Introduction to PYTHON

Demystifying the World of Artificial Intelligence and Exploring Its Potential

TUPLES

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

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

empty tuple

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)
```

- conveniently used to **swap** variable values

```
def quotient_and_remainder(x, y):
    q = x // y
    r = x % y
    return (q, r)
(quot, rem) = quotient_and_remainder(4,5)
```

integer division

LISTS

- **ordered sequence** of information, accessible by index
- a list is denoted by **square brackets**, []
- a list contains **elements**
  - usually homogeneous (ie, 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
```

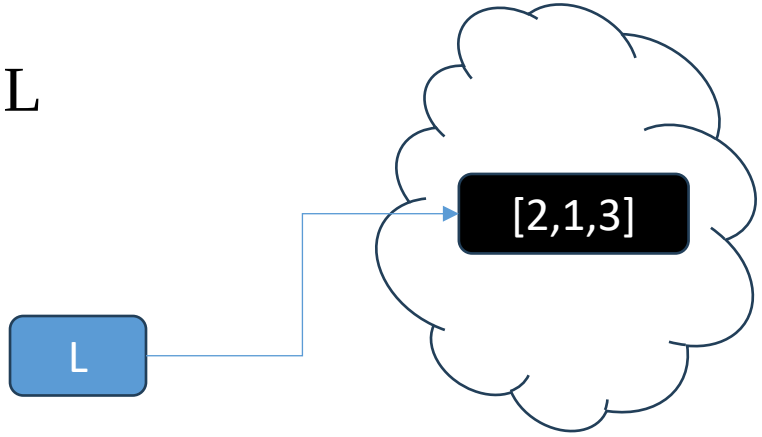
`L[i-1]` → evaluates to ‘a’ since `L[1]='a'` above

CHANGING ELEMENTS

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

```
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

like strings, can iterate over list elements directly

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

```
total = 0
  for i in L:
      total += i
  print total
```

- **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

all these operations mutate the list

```
L = [2,1,3,6,3,7,0] # do below in order
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 cs"
list(s)
s.split('<')
L = ['a','b','c']
''.join(L)
'_'.join(L)
```

→ s is a string
→ returns `['I','<','3',' ','c','s']`
→ returns `['I', '3 cs']`
→ L is a list
→ returns `"abc"`
→ returns `"a_b_c"`

OTHER LIST OPERATIONS

- sort() and sorted()
- reverse()
- and many more! **HERE**

```
L=[9,6,0,3]
sorted(L)
L.sort()
L.reverse()
```

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

→ **mutates** `L=[0,3,6,9]`

→ **mutates** `L=[9,6,3,0]`

List operators

- + is used to concatenate lists:

```
lst1 = ['Toronto', 'Montreal', 'Vancouver']
lst2 = ['Ottawa', 'London', 'Guelph']
print(lst1 + lst2)
```

```
['Toronto', 'Montreal', 'Vancouver', 'Ottawa', 'London', 'Guelph']
```

- * is used for repetition:

```
lst = ['CS1910','ENGN2020']
print(lst*4)
```

```
['CS1910', 'ENGN2020', 'CS1910', 'ENGN2020', 'CS1910', 'ENGN2020', 'CS1910', 'ENGN2020']
```

List operators

- **in** and **not in** are used to check membership:

```
lst = ['Toronto','Montreal','Vancouver','Ottawa','London']
print('Toronto' in lst)
print('Montreal' not in lst)
```

```
True
False
```

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**

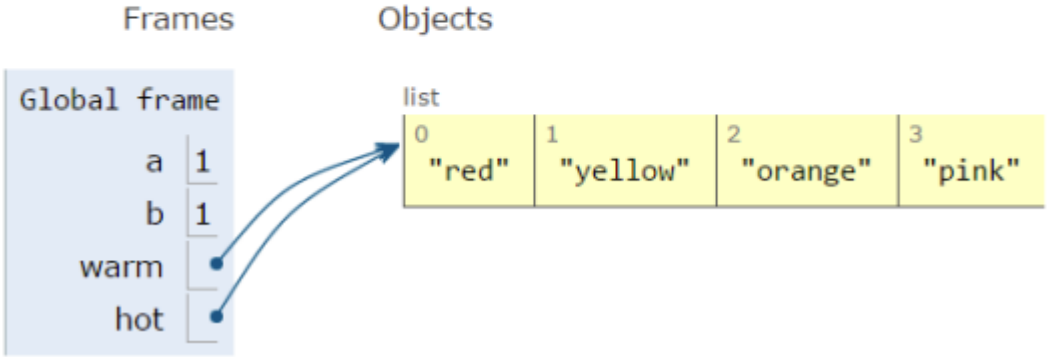
ALIASES

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

```
a = 1
b = a
print(a)
print(b)

warm = ['red', 'yellow', 'orange']
hot = warm
hot.append('pink')
print(hot)
print(warm)
```

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



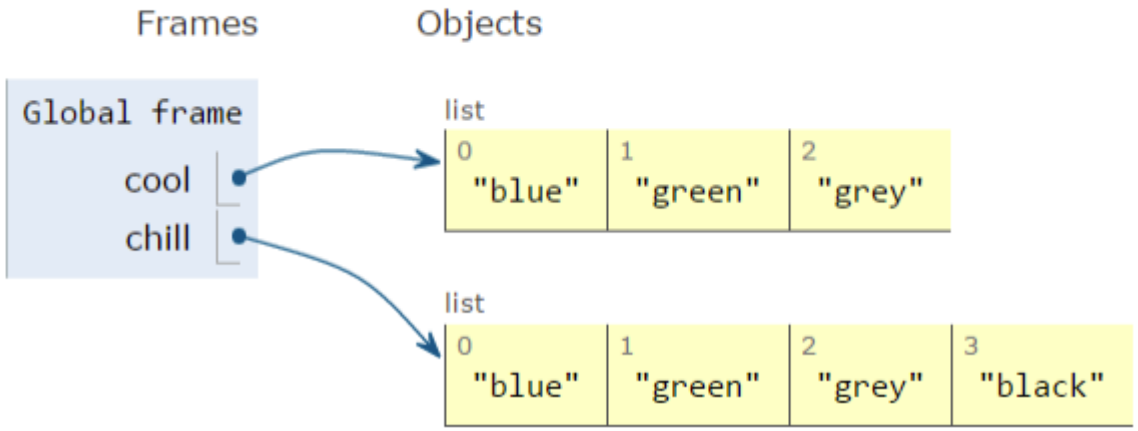
CLONING A LIST

- create a new list and **copy every element** using

  chill = cool[:]

```
cool = ['blue', 'green', 'grey']
chill = cool[:]
chill.append('black')
print(chill)
print(cool)
```

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



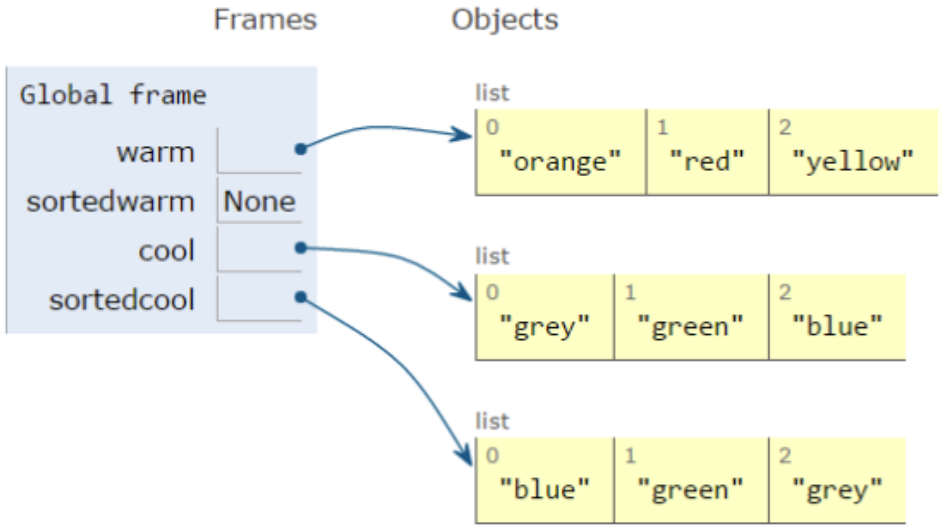
SORTING LISTS

- calling sort() **mutates** the list, returns nothing
- calling sorted() **does not mutate** list, must assign result to a variable

```
['orange', 'red', 'yellow']
None
['grey', 'green', 'blue']
['blue', 'green', 'grey']
```

```
warm = ['red', 'yellow', 'orange']
sortedwarm = warm.sort()
print(warm)
print(sortedwarm)

cool = ['grey', 'green', 'blue']
sortedcool = sorted(cool)
print(cool)
print(sortedcool)
```

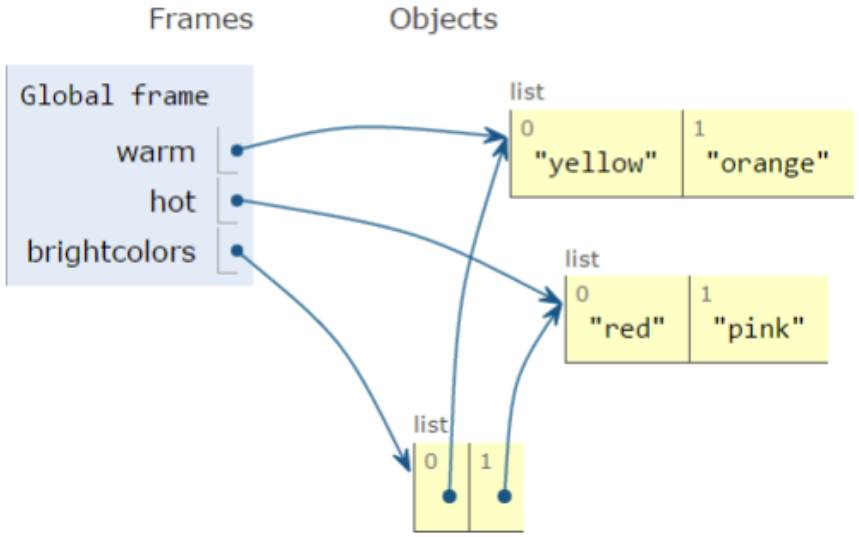


LISTS OF LISTS OF LISTS OF….

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

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

```
warm = ['yellow', 'orange']
hot = ['red']
brightcolors = [warm]
brightcolors.append(hot)
print(brightcolors)
hot.append('pink')
print(hot)
print(brightcolors)
```



MUTATION AND ITERATION

Try this in Python!

- avoid mutating a list as you are iterating over it

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

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

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

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

MUTATION AND ITERATION

Try this in Python!

- avoid mutating a list as you are iterating over it

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

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

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

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

- L1 **is** [2,3,4] **not** [3,4] **Why**?
  - 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

DICTIONARIES

HOW TO STORE STUDENT INFO

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

```
names = ['Ana', 'John', 'Denise', 'Katy']
grade = ['B', 'A+', 'A', 'A']
course = [2.00, 6.0001, 20.002, 9.01]
```

- 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

DICTIONARIES

HOW TO UPDATE/RETRIEVE STUDENT INFO

```
def get_grade(student, name_list, grade_list, course_list):
    i = name_list.index(student)
    grade = grade_list[i]
    course = course_list[i]
    return (course, grade)
```

DICTIONARIES

HOW TO UPDATE/RETRIEVE STUDENT INFO

```python
def get_grade(student, name_list, grade_list, course_list):
    i = name_list.index(student)
    grade = grade_list[i]
    course = course_list[i]
    return (course, grade)
```

- **messy** if have a lot of different info to keep track of
- must maintain **many lists** and pass them as arguments
- must **always index** using integers
- must remember to change multiple lists

DICTIONARIES

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

DICTIONARIES

A PYTHON DICTIONARY

- store pairs of data
  - key
  - value

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

custom index by label

element

empty dictionary

```
my_dict = {}
```

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

key1 val1 key2 val2 key3 val3 key4 val4

DICTIONARIES

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

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
grades['John']        → evaluates to 'A+'
grades['Sylvan']      → gives a KeyError
```

DICTIONARIES

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'
'Sylvan'	'A'

DICTIONARY OPERATIONS

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

- add an **entry**

```
grades['Sylvan'] = 'A'
```

- **test** if key in dictionary

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

- **delete** entry

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

DICTIONARIES

DICTIONARY OPERATIONS

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

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

  `grades.keys()` → returns `['Denise','Katy','John','Ana']`

  no guaranteed order

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

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

  no guaranteed order

DICTIONARIES

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**
  - careful with *float* type as a key

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"
- look up one item by another item
- **no order** is guaranteed
- key can be any **immutable** type

THANK YOU!

Any Question!