SEQUENCE DATA TYPES & FILE IO

Strings

- String is a subtype of the sequence data type
- Written with either single or double quote
- Note: there is no character data type in Python. A character is a string with one character.

```
s1 = "This is a string!"
s2 = 'Python is so awesome.'
```

Accessing a String

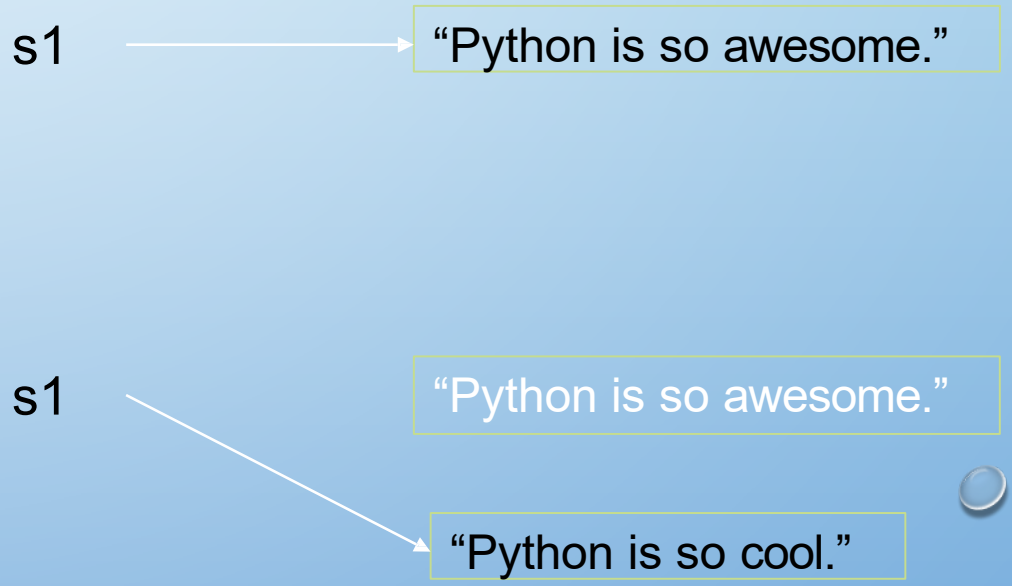
- Strings can be accessed element-wise as they are technically just sequences of character elements.
- String elements can be indexed with typical bracket notation and range of characters slicing

```
s1 = "This is a string!"
s2 = 'Python is so awesome.'
print(s1[3])
# output: s
print(s2[5:15])
# output: n is so awesome
```

Modifying a String

- Strings are *immutable* – you cannot update the value of an existing string object. However, you can reassign your variable name to a new string object to perform an "update".

```
s1 = "Python is so awesome."
print(s1)
s1 = "Python is so cool."
print(s1)
```



Modifying a String

Alternatively:

```
s1 = "Python is so awesome."
print(s1)
s1 = s1[:13]+"cool."
print(s1)
```

creates a substring “Python is so ”, which is concatenated with “cool.”, stored in memory and associated with the name s1.

“+” operator concatenates two strings

“*” operator concatenates multiple copies of a single string object.

in and not in test character membership within a string.

Escape Characters

most common:

- ‘\n’ – newline
- ‘\s’ – space
- ‘\t’ – tab

Built-in String Methods

- Note that these *return* the modified string value; we cannot change the string's value in place because they're immutable!
- s.upper() and s.lower()

```
s1 = "Python is so awesome."
print(s1.upper())
print(s1.lower())
```

Built-in String Methods

- s.isalpha(), s.isdigit(), s.isalnum(), s.isspace()
  - – return True if string *s* is composed of alphabetic characters, digits, either alphabetic and/or digits, and entirely whitespace characters, respectively.
- s.islower(), s.isupper()
  - – return True if string *s* is all lowercase and all uppercase, respectively.

```
print("WHOA".isupper())
print("12345".isdigit())
print("    \n ".isspace())
print("hello!".isalpha())
```

Build-in String Methods

- str.split([sep [,maxsplit]])
- – Split *str* into a list of substrings.
  - *sep* argument indicates the delimiting string (defaults to consecutive whitespace).
  - *maxsplit* argument indicates the maximum number of splits to be done (default is -1).
- str.rsplit([sep [,maxsplit]])
- Exercise: Let str = "TABLE_DUMP|1130191746|B|144.228.241.81|1239|128.186.0.0/16|1239 2914 174 11096 2553|IGP|144.228.241.81|0|-2|1239:321 1239:1000 1239:1011|NAG||", write python code to assign the last value in the field (in integer) before IGP to variable a?

Build-in String Methods

- str.strip([chars])
- – Return a copy of the string *str* with leading and trailing characters removed.
- *chars* string specifies the set of characters to remove (default is whitespace).
- str.rstrip([chars])
- – Return a copy of the string *str* with only trailing characters removed.

```
print("Python   programming    is fun!".split())
print("555-867-5309".split('-'))
print("***Python   programming    is fun***".strip('*'))
```

Built-in String Methods

- *str*.capitalize()
  - – returns a copy of the string with the first character capitalized and the rest lowercase.
- *str*.center(*width* [,*fillchar*])
  - – centers the contents of the string *str* in field-size *width*, padded by *fillchar* (defaults to a blank space). See also *str*.ljust() and *str*.rjust().
- *str*.count(*sub* [,*start*[, *end*]])
  - – return the number of non-overlapping occurrences of substring *sub* in the range *[start*, *end].* Can use slice notation here.
- *str*.endswith(*suffix*[, *start*[, *end*]])
  - – return True if the string *str* ends with suffix, otherwise return False.
  - Optionally, specify a substring to test. See also *str*.startswith().

```
print("i    LoVe   pYtHoN".capitalize())
print("centered".center(20,'*'))
print("mississippi".count("iss"))
print("mississippi".count("iss", 4, -1))
print("mississippi".endswith("ssi"))
print("mississippi".endswith("ssi", 0, 8))
```

Built-in String Methods

- *str*.find(*sub* [,*start*[, *end*]])
  - – return the lowest index in the string where substring *sub* is found, such that *sub* is contained in the slice *str*[*start*:*end*].
  - Return -1 if *sub* is not found.
  - See also *str*.rfind().
- *str*.index(*sub* [,*start*[, *end*]])
  - – identical to find(), but raises a *ValueError* exception when substring *sub* is not found.
  - See also *str*.rindex().
- *str*.join(*iterable*)
  - – return a string that is the result of concatenating all of the elements of *iterable*.
  - The *str* object here is the delimiter between the concatenated elements.
- *str*.replace(*old*, *new*[, *count*])
  - – return a copy of the string *str* where all instances of the substring *old* are replaced by the string *new* (up to *count* number of times).

Built-in String Methods

```
print("whenever".find("never"))
print("whenever".find("what"))
print("whenever".index("never"))
print("whenever".index("what"))


print("-".join(['555','867','5309']))
print("    ".join(['Python',  'is',  'awesome']))
print("whenever".replace("ever",   "ce"))
```

String formatting

- There are several ways for string format. The preferred way is to use f-strings.
  - By putting an *f* in front of a string literal, we create an f-string. Example:

```
t = f”This is an f-string.”
print(t)
```

- An f-string can have placeholders { }, which can contain expressions (variables, functions) and modifiers to format the value:

```
val = 500
t = f”This is an f-string. val = {val} ”
print(t)
```

Placeholders in f-string

- A placeholder can contain any expressions including constants and functions.
- An f-string can have any number of placeholders.

```
t = f"val = {500} "
print(t)
v = 500
t = f"v*2 = {v * 2} "
print(t)
price = 59
taxRate = 0.075
t = f"Total price is {price + price * taxRate} . "
print(t)
a = 'apple'
b = 'grape'
t = f"I like {a.upper() + ' ORANGE'} and {b.upper()}."
print(t)
```

Modifier

- We can further format the values in the in the placement holder with modifier
  - Modifier starts with a ":" and only takes some fixed format.
  - Placeholder with modifier has the form of {expression modifier}

- See the list of modifiers at https://www.w3schools.com/python/python_string_formatting.asp
- Exercise: Print the header of the ps command.

```
print( f"val = {1000000000:,} " )  # comma as thousand separator
print( f"val = {12.342344:.3f} " )  # float point number with a 3 decimals
print(f" {'Hello':>30} ")  # Right align to 30 letter space
print(f" {'100':^10} ")    # center of 10 digits
```

String format()

- Old form of string formatting
- The signature is:

```
str.format(*args, **kwargs)
```

- *args argument indicates that format accepts a variable number of positional arguments,
- **kwargs indicates that format accepts a variable number of keyword arguments.

- str can contain literal text or replacement fields, which are enclosed by braces {}.
- Each replacement field contains either the numeric index of a positional argument, or the name of a keyword argument. A copy of the string is returned where each replacement field is replaced with the string value of the corresponding argument.

String formatting examples

```
print('{0}{1}{2}'.format('a','b','c'))
print('{}{}{}'.format('a','b','c'))
print('{2}{1}{0}'.format('a','b','c'))
print('{2}{1}{0}'.format(*'abc'))
print('{0}{1}{0}'.format('abra',   'cad'))
```

```
print('Coords: {lat}, {long}'.format(lat='37.24N',   long='-115.81W'))
coord  =  {'lat':    '37.24N',  'long':    '-115.81W'}
print('Coords: {lat}, {long}'.format(**coord))
```

Lists

- When to use lists?
  - When you need a collection of elements of varying type.
  - When you need the ability to order your elements.
  - When you need the ability to modify or add to the collection.
  - When you don't require elements to be indexed by a custom value.
  - When you need a stack or a queue.
  - When your elements are not necessarily unique.

Creating Lists

- To create a list in Python, we can use bracket notation to either create an empty list or an initialized list.

```
mylist1 = [] # Creates an empty list
mylist2 = [expression1, expression2, ...]
mylist3 = [expression for variable in sequence]
```

- The first two are referred to as *list displays*, where the last example is a *list comprehension*.

Creating Lists

- We can also use the built-in list constructor to create a new list.

```
mylist1    = list() #    Creates    an    emptylist
mylist2    =      list(sequence)
mylist3    =      list(expression for variable in sequence)
```

- The sequence argument in the second example can be any kind of sequence object or iterable. If another list is passed in, this will create a copy of the argument list.

Creating Lists

- Note that you cannot create a new list through assignment.
  - Assignment creates an alias, but not the new list object.

```
#   mylist1 and    mylist2 point to   the    same   list
mylist1 = mylist2 = []
# mylist3  mylist3  mylist4    and    mylist4
mylist3=[]
mylist4=mylist3
mylist5    =  [];    mylist6    =  [] #  different  lists
```

Accessing list elements

- If the index of the desired element is known, you can simply use bracket notation to index into the list.
- If the index is not known, use the `index()` method to find the first index of an item. An exception will be raised if the item cannot be found.

```
mylist  =  [34,67,45,29]
print(mylist[2])
mylist =  [34,67,45,29]
print(mylist.index(67))
```

Slicing

- **The length of the list is accessible through `len(mylist)`.**
- **Slicing is an extended version of the indexing operator and can be used to grab sublists.**

```
mylist[start:end]  # items from start to end-1
mylist[start:]     # items from start to end of the array
mylist[:end]       # items from beginning to end-1
mylist[:]          # a copy of the whole array
```

- **You may also provide a step argument with any of the slicing constructions above.**

```
mylist[start:end:step]
# items from start to end-1, incremented by step
```

Slicing

- The start or end arguments may be negative numbers, indicating a count from the end of the array rather than the beginning. This applies to the indexing operator.

```
mylist[-1]          # last element in the list
Mylist[-2:]         # the last two items of the array
mylist[:-2]         # all except the last two items
```

- Examples:

```
mylist  =  [34,   56,    29,    73,    19,    62]
print(mylist[-2])
print(mylist[-4::2])
```

Inserting Elements

- To add an element to an existing list, use the append() method.
- Use the extend() method to add all of the items from another list

```
mylist  =  [34,   56,    29,    73,    19,    62]
mylist.append(47)
print(mylist)

mylist =  [34,   56,    29,    73,    19,    62]
mylist.extend([47,81])
print(mylist)
```

Inserting/Removing Elements

- Use the insert(*pos, item*) method to insert an item at the given position. You may also use negative indexing to indicate the position.
- Use the remove() method to remove the first occurrence of a given item. An exception will be raised if there is no matching item in the list.

```
mylist  =  [34,   56,    29,    73,    19,    62]
mylist.insert(2,47)
print(mylist)
mylist =  [34,   56,    29,    73,    19,    62]
mylist.remove(29)
print(mylist)
```

List as a Stack

- You can use lists as a quick stack data structure.
- The append() and pop() methods implement a LIFO structure.
- The pop(*index*) method will remove and return the item at the specified index. If no index is specified, the last item is popped from the list.

```
stack   =  [34,   56,    29,    73,    19,    62]
stack.append(47)
print(stack)
stack.pop()
print(stack)
```

List as a Queue

- Lists *can* be used as queues natively since insert() and pop() both support indexing. However, while appending and popping from a list are fast, inserting and popping from the beginning of the list are slow.
- Use the special *deque* object from the *collections* module.

```
from collections import deque
queue = deque([35, 19, 67])
print(queue)
queue.append(42)
queue.append(23)
print(queue)
print(queue.popleft())
print(queue)
print(queue.popleft())
print(queue)
```

Some other useful operations

- The `count(x)` method will give you the number of occurrences of item x within the list.
- The `sort()` and `reverse()` methods sort and reverse the list in place. The `sorted(mylist)` and `reversed(mylist)` built-in functions will return a sorted and reversed copy of the list, respectively.

```
mylist = ['a', 'b', 'c', 'd', 'a', 'f', 'c']
print(mylist.count('a'))
mylist = [5, 2, 3, 4, 1]
mylist.sort()
print(mylist)
mylist.reverse()
print(mylist)
```

Custom Sort

- Both the `sorted()` built-in function and the `sort()` method of lists accept some optional arguments.

```
sorted(iterable[, cmp[, key[, reverse]]])
```

- The *cmp* argument specifies a custom comparison function of two arguments which should return a negative, zero or positive number depending on whether the first argument is considered smaller than, equal to, or larger than the second argument. The default value is None.
- The *key* argument specifies a function of one argument that is used to extract a comparison key from each list element. The default value is None.
- The *reverse* argument is a Boolean value. If set to True, then the list elements are sorted as if each comparison were reversed.

Custom Sort

```
mylist  =  ['b',  'A',   'D',   'c']
mylist.sort(key    =  str.lower)
print(mylist)
```

Set

- When to use set?
  - When the elements must be unique.
  - When you need to be able to modify or add to the collection.
  - When you need support for mathematical set operations.
  - When you don't need to store nested lists, sets, or dictionaries as elements.

Creating Set

- Create an empty set with the set constructor.

```
myset=    set()
myset2    =    set([])    #    both  are empty sets
```

- Create an initialized set with the set constructor or the { } notation. Do not use empty curly braces to create an empty set – you'll get an empty dictionary instead.

```
myset=    set(sequence)
myset2    =    {expression    for  variable  in
sequence}
```

Hashable Items

- The way a set detects non-unique elements is by indexing the data in memory, creating a hash for each element. This means that all elements in a set must be *hashable*.
- All of Python's immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are. Objects which are instances of user-defined classes are also hashable by default.

Mutable operations

- *set |= other | ...*
  - Update the set, adding elements from all others.
- *set &= other & ...*
  - Update the set, keeping only elements found in it and all others.
- *set -= other | ...*
  - Update the set, removing elements found in others.
- *set ^= other*
  - Update the set, keeping only elements found in either set, but not in both.

```
s1  =  set('abracadabra')
s2 =  set('alacazam')
print(s1)
print(s2)
s1 |= s2
print(s1)
s1 =  set('abracadabra')
s1 &= s2
print(s1)
```

Set Operations

- The following operations are available for both set and frozenset types.
- Comparison operators >=, <= test whether a set is a superset or subset, respectively, of some other set. The > and < operators check for proper supersets/subsets.

```
s1 = set('abracadabra')
s2 = set('bard')
print(s1 >=    s2)
print(s1 > s2)
print(s1 <= s2)
```

Set Operations

- **Union:** `set | other | …`
  - Return a new set with elements from the set and all others.
- **Intersection:** `set & other & …`
  - Return a new set with elements common to the set and all others.
- **Difference:** `set – other – …`
  - Return a new set with elements in the set that are not in the others.
- **Symmetric Difference:** `set ^ other`
  - Return a new set with elements in either the set or other but not both.

```
s1  =  set('abracadabra')
print(s1)
s2 =  set('alacazam')
print(s2)
print(s1|s2)
print(s1&s2)
print(s1-s2)
print(s1^s2)
```

Tuples

- **When to use tuples?**
  - When storing elements that will not need to be changed.
  - When performance is a concern.
  - When you want to store your data in logical immutable pairs, triples, etc.

Creating Tuples

- With an empty set of parentheses
- Pass a sequence type object into the tuple() constructor.
- By listing comma-separated values. Note: These do not need to be in parentheses but they can be.
- One quirk: to initialize a tuple with a single value, use a trailing comma.

```
t1 = (1, 2, 3, 4)
t2 = "a","b","c","d"
t3 = ()
t4 = ("red",   )
print(t1)
print(t2)
print(t3)
print(t4)
```

Tuple Operations

- Similar to lists and support a lot of the same operations.
- Accessing elements: use bracket notation (e.g. t1[2]) and slicing.
- Use len(t1) to obtain the length of a tuple.
- The universal immutable sequence type operations are all supported by tuples.
  - +, *
  - in, not in
  - min(t), max(t), t.index(x), t.count(x)

Packing/Unpacking

- packing “packs” a collection of items into a tuple
- unpack a tuple via Python’s multiple assignment feature

```
s = "Susan", 19, "CS" # tuple packing
print(s)
name, age, major = s # tuple unpacking
print(name)
print(age)
print(major)
```

Dict

- When to use dictionaries?
  - When you need to create associations in the form of key:value pairs.
  - When you need fast lookup for your data, based on a custom key.
  - When you need to modify or add to your key:value pairs.

Creating a Dictionary

- Create an empty dictionary with empty curly braces or the dict() constructor.
- You can initialize a dictionary by specifying each key:value pair within the curly braces.
- Note that keys must be *hashable* objects.

```
#constructing a dictionary
d1 = {}
d2 = dict()    #  both   empty
d3 = {"Nme":   "Susan",   "Age": 19,    "Major":   "CS"}
d4 = dict(Name="Susan",    Age=19,    Major="CS")
d5 =dict(zip(['Name',  'Age', 'Major'],  ["Susan",  19,    "CS"]))
d6 = dict([('Age', 19),   ('Name',   "Susan"),  ('Major',  "CS")])
```

Note: zip takes two equal-length collections and merges their corresponding elements into tuples.

Accessing the Dictionary

- To access a dictionary, simply index the dictionary by the key to obtain the value. An
- exception will be raised if the key is not in the dictionary

```
d1  =  {'Age':19, 'Name':"Susan",    'Major':"CS"}
print(d1['Age'])
print(d1['Name'])
```

Updating the Dictionary

- Simply assign a key:value pair to modify it or add a new pair. The del keyword can be used to delete a single key:value pair or the whole dictionary. The clear() method will clear the contents of the dictionary.

```
d1  =  {'Age':19, 'Name':"Susan",    'Major':"CS"}
d1['Age']  =  21
d1['Year'] =  "Junior"
print(d1)
del d1['Major']
print(d1)
d1.clear()
print(d1)
```

Some built-in Dictionary methods

```
d1={'Age':19,   'Name':"Susan",    'Major':"CS"}
print(d1.__contains__('Age'))  #  True   if key    exists
print(d1.__contains__('Year')) #  False  otherwise
print(d1.keys())   #  Return a  list   of keys
print(d1.items())  #  Return a  list   of key:value  pairs
print(d1.values()) #  Returns    a  list   of values
print(d1.pop('Age'))
print(d1)
print(d1.popitem())
print(d1)
print('Major' in d1)
print('Name' in d1)
print('Major' not in d1)
print('Name' not in d1)
```

Note: `in`, `not in`, `pop(`*key*`)`, **and** `popitem()` **are also supported.**

Ordered Dictionary

- Dictionaries do not remember the order in which keys were inserted. An ordered dictionary implementation is available in the collections module. The methods of a regular dictionary are all supported by the OrderedDict class.

- An additional method supported by OrderedDict is the following:

```
OrderedDict.popitem(last=True)   # pops   items  in     LIFO   order
```

Ordered Dictionary

```
#   regular    unsorted   dictionary
import collections
d  =  {'banana': 3, 'apple':   4, 'pear':    1, 'orange':  2}
#  dictionary sorted by key
d2 = collections.OrderedDict(sorted(d.items(), key=lambda t: t[0]))
print(d2)
#  dictionary sorted by value
d3 = collections.OrderedDict(sorted(d.items(), key=lambda t: t[1]))
print(d3)
#  dictionary sorted by length of the    key    string
d4 = collections.OrderedDict(sorted(d.items(), key=lambda t: len(t[0])))
print(d4)
```

Ordered Dictionary with custom comparison function

```
#  custom_comparison.py
import collections
Import functools

d  =  {'banana': 3, 'apple':   4, 'pear':    1, 'orange':  2, ‘kiwi’: 10}
def comp(a, b):
    if (len(a[0]) > len(b[0]):
        return 1
    elif len(a[0]) < len(b[0]):
        return -1
    else :
        if (a[1] > b[1]):
           return 1
        else:
           return -1

#  dictionary sorted by key
d2 = collections.OrderedDict(sorted(d.items(), key=functools.cmp_to_key(comp)))
print(d2)
```

File Operations: open/close

- A file must be open before other operations
  - file_object = open(file_name [, access_mode][, buffering])
  - Access mode: *r* for read, *w* for write, r+ for read and write, etc.
  - Buffering usually uses default
- A file object has several objects such as name, closed, mode
- Opened file should be closed.
  - close() flushes buffered info and close the file object – no more write can be done.
  - Without close(), file may not be in the final state.

```
#lect4/open.py
fobj = open('open.py', 'r+')
print(f'File name: {fobj.name}')
print(f'Access mode: {fobj.mode}')
print(f'Closed? {fobj.closed}")
fobj.close()
print(f'File name: {fobj.name}')
print(f'Access mode: {fobj.mode}')
print(f'Closed? {fobj.closed}")
```

File Operations: read/write

- The *write()* method: *fileojb.write(string)*
  - Writes a string to an open file. Note that Python string can be binary data, not just text.
  - Write does not add '\n' at the end of the string.
- The *read()* method: *fileojb.read([count])*
  - Count is the number of bytes to read.
  - Without count, read the whole file

```
#lect4/write.py
f = open('foo.txt', 'w')
f.write('Python is great!')
f.write('I love Python!')
f.close()
```

```
#lect4/read.py
f = open('open.py', 'r')
s = f.read()
f.close()
print(s)
```

File Operations: tell/seek

- The *tell()* method: *fileojb.tell()*
  - Tell the current position within the file (the next read or write will start from the position)
- The *seek()* method: *fileojb.seek(offset, [from])*
  - Move the current position to the specified position.
  - From = 0 - beginning of the file, 1 – current position, 2 – end of the file.
    - f.seek(0, 0) – move to the beginning of the file
    - f.seek(0, 2) – move to the end of the file

```
#lect4/seek.py
f = open('foo.txt', 'w')
f.write('Python is great!')
f.write('I love Python!')
print(f.tell())
f.seek(0, 0)
f.write('XXXX')
print(f.tell())
f.close()
```

Managing file access using with

- An open file must be closed, but often after multiple file operations.
- Enclosing file open in the with block, the file will be close when the block finishes execution.

```
# lect4/with
with open ('open.txt', 'r') as f:
    for line in f:
        print(line.strip())
```