DS5010

Intro to programming for Data Science

LECTURE 2

TODAY

- review session 1
- iterations with for/while loops
- lists, tuples, dictionaries
- mutable and immutable objects
- read/write files

REVIEW

- primitive objects and operators
- variables and types
- rules of sum and product
- branching and conditionals

CONTROL FLOW: `while` LOOPS

```
while <condition>:
    <expression>
    <expression>
    ...
```

- `<condition>` evaluates to a Boolean
- if `<condition>` is `True`, do all the steps inside the while code block
- check `<condition>` again
- repeat until `<condition>` is `False`

`while` LOOP EXAMPLE

Guess and check game

- Sansa as the programmer generates a random integer between 0 – 20.
- Joffrey as the user will input his guess.
- if his guess matches Sansa's number, a success message will be printed.
- otherwise, Joffrey has to keep guessing.

```
n = random.randint(0, 20)
g = int(input('Guess what number is in my mind?'))
while n != g:
    g = int(input('Guess what number is in my mind?'))
print('That was right!!')
```

CONTROL FLOW: `while` and `for` LOOPS

- iterate through numbers in a sequence

```
# more complicated with while loop
n = 0
while n < 5:
    print(n)
    n = n+1

# shortcut with for loop
for n in range(5):
    print(n)
```

CONTROL FLOW: `for` LOOPS

```
for <variable> in range(<some_num>):
    <expression>
    <expression>
    ...
```

- each time through the loop, <variable> takes a value
- first time, <variable> starts at the smallest value
- next time, <variable> gets the prev value + 1
- etc.

range(start,stop,step)

- default values are `start = 0` and `step = 1` and optional
- loop until value is `stop -1`

```
mysum = 0
for i in range(7, 10):
    mysum += i
print(mysum)

mysum = 0
for i in range(5, 11, 2):
    mysum += i
print(mysum)
```

`break` STATEMENT

- immediately exits whatever loop it is in
- skips remaining expressions in code block
- exits only innermost loop!

```
while <condition_1>:
    while <condition_2>:
        <expression_a>
        break
        <expression_b>
     <expression_c>
```

`break` STATEMENT

```
mysum= 0
for i in range(5, 11, 2):
    mysum += i
    if mysum == 5:
        break
    mysum += 1
print(mysum)
```

- what happens in this program?

`for` VS `while` LOOPS

`for loops`

- **know** number of iterations
- can **end early** via `break`
- uses a **counter**
- **can rewrite** a `for` loop using a `while` loop

`while loops`

- **unbounded** number of iterations
- can **end early** via `break`
- can use a **counter but must initialize** before loop and increment it inside loop
- **may not be able to rewrite** a `while` loop using a `for` loop

TUPLES

- an ordered sequence of elements, can mix element types.
- cannot change element values, **immutable**
- represented with parentheses

remember strings?

```
te= ()
```
empty tuple

```
t = (2,"stark",3)
t[0]                    → evaluates to 2
(2,"stark",3) + (5,6)   → evaluates to (2,"stark",3,5,6
t[1:2]                  → slice tuple, evaluates to ("stark",)
t[1:3]                  → slice tuple, evaluates to ("stark",3
len(t)                  → evaluates to 3
t[1] = 4                → gives error, can't modify object
```

extra comma means a tuple with one element

TUPLES

- conveniently used to **swap** variable values

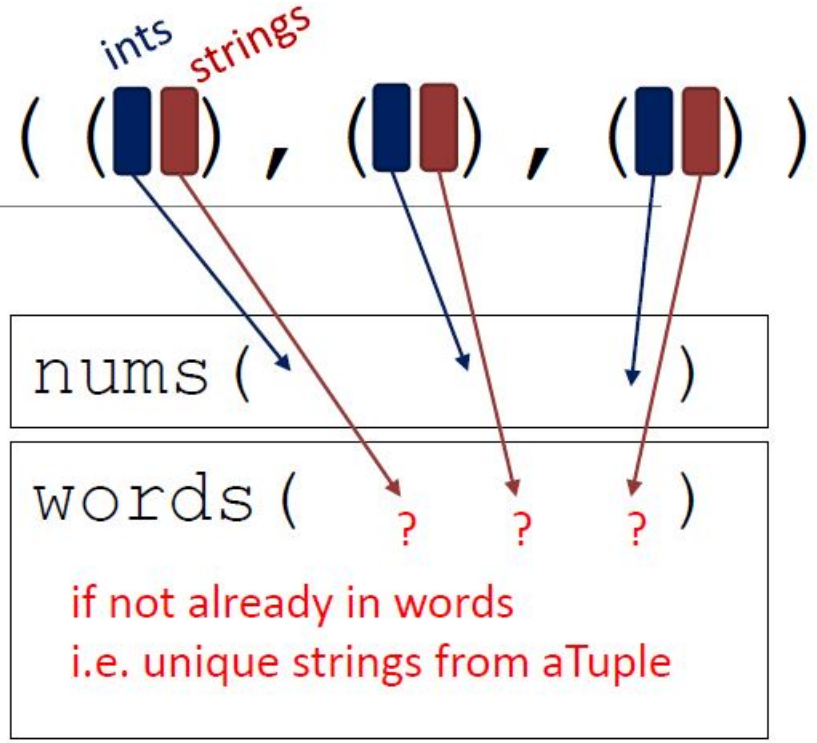
```
x = y
y = x
```

```
temp = x
x = y
y = temp
```

```
(x, y) = (y, x)
```

- used to **return more than one value** from a function (will return to this when learning about functions)

MANIPULATING TUPLES



- can **iterate** over tuples

empty tuple

singleton tuple

```
nums = ()
words = ()
for t in aTuple:
    nums = nums + (t[0],)
    if t[1] not in words:
        words = words + (t[1],)
min_n = min(nums)
max_n = max(nums)
unique_words_count = len(words)
print(min_n, max_n, unique_words_count)
```

LISTS

- **ordered sequence** of information, accessible by index
- a list is denoted by **square brackets**, [ ]
- a list contains **elements**
  - usually homogeneous (i.e, all integers)
  - can contain mixed types (not common)
- list elements can be changed so a list is **mutable**

INDICES AND ORDERING

```
a_list= []
```
empty list

```
L = [2, 'a', 4, [1,2]]
```

`len(L)` → evaluates to 4

`L[0]` → evaluates to 2

`L[2]+1` → evaluates to 5

`L[3]` → evaluates to `[1,2]`, another list!

`L[4]` → gives an error

```
i= 2
```

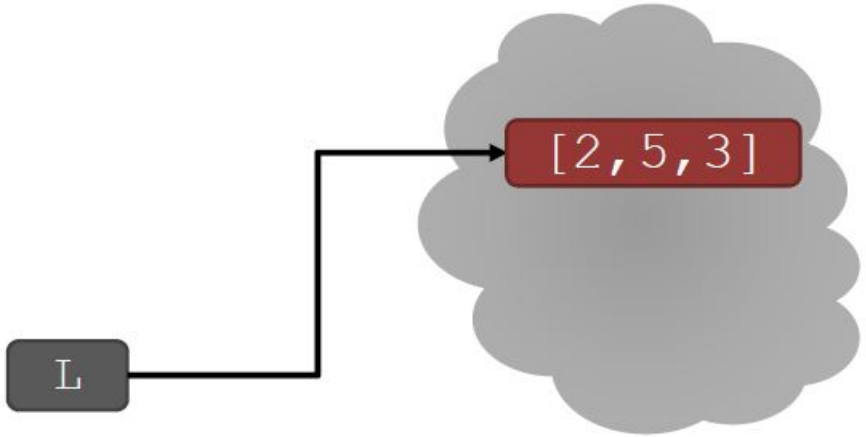
`L[i-1]` → evaluates to 'a' since `L[1]='a'` above

CHANGING ELEMENTS

- lists are **mutable**!
- assigning to an element at an index changes the value at that index

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

- `L` is now `[2, 5, 3]`, note this is the **same object** `L`



ITERATING OVER A LIST

- compute the **sum of elements** of a list
- common pattern, iterate over list elements

```
total = 0
for i in range(len(L)):
    total += L[i]
print(total)
```

```
total = 0
for e in L:
    total += e
print(total)
```

like strings, can iterate over list elements directly

- Notice list elements are indexed `0` to `len(L)-1`
- `range(n)` goes from `0` to `n-1`

OPERATIONS ON LISTS -ADD

- **add** elements to end of list with `L.append(element)`
- **mutates** the list!

```
L = [2,1,3]
L.append(5)
```
→ L is now `[2,1,3,5]`

- what is the dot?
  - lists are Python objects, everything in Python is an object
  - objects have data
  - objects have methods and functions
  - access this information by object_name.do_something()
  - will learn more about these later

OPERATIONS ON LISTS -ADD

- to combine lists together use **concatenation**, + operator, to give you a new list
- **mutate** list with `L.extend(some_list)`

```
L1 = [2,1,3]
L2 = [4,5,6]
L3 = L1 + L2
```

→ `L3` is `[2,1,3,4,5,6]`
`L1`, `L2` unchanged

```
L1.extend([0,6])
```

→ mutated `L1` to `[2,1,3,0,6]`

OPERATIONS ON LISTS -REMOVE

- delete element at a **specific index** with `del(L[index])`
- remove element at **end of list** with `L.pop()`, returns the removed element
- remove a **specific element** with `L.remove(element)`
  - looks for the element and removes it
  - if element occurs multiple times, removes first occurrence
  - if element not in list, gives an error

```
L = [2,1,3,6,3,7,0] # do below in order
```

all these operations mutate the list

```
L.remove(2) → mutates L = [1,3,6,3,7,0]
L.remove(3) → mutates L = [1,6,3,7,0]
del(L[1])   → mutates L = [1,3,7,0]
L.pop()     → returns 0 and mutates L = [1,3,7]
```

CONVERT LISTS TO STRINGS AND BACK

- convert string to list with `list(s)`, returns a list with every character from `s` an element in `L`
- can use `s.split()`, to split a string on a character parameter, splits on spaces if called without a parameter
- use `''.join(L)` to turn a list of characters into a string, can give a character in quotes to add char between every element

`s = "I<3 DS"` → `s` is a string

`list(s)` → returns `['I','<','3',' ','D','S']`

`s.split('<')` → returns `['I', '3 DS']`

`L = ['a','b','c']` → `L` is a list

`''.join(L)` → returns `"abc"`

`'_'.join(L)` → returns `"a_b_c"`

OTHER LIST OPERATIONS

- sort() and sorted()
- reverse() and reversed()
- and many more!
  https://docs.python.org/3/tutorial/datastructures.html

```
L=[9,6,0,3]
```

`sorted(L)` → returns sorted list, does **not mutate** `L`

`L.sort()` → **mutates** `L=[0,3,6,9]`

`reversed(L)` → returns a reversed list, does **not mutate** `L`

`L.reverse()` → **mutates** `L=[9,6,3,0]`

LISTS IN MEMORY

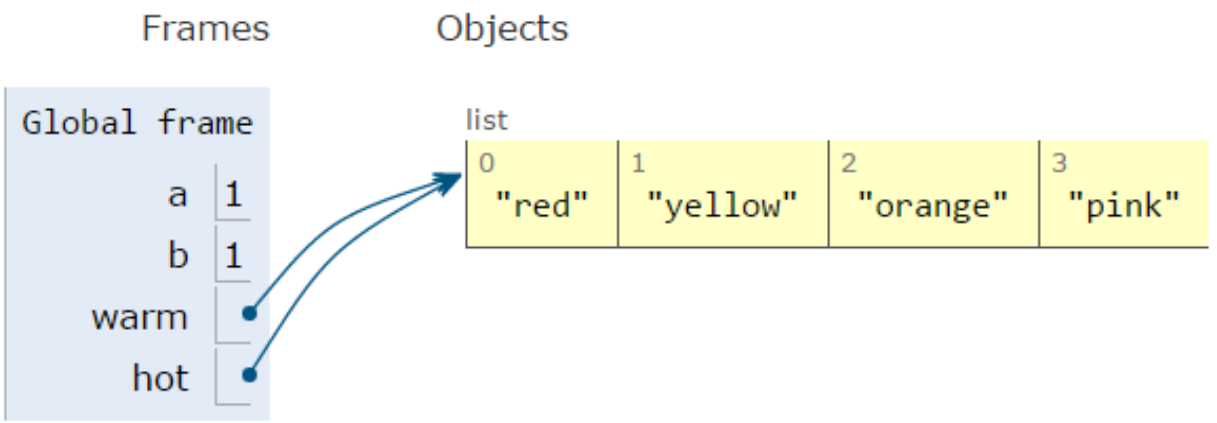
- lists are **mutable**
- behave differently than immutable types
- is an object in memory
- variable name points to object
- any variable pointing to that object is affected
- key phrase to keep in mind when working with lists is **side effects**

ALIASES

- `hot` is an **alias** for `warm`–changing one changes the other!
- `append()` has a side effect

```
1  a = 1
2  b = a
3  print(a)
4  print(b)
5
6  warm = ['red', 'yellow', 'orange']
7  hot = warm
8  hot.append('pink')
9  print(hot)
10 print(warm)
```

```
1
1
['red', 'yellow', 'orange', 'pink']
['red', 'yellow', 'orange', 'pink']
```

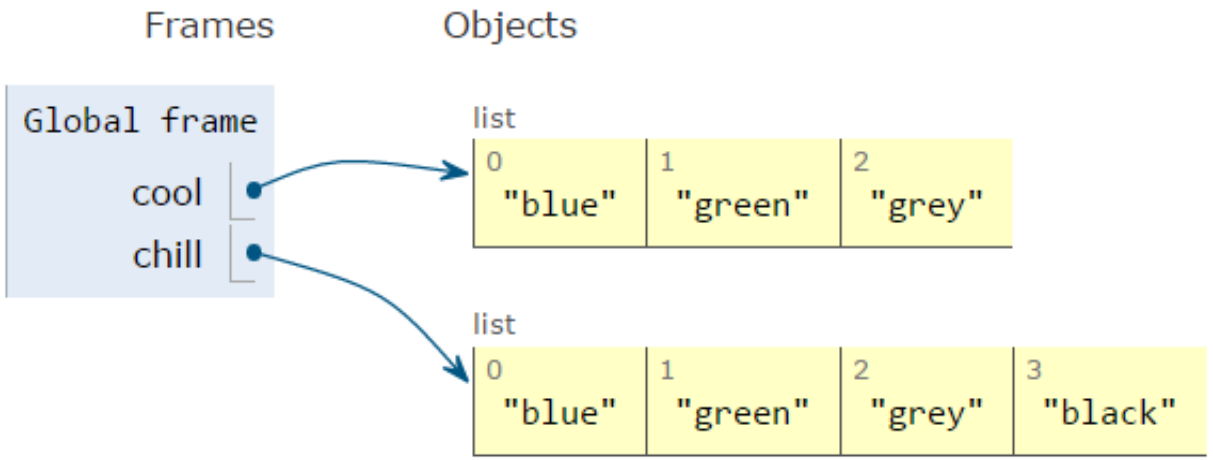


CLONING A LIST

- create a new list and **copy every element** using `chill = cool[:]`

```
1  cool = ['blue', 'green', 'grey']
2  chill = cool[:]
3  chill.append('black')
4  print(chill)
5  print(cool)
```

```
['blue', 'green', 'grey', 'black']
['blue', 'green', 'grey']
```

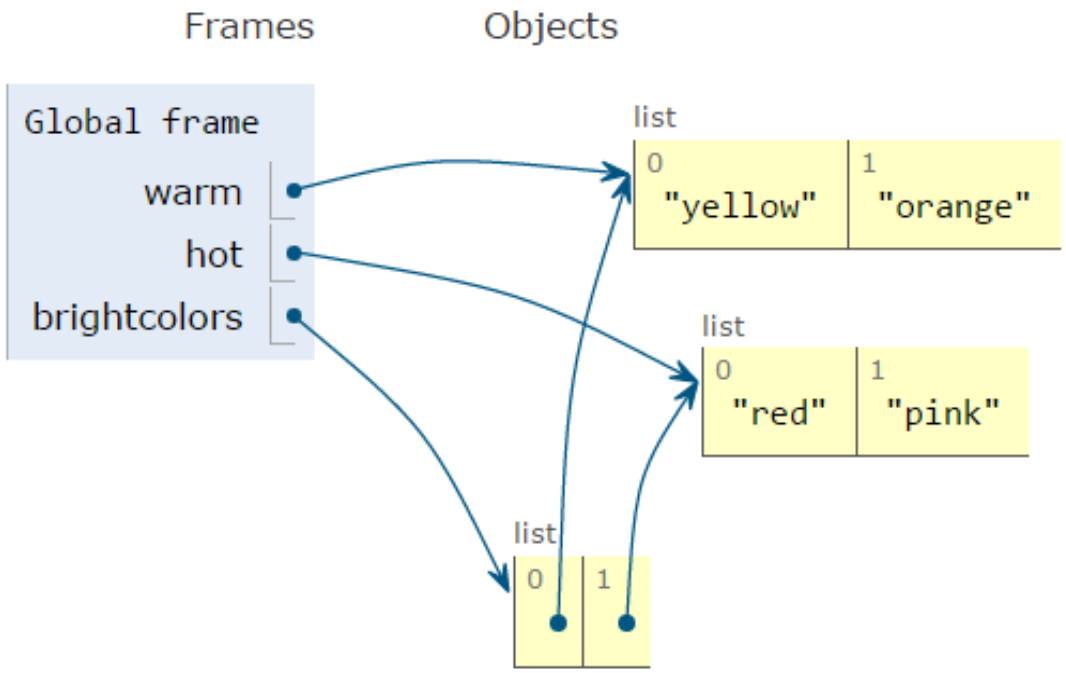


LISTS OF LISTS OF LISTS OF….

- can have **nested** lists
- side effects still
  possible after mutation

```
1  warm = ['yellow', 'orange']
2  hot = ['red']
3  brightcolors = [warm]
4  brightcolors.append(hot)
5  print(brightcolors)
6  hot.append('pink')
7  print(hot)
8  print(brightcolors)
```

```
[['yellow', 'orange'], ['red']]
['red', 'pink']
[['yellow', 'orange'], ['red', 'pink']]
```



MUTATION AND ITERATION

- **avoid** mutating a list as you are iterating over

```
L1 = [1, 2, 3, 4]
L2 = [1, 2, 5, 6]

for e in L1:
    if e in L2:
        L1.remove(e)
```

```
L1_copy = L1[:]
for e in L1_copy:
    if e in L2:
        L1.remove(e)
```

`L1` is `[2,3,4]` not `[3,4]` Why?

clone list first, note that `L1_copy = L1` does NOT clone

- Python uses an internal counter to keep track of index it is in the loop
- mutating changes the list length but Python doesn't update the counter
- loop never sees element 2

DICTIONARY

- so far, can store using separate lists for every info

```
names = ['Arya', 'Tyrion', 'Robert', 'Elia']
grade = ['A+', 'A', 'A-', 'A']
course = [5800, 5010, 5020, 7600]
```

- a **separate list** for each item
- each list must have the **same length**
- info stored across lists at **same index**, each index refers to info for a different person

HOW TO UPDATE/RETRIEVE STUDENT INFO

```
student = input()
i = names.index(student)
grade = grades[i]
course = courses[i]
print(course, grade)
```

- **messy** if have a lot of different info to keep track of
- must **always index** using integers
- must remember to change multiple lists

A BETTER AND CLEANER WAY – A DICTIONARY

- nice to **index item of interest directly** (not always int)
- nice to use **one data structure**, no separate lists

A list

0	Elem 1
1	Elem 2
2	Elem 3
3	Elem 4
...	...

index element

A dictionary

Key 1	Val 1
Key 2	Val 2
Key 3	Val 3
Key 4	Val 4
...	...

custom index by label element

A PYTHON DICTIONARY

- store pairs of data
  - key
  - value

'Arya'	'A+'
'Tyrion'	'A'
'Robert'	'A-'
'Elia'	'A'

custom index by label

element

empty dictionary

```
my_dict = {}
```

```
grades = {'Arya':'A+', 'Tyrion':'A', 'Robert':'A-', 'Elia':'A'}
```

key1 val1 key2 val2 key3 val3 key4 val4

DICTIONARY LOOKUP

- similar to indexing into a list
- **looks up** the **key**
- **returns** the **value** associated with the key
- if key isn't found, get an error

'Arya'	'A+'
'Tyrion'	'A'
'Robert'	'A-'
'Elia'	'A'

```
grades = {'Arya':'A+', 'Tyrion':'A', 'Robert':'A-', 'Elia':'A'}
```

`grades['Arya']` → evaluates to `'A+'`

`grades['Sylvan']` → gives a `KeyError`

DICTIONARY OPERATIONS

- **add** an entry

```
grades['Theon'] = 'B'
```

- **test** if key in dictionary

```
'Robert' in grades → returns True
'Daniel' in grades → returns False
```

- **delete** entry

```
del(grades['Arya'])
```

'Arya'	'A+'
'Tyrion'	'A'
'Robert'	'A-'
'Elia'	'A'

DICTIONARY OPERATIONS

```
grades = {'Arya':'A+', 'Tyrion':'A', 'Robert':'A-', 'Elia':'A'}
```

- get an **iterable that acts like a tuple of all keys**

```
grades.keys() → returns ['Arya', 'Tyrion', 'Robert', 'Elia’]
```

- get an **iterable that acts like a tuple of all values**

```
grades.values() →  returns ['A+', 'A', 'A-', 'A']
```

DICTIONARY KEYS and VALUES

- values
  - any type **(immutable and mutable)**
  - can be **duplicates**
  - dictionary values can be lists, even other dictionaries!
- keys
  - must be **unique**
  - **immutable** type (`int, float, string, tuple, bool`)
    - actually need an object that is **hashable,** but think of as immutable as all immutable types are hashable
  - careful with `float` type as a key
- **no order** to keys or values! (As of python 3.6+ insertion order is guaranteed by language)

```
d = {4:{1:0}, (1,3):"twelve", 'const':[3.14,2.7,8.44]}
```

`list` vs `dict`

- **ordered** sequence of elements
- look up elements by an integer index
- indices have an **order**
- index is an **integer**

- **matches** “keys” to “values”
- lookup one item by another item
- **no order** is guaranteed (as of python 3.6+ insertion order is guaranteed)
- key can be any **immutable** type

READING FILES

What you need in order to **read** information from a file

1. **open** the file and associate the file with a variable.
2. a command to **read** the information.
3. a command to **close** the file.

READING FILES

- use open statement to open a file for reading:

```
f = open('mydata.txt')
```

f now points to a file object

It will give error If file does not exist

- you can read the **entire** content as a string

```
content = f.read()
```

- or, read just one line of the file at a time

```
line = f.readline()
```

- or, read all the lines, as a list of lines

```
lines = f.readlines()
```

- don't forget to close the file after reading is finished

```
f.close()
```

ITERATING THROUGH LINES

- you can iterate through the lines of a file using a `for` loop

```
f = open('mydata.txt')

for line in f:
    <statement>

f.close()
```

USING **with** PATTERN

- we always forget to close the files, after we are finished using them
- a good pattern is to use **with** statements in python so the close operation will be done automatically.

```
with open('mydata.txt') as f:
    for line in f:
        <statements>
```

WRITING INTO FILES

What you need in order to **read** information from a file

1. **open** the file and associate the file with a variable.
2. decide whether you want to delete all the preexisting content and write fresh or you want to append at the end of the file
3. a command to **write** into the file.
4. a command to close the file.

WRITING AS A NEW FILE

- use open statement with ‘w’ option to open the file for writing as new file

```
f = open('data.txt', 'w')
```

the file will be created or overridden if it already exists.

- write content into file using **write** method

```
f.write('blah blah blah')
```

it only accepts strings

- **close** the file so the content that you have written gets saved

```
f.close()
```

don’t forget to write '\n' if you want to continue to a new line.

APPENDING TO A FILE

- the same as before, just open the file with `'a'` option.

```
f = open('data.txt', 'a')
```

USING **with** PATTERN

- a good pattern is to use **with** statements in python so the close operation will be done automatically.
- Assume `new_lines` is a list of strings, which we want to append each as a new line in `data.txt`

```
with open('data.txt', 'w') as f:
    for l in new_lines:
        f.write(l)
        f.write('\n')
```