parameter accepting arbitrary number of arguments should appear last in the function definition

Functions


```
def grocery_to_do(money, *grocery_items): #the use of * before grocery_items is to allow an arbitrary number of arguments
    acronym = ''
    for i in grocery_items:
        acronym += i[0].title()
    print('You have {}$'.format(money)) # this is another way to write "print('You have ' + str(money) + '$')" using place holder {}
    print('Your acronym is', acronym)

grocery_to_do(40, 'milk', 'bread', 'meat', 'tomato')
```

```
You have 40$
Your acronym is MBMT
```

Modules and Packages

UIC BUSINESS

- As we develop our programs, it is more convenient to put functions into a separate file called *module*, and then *import* them when they are needed
  - *importing* a module within a code makes the content of the module available in that program
  - modules can store multiple classes and variables
  - to create a module, we can put the definition of functions in a file with extension *.py*
- We will see plenty of examples when we use existing Python packages and modules to conduct different machine learning tasks