Chapter Three
Data Handling in Python

Kibret Zewdu (MSc.)

Department of Computer Science
College of Informatics
University of Gondar

January 30, 2023

Outline

1. Data Types
2. Number in Python
   - Integers
   - Floating point numbers in Python
   - Complex numbers
3. String in Python
4. Boolean In Python
5. List In Python
6. Tuples In Python
7. Common Properties of of Sequence
8. Dictionary In Python
9. Set In Python
   - Set Operations

Data Types

- **Data Type** specifies which type of value a variable can store.
- **Data Types In Python**
  1. Number
  2. String
  3. Boolean
  4. List
  5. Tuple
  6. Set
  7. Dictionary
- ***type()***: function is used to determine a variable's type in Python.

Number in Python

- used to store numeric values.
- Python has three numeric types:
  - Integers
  - Floating point numbers
  - Complex numbers.

Number in Python

Integers

- Integers or int are positive or negative numbers with no decimal point.
- Integers in Python 3 are of unlimited size.

Example:

```
    a= 100
    b= -100
    c= 1*20
    print(a)
    print(b)
    print(c)

```

Number in Python

Type Conversion of Integer

int() function converts any data type to integer.

Example

```
    a = "101" # string
    b=int(a) # converts string data type to integer.
    c=int(122.4) # converts float data type to integer.
    print(b)
    print(c)

```

Output

```
101
122
```

Number in Python

Floating point numbers

- It is a positive or negative real numbers with a decimal point.

Example

```python
    a = 101.2
    b = -111.23
    c = 2.3*3
    print(a)
    print(b)
    print(c)

```

Output

```
101.2
-111.23
6.8999999999999995
```

Number in Python

Type Conversion of Floating point numbers

float() function converts any data type to floating point number.

Example

```python
    a = "120.35" # string
    b=float(a) # converts string data type to floating point number.
    c=float(122) # converts integer data type to floating point number.
    print(b)
    print(c)

```

Output

```
120.35
122.0
```

Number in Python

Complex numbers

- Complex numbers are combination of a real and imaginary part.
- Complex numbers are in the form of X+Yj, where X is a real part and Y is imaginary part.

Example

```
    a = complex(5) # convert 5 to a real part  and zero imaginary part
    print(a)
    b=complex(101,23) # convert 101 with real part and 23 as imaginary part
    print(b)

```

Output

```
(5+0j)
(101+23j)
```

String in Python

- A **string** is a sequence of symbols/characters delimited by a single quote ('), double quotes("), triple single quotes (”’), or triple double quotes (""").
- Python 3, all string sequences are Unicode characters by default.

Example

```
    str = 'Computer Science'
    print('str-', str) # print Computer Science
    print('str[0]-', str[0])# print first char 'C'
    print('str[1:3]-', str[1:3]) # print string from position 1 to 3 'om'
    print('str[3:]-', str[3:]) # print string staring from 3rd char 'puter Science'
    print('str *2-', str *2 ) # print string two times
    print("str +'yes'-", str +'yes') # concatenated string

```

String in Python

Output

```
str- Computer Science
str[0]- c
str[1:3]- om
str[3:]- puter Science
str *2- Computer scienceComputer Science
str +'yes'- Computer Scienceyes
```

String in Python

- We can insert a single quote in a string delimited for double quote, and vice versa:

```
    "A single quote (') inside a double quote"
    'Here "double quotes" inside single quotes'

```

- Remember is that if we begin a string with a type of quote, we must finish it with the same type of quote.
- The following string is not valid:

```
    >>> "Mixing quote types leads to the dark side'

```

File <̈stdin>,̈ line 1
"Mixing quote types leads to the dark side'
SyntaxError: EOL while scanning single-quoted string

String

String Manipulation

- **Strings** are immutable. Once a string is created, it can't be modified.
- In the following example there is a string that represents an amino-acid sequence and it is called signal_peptide:

```
    >>> signal_peptide = 'MASKATLLLAFTLLFATCIA'
    >>> signal_peptide.lower()
    'maskatlllaftllfatcia'
    #The original string has not been modifie
    >>> signal_peptide
    'MASKATLLLAFTLLFATCIA'
    #To get new lower case string with the same name as the previous one, we need to assign it:
    >>> signal_peptide = signal_peptide.lower()
    >>> signal_peptide
    'maskatlllaftllfatcia'

```

String

Methods Associated with Strings

1. **replace**(old,new[,count]): Allows us to replace a portion of a string (old) with another (new). If the optional argument count is used, only the first count occurrences of old will be replaced:

```
    >>> dna_seq = ’GCTAGTAATGTG’
    >>> m_rna_seq = dna_seq.replace(’T’,’U’)
    >>> m_rna_seq
    ’GCUAGUAAUGUG’

```

2. **count**(sub[, start[, end]]): Counts how many times the substring sub appears, between the start and end positions (if available).

```
   >>> dna_seq
   ’GCTAGTAATGTG’
   >>> g = dna_seq.count("G")
   4
```

String

Methods Associated with Strings

- **find**(sub[,start[,end]]): Returns the position of the substring sub, between the start and end positions (if available). If the substring is not found in the string, it returns the value -**1**:

```
    >>> m_rna_seq
     'GCUAGUAAUGUG'
    >>> m_rna_seq.find('AUG')
     7
    >>> m_rna_seq.find('GGG')
     -1

```

- **index**(sub[,start[,end]]): Works like ***find***(). **index** will raise a ***ValueError*** exception when the substring is not found. This method is recommended over **find()** because the value -**1** could be interpreted as a valid value, while a ValueError returned by index() can't be taken as a valid value.

String

Methods Associated with Strings

- **split**([sep [,maxsplit]]): Separates the “words” of a string and returns them in a list.
  If a separator (sep) is not specified, the default separator will be a white space:

```
>>> 'This string has words separated by spaces'.split()
['This', 'string', 'has', 'words', 'separated', 'by', 'spaces']
>>> "Tedy Afro, aa-2333, tedyafr@example.com".split(",")
['Tedy Afro', 'aa-2333', 'tedyafr@example.com']

```

- **Bioinformatics Application**: Parsing *BLAST* Files.
- The BLAST standalone executable can generate output as a “tab separated file”.

String

Methods Associated with Strings

- **join([seq])**: Joins the sequence using a string as a “glue character”:

```
';'.join(['Alex Doe', '5555-2333', 'nobody@example.com'])
'Alex Doe;5555-2333;nobody@example.com'

```

- To join a sequence without any glue character, use empty quotes (**""**):

```
>>> ''.join(['A','C','A','T'])
'ACAT'

```

Boolean In Python

It is used to store two possible values either **true** or **false**

Example

```
str="comp sc"
upper_case=str.isupper() \# test if string contains upper case
isMale = True \# Boolean true value
print(upper_case)
print (isMale)


```

Output

```
False
True

```

List In Python

List

- List are ordered collections of items.
- It is represented by elements separated by commas and enclosed between **square brackets**.
- A list can hold similar or different kinds of data.
- You can also create an empty list and add elements later.

- To define and name a list:

```
    >>> list1 = [1, 2, 3, 4, 5]
    >>> list2 = [1, ’two’, 3, 4, ’last’]
    >>> nested_list = [1, ’two’, list1, 4]
    >>> empty_list = []

```

List In Python

Accessing List Elements In Python

- list elements can be accessed using their index starting at zero.

```
    >>> first_list = [1, 2, 3, 4, 5]
    first_list[0]
    first_list[1]

```

- Negative numbers are used to access lists from the right:

```
    >>> first_list = [1, 2, 3, 4, 5]
    >>> first_list[-1] \# returns 5
    >>> first_list[-4] \# returns 2

```

List

List with Multiple Repeated Items

- You can also turn a non-list object into a list by using the built-in function `list()`:

```
    >>> aseq = "atggctaggc"
    >>> list(aseq)
    ['a', 't', 'g', 'g', 'c', 't', 'a', 'g', 'g', '
    c']

```

- To initialize a list with the same item repeated multiple times, you can use the ***** operator like this:

```
    >>> samples = ['red'] * 5
    >>> samples
    ['red', 'red', 'red', 'red', 'red']

```

List

List Comprehension

- A list can be created from another list.

```
>>> [3 * x for x in a]
[0, 3, 6, 9, 12, 15]%

```

- To find all animals whose name contains letter 'i':

```
>>> animals = [' Lion', ' Dog ', 'Tiger ']
>>> [x.strip() for x in animals if 'i' in x]
['lion', 'Tiger']

```

List

Modifying Lists

- Lists can be modified by adding, removing, or changing their elements.
- **Adding**: There are three ways to add elements into a list: **append**, **insert**, and **extend**.

1. **append(element)**: Adds an element at the end of the list.

```
    >>> first_list.append(99)

```

2. **insert(position,element)**: Inserts the element element at the position position.

```
    >>> first_list.insert(2,50)

```

3. **extend(list)**: Extends a list by adding a list to the end of the original list.

```
    >>> first_list.extend([6,7,8])
    >>> [1,2,3] + [4,5] # + is the same
```

List

Modifying Lists

- **Removing**: There are three ways to remove elements from a list: **pop**, **remove** and **del**
  - **pop(/index/)**: Removes the element in the index position and returns it to the point where it was called.
  - Without parameters, it returns the last element.

```
    >>> first_list
    [1, 2, 50, 3, 4, 5, 99, 6, 7, 8]
    >>> first_list.pop()
    8
    >>> first_list.pop(2)
    50
    >>> first_list
    [1, 2, 3, 4, 5, 99, 6, 7]

```

List

Modifying Lists

- **remove(element)**
  - Removes the element specified in the parameter.
  - In the case where there is more than one copy of the same object in the list, it removes the first one, counting from the left.
  - this function does not return anything.

```
    >>> first_list.remove(99)
    >>> first_list
    [1, 2, 3, 4, 5, 6, 7]

```

  - Trying to remove a nonexistent element raises an error
- **del**: it is a command to delete an element from the list.

```
    del first_list[0]

```

List

Copying a List

- To copy a list, use the **copy** method from the copy module.

```
    >>> import copy
    >>> a = [1, 2, 3]
    >>> b = copy.copy(a)
    >>> b.pop()
    3
    >>> a
    [1, 2, 3]
```

- Without using the **copy** module

```
    b = a[:]
```

- Note: The assignment operator (=) doesn't copy the values, it copies a reference to the original object.

Tuples In Python

Tuples are Immutable Lists

- A **tuple** is a collection of ordered objects with the characteristic that once created, it cannot be modified. They are referred to as “immutable lists.”
- **Immutable objects** cannot be modified after they are created.
- **Tuple** elements are enclosed between parentheses:

```
  point = (23, 56, 11) \# Here elements are enclosed by ()

```

- When the tuple has only one element, you should use a trailing comma to tell python interpreter that it is a tuple not an expression:

```
 lone_element_tuple = (5,)

```

Common Properties of of Sequence

Indexing

- You can apply these properties to lists, tuples, and strings.
- **Indexing**: elements in the sequences are ordered, we can gain access to any element through an index that begins at zero.

```
    >>> point = (23, 56, 11)
    >>> point[0]
    23
    >>> sequence = 'MRVLLVALALLALAASATS'
    >>> sequence[5]
    'V'
    >>> parameters = ['UniGene', 'dna', 'Mm.248907', 5]
    >>> parameters[2]
    'Mm.248907'

```

Common Properties of of Sequence

Indexing

- To access the elements of a sequence from the right by using ***negative numbers***

```
>>> parameters[-3]
'dna'
>>> sequence[-1]
'S'

```

- To access an element that is inside a sequence, which is itself inside another sequence, you need to use another index:

```
>>> seqdata = ('MRVLLVALALLA', 12, '5FE9EEE8EE2DC2C7')
>>> seqdata[0][5] # The 6th element of the first element in the sequence
'V'

```

Common Properties of of Sequence

Slicing

- used to select a portion of a sequence.
- consists of using two indexes separated by a colon (**:**).

```
    >>> string ="Python"
    >>> string[0:2]
    'Py'

```

- If the first sub-index is omitted, the index value defaults to the first position (**0**):

```
    >>> string[:2]
    'Py'

```

- If the second sub-index is omitted, the index value defaults to the last position (-1)

```
    >>> string[4:] # from index 4 to the end
    'on'
```

Common Properties of of Sequence

Slicing

- There is a third, optional index to skip positions (step argument):

```
    >>> string[1:5]
    'ytho'
    >>> string[1:5:2]
    'yh'

```

- A step with a negative number is used to count backwards. So -1 (in the third position) can be used to invert a sequence:

```
    >>> string[::-1]
    'nohtyP'

```

- Note that slicing always returns another sequence

Common Properties of of Sequence

Membership Test

- To verify whether an element belongs to a sequence, using the **in** keyword:

```
>>> point = (23, 56, 11)
>>> 11 in point
True
>>> my_sequence = 'MRVLLVALALLALAASATS'
>>> 'X' in my_sequence
False

```

Common Properties of of Sequence

Concatenation

- concatenate two or more sequences of the same class using the "+" sign:

```
>>> point = (23, 56, 11)
>>> point2 = (2, 6, 7)
>>> point + point2
(23, 56, 11, 2, 6, 7)
>>> dna_seq = 'ATGCTAGACGTCCTCAGATAGCCG'
>>> tata_box = 'TATAAA'
>>> tata_box + dna_seq
'TATAAAATGCTAGACGTCCTCAGATAGCCG'

```

- Sequences of different types can't be concatenated.

Common Properties of of Sequence

len, max, and min

- **len()** returns the length (the number of items) of a sequence:

```
    >>> my_sequence = 'MRVLLVALALLALAASATS'
    >>> len(my_sequence)
    19

```

- **max()** and **min()** applied over a sequence of numbers return, as expected, the maximum and the minimum value:

```
    >>> min(point)
    11

```

- **max()** and**min()** applied to strings return a character according to the maximum or minimum value of its ASCII code:

```
    >>> my_sequence = 'MRVLLVALALLALAASATS'
    >>> max(my_sequence)
    'V'
```

Common Properties of of Sequence

Turn a Sequence into a List

- To convert a sequence (like a tuple or a string) into a list, use the list() method:

```
>>> tata_box = 'TATAAA'

>>> list(tata_box)

['T', 'A', 'T', 'A', 'A', 'A']
```

- Using a list provides us with methods to indirectly modify a string

Dictionary In Python

Dictionary

- are ***unordered collection*** of items and each item consist of a key and a value.
- They are defined by enclosing is *key:value* pairs between curly brackets {}
- The key is the index used to retrieve the value.

Example

```
my_dict = {'subject': 'Python', 'class': 'Pg Building'}
print(my_dict)
print ("Subject : ", my_dict['subject'])

```

Output: {'subject': 'Python ', 'class': 'Pg Building'}
Subject : 'Python'

Dictionary In Python

- This dictionary works as a translation table that allows us to translate between the one-letter amino acid code to a three-letter code.

```
    >>> iupac = {‘A’:‘Ala’,‘C’:‘Cys’,‘E’:‘Glu’}
    >>> iupac[‘E’]
    ‘Glu’

```

- Not every object can be used as a dictionary key.
- Only immutable objects like strings, tuples and numbers can be used as keys.

Dictionary In Python

- A dictionary can also be created from a sequence with **dict()**:

```
>>> rgb = [(‘red’,‘ff0000’), (‘green’,‘00ff00’), (‘blue’,‘0000ff’)]
>>> colors = dict(rgb)
>>> colors
{’red’: ’ff0000’, ’green’: ’00ff00’, ’blue’: ’0000ff’}
```

- **dict** also accepts `name=value` pairs in the argument list:

```
>>> rgb = dict(‘red’=‘ff0000’, ‘green’=‘00ff00’, ‘blue’=‘0000ff’)
```

- To create an empty dictionary and add elements later:

```
>>> rgb = {}
>>> rgb[’red’] = ’ff0000’
>>> rgb[’green’] = ’00ff00’
```

Dictionary In Python

- **len()**, returns the number of elements in the dictionary:

```
>>> len(iupac)
3

```

- To add values to a dictionary,

```
>>> iupac['S'] = 'Ser'

```

- Dictionaries are unordered because they don't keep track of the order of their elements.
- When you request to see the contents of the dictionary, you may or may not get the elements in the same order as they were entered

Dictionary In Python

Dictionary Methods

- To get the keys or values of a dictionary, there are methods like **keys()** and **values()**:

```
    >>> iupac = {'E': 'Glu', 'X': 'Xaa', 'C': 'Cys', 'A': 'Ala'}
    >>> iupac.keys()
    dict_keys(['E', 'X', 'C', 'A'])
    >>> iupac.values()
    dict_values(['Glu', 'Xaa', 'Cys', 'Ala'])

```

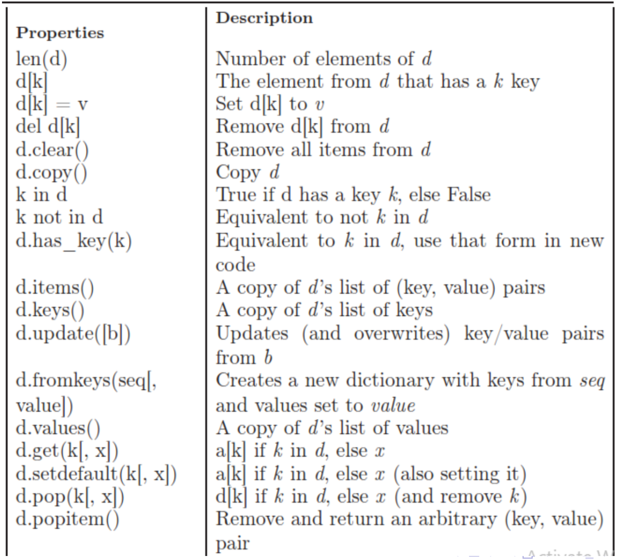
- **items()** is another method to access elements of dictionaries which returns a **dictionary view** a tuple of every key-value pair.

```
    >>> iupac.items()
    dict_items([('E', 'Glu'), ('A', 'Ala'), ('C', 'Cys'), ('X', 'Xaa')])

```

Dictionary In Python

Dictionary Methods



Properties	Description
len(d)	Number of elements of *d*
d[k]	The element from *d* that has a *k* key
d[k] = v	Set d[k] to *v*
del d[k]	Remove d[k] from *d*
d.clear()	Remove all items from *d*
d.copy()	Copy *d*
k in d	True if d has a key *k*, else False
k not in d	Equivalent to not *k* in *d*
d.has_key(k)	Equivalent to *k* in *d*, use that form in new code
d.items()	A copy of *d*'s list of (key, value) pairs
d.keys()	A copy of *d*'s list of keys
d.update([b])	Updates (and overwrites) key/value pairs from *b*
d.fromkeys(seq[, value])	Creates a new dictionary with keys from *seq* and values set to *value*
d.values()	A copy of *d*'s list of values
d.get(k[, x])	a[k] if *k* in *d*, else *x*
d.setdefault(k[, x])	a[k] if *k* in *d*, else *x* (also setting it)
d.pop(k[, x])	d[k] if *k* in *d*, else *x* (and remove *k*)
d.popitem()	Remove and return an arbitrary (key, value) pair

Dictionary In Python

Query Dictionary Values

- To query a value from a dictionary without the risk of invoking an exception, use **get(k,x)**, where k is the key of the element to extract, while x is the element that will be returned in case k is not found as a key of the dictionary.

```
>>> iupac = {'E': 'Glu', 'X': 'Xaa', 'C': 'Cys', 'A': 'Ala'}
>>> iupac.get('A','No translation available')
'Ala'
>>> iupac.get('Z','No translation available')
'No translation available'

```

- If you omit x, and there is no k key in the dictionary, it returns **None**.

```
>>> iupac.get('Z')
None
```

Dictionary In Python

Dictionary Methods

Erasing Elements

- To erase elements from a dictionary, use the **del** instruction:

```
>>> iupac = {'E': 'Glu', 'X': 'Xaa', 'C': 'Cys', 'A': 'Ala'}
>>> del iupac['A']
>>> iupac
{'C': 'Cys', 'X': 'Xaa', 'E': 'Glu'}

```

Sets

Set

- It is an unordered collection of unique and immutable (which cannot be modified)items.
- Every element is unique and is defined by {}
- The most common uses of sets are **membership testing**, **duplicate removal**, and the application of **mathematical operations**:
    - **intersections**,
    - **unions**,
    - **differences**, and
    - **symmetrical differences**.

Sets

Set In Python

Example

```python
set1={11,22,33,22}
print(set1)
set2={11,22,33,22, 44, 55}
print(set2)

```

Output

```
{33, 11, 22}
{33, 22, 55, 11, 44}

```

Set In Python

Creating a Set

- Sets are created with the instruction **set()**:

```
    >>> first_set = {’CP0140.1’,’XJ8113.5’,’EF3616.3’}

```

- To create an empty set and then add the elements as needed:

```
    >>> first_set = set()
    >>> first_set.add(’CP0140.1’)
    >>> first_set.add(’XJ8113.5’)
    >>> first_set
    {’CP0140.1’,’XJ8113.5’}

```

Set In Python

Creating a Set

- set by comprehension:

```
>>> {2 * x for x in [1,2,3]}
{2, 4, 6}

```

- A **set** does not accept repeated elements

```
>>> {2*x for x in [1,1,2,2,3,3]}
{2, 4, 6}
>>> uniques = {2,2,3,4,5,3} # Removes duplicate elements

```

Set Operations

Intersection

- To get the common elements in two sets, use the operator **intersection()**:

```
>>> set1 = {1, 3, 5, 7}
>>> set2 = {4, 5, 2, 7, 8}
>>> common = set1.intersect(set2)
{5, 7}

```

- It is equivalent to &:

```
>>> common = set1 & set2
{5, 7}

```

Set Operations

Union

- The union of two (or more) sets is the operator **union()**
- its abbreviated form is |

```
>>> set1.union(set2)
{1, 2, 3, 4, 5, 7, 8}

```

- It is equivalent to |:

```
>>> set1 | set2
{1, 2, 3, 4, 5, 7, 8}

```

Set Operations

Difference

- A **difference** is the resulting set of elements that belongs to one set but not to the other and it achieved in python using the method name **difference()**.
- Its shorthand is -

```
>>> set1.difference(set2)
{1, 3}

```

- It is equivalent to -:

```
>>> set1 - set2
 {1, 3}

```

Set Operations

Symmetric Difference

- A symmetric difference refers to those elements that are not a part of the intersection and its operator is **symmetric_difference()**
- and it is shortened as ˆ

```
 >>> set1.symmetric_difference(set2)
 {1, 2, 3, 4, 8}

```

- It is equivalent to ˆ:

```
  >>> set1 ^ set2
  {1, 3}

```

Set Operations

Shared Operations with Other Data Types

Maximum, Minimum, and Length

- Sets share some properties with sequences, such as :
  - **max**,
  - **min**,
  - **len**,
  - **in**, etc.
- As we can expect, these properties work in the same way.
- Type ***help(set())*** in the console to see all methods associated with sets.

End of The Chapter

Figure: