COMP 204: Computer programming for Life Sciences

Python programming: Lists

Mathieu Blanchette
based on material from Yue Li, Christopher J.F. Cameron and Carlos G. Oliver

The need for compound data types

Until now, our variable could only hold one value at a time... except for Strings, which is a sequence of many characters.

This is limiting. If we want to store 1000 numbers, we would need 1000 variables!

```
weight0 = 45.6
weight1 = 12.3
weight2 = 24.5
...
weight998 = 45.2
weight999 = 42.4
```

And what if we don't know the number of elements ahead of time?

Compound data types - Lists and Tuples

Compound types allow us to store multiple values in one variable. The most basic compound type is called a **Sequence**. There are many types of Sequences:

- Strings: Specifically for chains of characters
- Lists:
  - Ordered collection of objects of *any number of objects* of *any types*
  - Mutable: They can grow or shrink, and their content can be modified
  - Useful when the number of objects to be stored is not known ahead of time
- Tuples:
  - Ordered collection of objects of a *fixed number* of objects of *any types*
  - Immutable: Once created, a tuple cannot be modified. Returns new objects when attempting to update
  - Useful when the number of objects to be stored is known ahead of time
  - Allows faster operations than lists

Lists and Tuples - examples

A *list* is created using *square brackets*, with items separated by commas

A *tuple* is created using *parentheses*, with items separated by commas.

```
# a list of 5 integers
ages = [10, 20, 30, 40, 50]

# a list of 3 strings
names = ["Sarah","John","Mary"]

# a list of both strings and integers
mixed = ["Bill", 50, "Amy", 32, "Roger", 76]

# an empty list
L = [ ]


# Example of tuples:

# a tuple of 3 float
xyz = (0.3, -0.5, 1.2)

# a tuple of one string and one integer
carbon = ("C",12)
```

Lists and Tuples - more examples

The elements of lists or tuples can themselves be objects of compound types!

```
# a list of tuples (atom, mass)
periodicTable = [ ("H",1), ("C",12), ("N", 14) ]

# a list of lists
molecules = [ ["C","O","O"], ["N","O"], ["O","O"] ]

# a list of tuples, where each tuple is a
# pair of a String and a list of Strings
moleculesWithNames = [ ("carbon dioxyde", ["C","O","O"]),
                       ("nitrous oxyde", ["N","O"] ) ]
```

Accessing elements of Lists or Tuples - indexing

Like for Strings, we can access elements of lists or tuples by *indexing*.

Note: this example uses a List, but the same works for a Tuple.

```
names = ["Sarah", "Zheng", "Amol", "Vladimir","Juanita"]

firstName = names[0]    # Sarah
secondName = names[1]   # Zheng
lastName = names[4]     # Juanita

nbNames = len(names)   # 5

lastName = names[ nbNames -1 ]  # Still Juanita
wrong = names[ nbNames ]  # Error: list index out of range

penultimateName = names [ nbNames - 2]  # Vladimir

lastName = names[-1]     # Juanita
penultimateName = names[-2]     # Vladimir

someNames = names[1:3]  # ["Zheng", "Amol"]
allButFirst = names[1:5]  # ["Zheng", "Amol", "Vladimir","Juanita"]
allButLast = names[0:4]  # ["Sarah", "Zheng", "Amol", "Vladimir" ]
```

Accessing elements of Lists or Tuples - indexing

We can also access values within a nested list

```
# a list of tuples (atom, mass)
periodicTable = [ ("H",1), ("C",12), ("N", 14) ]

# a list of lists
molecules = [ ["C","O","O"], ["N","O"], ["O","O"] ]

# a list of tuples, where each tuple is a
# pair of a String and a list of Strings
moleculesWithNames = [ ("carbon dioxyde", ["C","O","O"]),
                       ("nitrous oxyde", ["N","O"] ) ]

# indexing tuple value in a list
periodicTable[1][1] # 12

# indexing list value inside a tuple inside another list
moleculesWithNames[1][1][0] # "N"
```

Modifying the content of a List

Because lists are *mutable*, their content can be modified.

```python
names = ["Sarah", "Zheng", "Amol", "Vladimir","Juanita"]

names[1] = "Lin"  # Zheng is replaced by Lin
names[4] = "Consuela"  # Juanita is replaced by Consuela

names[5] = "John"  # Error: Index of out range

# we can replace multiple elements of the list at once
names[2:4] = ["Prakash", "Boris"]

# or replace a portion of a list with another one
names[2:4] = ["Prakash", "Boris", "John", "Paul"]
```

Note: This would not work on tuples, because they are immutable.

Assigning by copy versus assigning by reference

Assignments of values to variables behaves differently for simple and compound types:

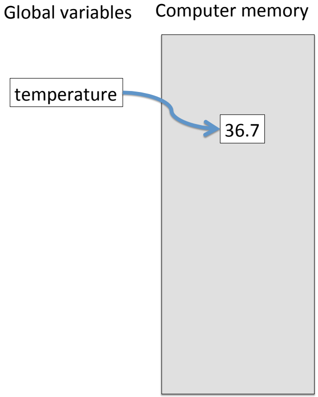
Assigning by copy

- For simple types (int, float, boolean), writing b = a creates a new variable b, separate from a, whose value is set to that of a

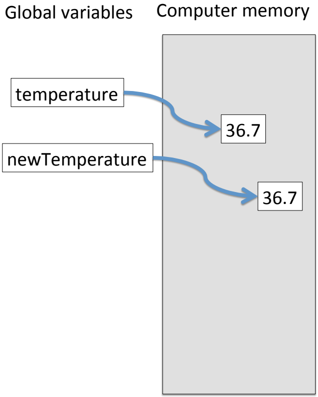
Assigning by reference

- For compound types (lists, tuples, and more), writing b = a creates a new variable b that refers to the same compound object as a. Modifying the content of a also modifies the content of b.

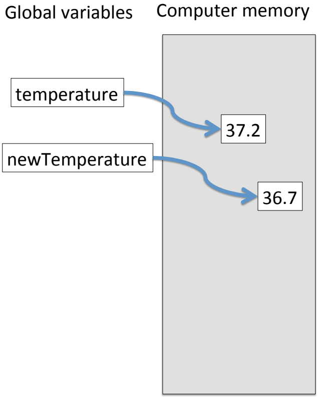
```
# example with a simple type (e.g. float)
temperature = 36.7
newTemperature = temperature
print(temperature, newTemperature) # 36.7 36.7
temperature = 37.2
print(temperature, newTemperature) # 37.2 36.7


# example with a compound type (e.g. list)
names = ["Sarah", "Zheng", "Amol"]
otherNames = names
names[1] = "Lin" # Zheng is replaced by Lin
print(names, otherNames) # ["Sarah", "Lin", "Amol"]
                         # ["Sarah", "Lin", "Amol"]
otherNames[2] = "Ahmed"
print(names, otherNames) # ["Sarah","Lin","Ahmed"]
                         # ["Sarah","Lin","Ahmed"]
```



```
# example with a simple type (e.g. float)
temperature = 36.7
newTemperature = temperature
print(temperature, newTemperature) # 36.7 36.7
temperature = 37.2
print(temperature, newTemperature) # 37.2 36.7


# example with a compound type (e.g. list)
names = ["Sarah", "Zheng", "Amol"]
otherNames = names
names[1] = "Lin"  # Zheng is replaced by Lin
print(names, otherNames) # ["Sarah", "Lin", "Amol"]
                         # ["Sarah", "Lin", "Amol"]
otherNames[2] = "Ahmed"
print(names, otherNames) # ["Sarah","Lin","Ahmed"]
                         # ["Sarah","Lin","Ahmed"]
```

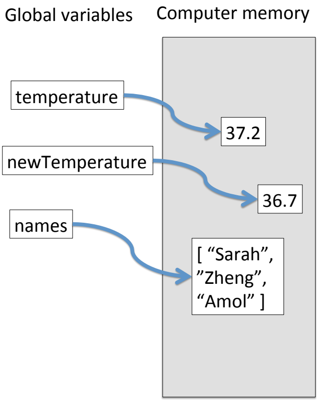


```
# example with a simple type (e.g. float)
temperature = 36.7
newTemperature = temperature
print(temperature, newTemperature) # 36.7 36.7
temperature = 37.2
print(temperature, newTemperature) # 37.2 36.7


# example with a compound type (e.g. list)
names = ["Sarah", "Zheng", "Amol"]
otherNames = names
names[1] = "Lin"  # Zheng is replaced by Lin
print(names, otherNames) # ["Sarah", "Lin", "Amol"]
                         # ["Sarah", "Lin", "Amol"]
otherNames[2] = "Ahmed"
print(names, otherNames) # ["Sarah","Lin","Ahmed"]
                         # ["Sarah","Lin","Ahmed"]
```

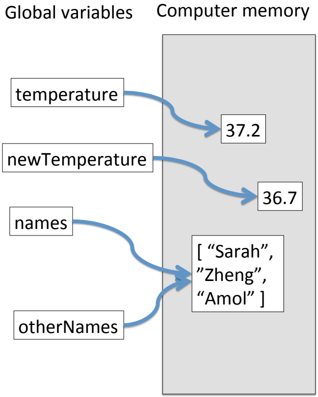


```
# example with a simple type (e.g. float)
temperature = 36.7
newTemperature = temperature
print(temperature, newTemperature) # 36.7 36.7
temperature = 37.2
print(temperature, newTemperature) # 37.2 36.7
```

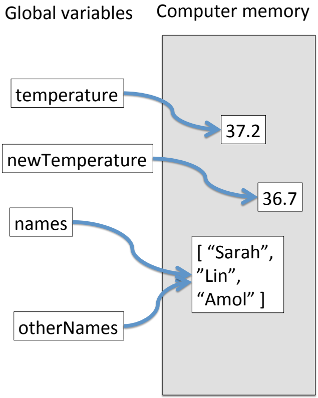
```
# example with a compound type (e.g. list)
names = ["Sarah", "Zheng", "Amol"]
otherNames = names
names[1] = "Lin" # Zheng is replaced by Lin
print(names, otherNames) # ["Sarah", "Lin", "Amol"]
                         # ["Sarah", "Lin", "Amol"]
otherNames[2] = "Ahmed"
print(names, otherNames) # ["Sarah","Lin","Ahmed"]
                         # ["Sarah","Lin","Ahmed"]
```



```
# example with a simple type (e.g. float)
temperature = 36.7
newTemperature = temperature
print(temperature, newTemperature) # 36.7 36.7
temperature = 37.2
print(temperature, newTemperature) # 37.2 36.7


# example with a compound type (e.g. list)
names = ["Sarah", "Zheng", "Amol"]
otherNames = names
names[1] = "Lin" # Zheng is replaced by Lin
print(names, otherNames) # ["Sarah", "Lin", "Amol"]
                         # ["Sarah", "Lin", "Amol"]
otherNames[2] = "Ahmed"
print(names, otherNames) # ["Sarah","Lin","Ahmed"]
                         # ["Sarah","Lin","Ahmed"]
```

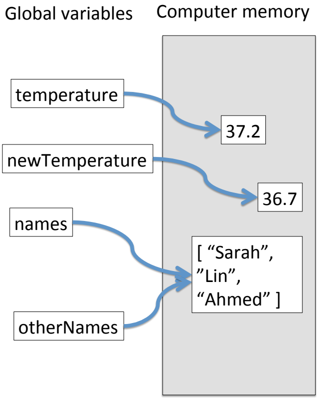


```
# example with a simple type (e.g. float)
temperature = 36.7
newTemperature = temperature
print(temperature, newTemperature) # 36.7 36.7
temperature = 37.2
print(temperature, newTemperature) # 37.2 36.7


# example with a compound type (e.g. list)
names = ["Sarah", "Zheng", "Amol"]
otherNames = names
names[1] = "Lin" # Zheng is replaced by Lin
print(names, otherNames) # ["Sarah", "Lin", "Amol"]
                         # ["Sarah", "Lin", "Amol"]
otherNames[2] = "Ahmed"
print(names, otherNames) # ["Sarah","Lin","Ahmed"]
                         # ["Sarah","Lin","Ahmed"]
```



Important: Here, both variables names and otherNames point to the *same* list. So modifying the names list also modifies the content of otherNames. names and otherNames are *aliases* for the same list.

```
# example with a simple type (e.g. float)
temperature = 36.7
newTemperature = temperature
print(temperature, newTemperature) # 36.7 36.7
temperature = 37.2
print(temperature, newTemperature) # 37.2 36.7


# example with a compound type (e.g. list)
names = ["Sarah", "Zheng", "Amol"]
otherNames = names
names[1] = "Lin" # Zheng is replaced by Lin
print(names, otherNames) # ["Sarah", "Lin", "Amol"]
                         # ["Sarah", "Lin", "Amol"]
otherNames[2] = "Ahmed"
print(names, otherNames) # ["Sarah","Lin","Ahmed"]
                         # ["Sarah","Lin","Ahmed"]
```



Global variables

Computer memory

temperature

37.2

newTemperature

36.7

names

[“Sarah”,
”Lin”,
“Ahmed”]

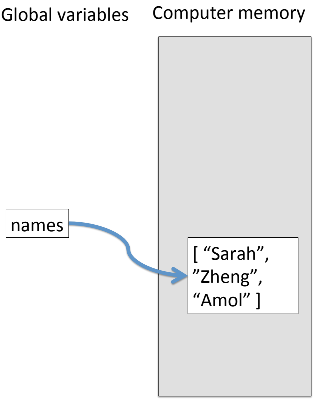
otherNames

... and vice-versa modifying the content of the list otherNames also modifies names

Cloning lists

What if we want names and otherNames to actually correspond to different lists, but we want otherNames to be initialized from names?

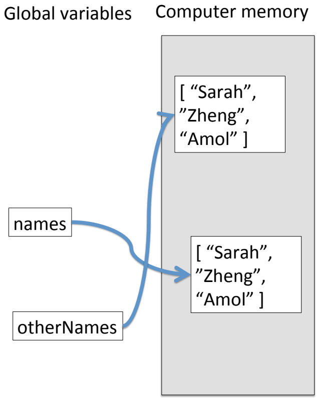
```
names = ["Sarah", "Zheng", "Amol"]
otherNames = names[:] # this clones the List names
names[1] = "Lin" # Zheng is replaced by Lin
print(names, otherNames) # ["Sarah", "Lin", "Amol"]
                         # ["Sarah", "Zheng", "Amol"]
```



Cloning lists

What if we want names and otherNames to actually correspond to different lists, but we want otherNames to be initialized from names?

```
names = [”Sarah”, ”Zheng”, ”Amol”]
otherNames = names[:] # this clones the List names
names[1] = ”Lin” # Zheng is replaced by Lin
print(names, otherNames) # [”Sarah”, ”Lin”, ”Amol”]
                         # [”Sarah”, ”Zheng”, ”Amol”]
```

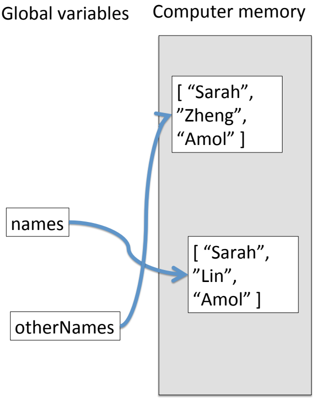


Note the use of [:]. This is what tells the interpreter to clone the names list. Now names and otherNames point to *different* lists, which just happen to contain identical content.

Cloning lists

What if we want names and otherNames to actually correspond to different lists, but we want otherNames to be initialized from names?

```
names = ["Sarah", "Zheng", "Amol"]
otherNames = names[:]  # this clones the List names
names[1] = "Lin"   # Zheng is replaced by Lin
print(names, otherNames) # ["Sarah", "Lin", "Amol"]
                         # ["Sarah", "Zheng", "Amol"]
```



Changing names does not change otherNames.

More on adding items to a list

.append(someObject) adds a single item to the end of the list

.extend(someList) adds items from another list to the end of the list

.insert(index, someObject) inserts an item at a given index

- Moves the remaining items to the right

```
names = ["Zheng", "Amol"]
otherNames = ["Chris","Irene"]

names.append("Bill")  # names is now
                      # ["Zheng", "Amol", "Bill"]

names.extend(otherNames) # names is now
         # ["Zheng", "Amol", "Bill", "Chris","Irene"]

names.insert(2,"Laura") # names is now
       # ["Zheng", "Amol", "Laura", "Bill", "Chris","Irene"]
```

Deleting items from a list

del someSlice statement can be used to remove an item or slice of items

```
names = ["Sarah", "Zheng", "Amol", "Vladimir"]

del names[1]  # removes Zheng from the list
              # names is now ["Sarah", "Amol", "Vladimir"]

del names[0:2] # removed Sarah and Amol
               # names is now ["Vladimir"]
```

.pop(index) will remove an individual and return it

```
names = ["Sarah", "Zheng", "Amol", "Vladimir"]

removedName = names.pop(2)
# names is now ["Sarah", "Zheng", "Vladimir"]
# removedName is now "Amol"
```

del statement and **.pop()** behave quite similarly, except **.pop()** returns the removed item

Deleting items from a list

.remove(someObject) removes the first instance of a matching item in a list

```
names = ["Sarah", "Zheng", "Amol", "Vladimir", "Zheng"]

names.remove("Zheng")
# names is now ["Sarah", "Vladimir", "Amol", "Zheng"]

names.remove("Zheng")
# names is now ["Sarah", "Vladimir", "Amol"]

names.remove("Billy") # causes exception: not in list
```

If no matching item is found in the list, Python raises a ValueError exception

Searching lists

.index(someObject) returns the index of the first matching item in a list

```
names = ["Sarah", "Zheng", "Amol", "Vladimir", "Zheng"]

indexVlad = names.index("Vladimir") # indexVlad is 3

indexZheng = names.index("Zheng") # indexZheng is 1

indexBob = names.index("Bob") # ValueError Exception
                              # Bob is not in list
```

.index(someObject) performs a linear search, and stops at the first match

- If no matching item is found, Python raises a ValueError exception

.count(someObject) returns the number of occurrences of the object in the list

```
names = ["Sarah", "Zheng", "Amol", "Vladimir", "Zheng"]

nbZheng = names.count("Zheng")  # 2
nbAmol = names.count("Amol")  # 1
nbBob = names.count("Bob")  # 0
```

Reversing the order of a list

.reverse() allows you to quickly reverse the order of a list

```
names = ["Sarah", "Zheng", "Amol", "Vladimir", "Zheng"]

names.reverse()
# names is now ["Zheng", Vladimir", "Amol", "Zheng","Sarah"]
```

Reversing is fast

- Temporarily reversing a list can often speed things up
- Remove and insert many items at the end of the list

Sorting lists

.sort() sorts a list in place

```
    L = [2,1,3,4,5,1,6]
    L.sort()
    print(L) # prints '[1,1,2,3,4,5,6]'
```

If you require a *clone* of the sorted list, use the **sorted()** function

```
    L = [2,1,3,4,5,1,6]
    sorted_L = sorted(L)
    print(L) # prints '[2,1,3,4,5,1,6]'
    print(sorted_L) # prints '[1,1,2,3,4,5,6]'
```

Other useful functions/methods

min() returns the smallest item in a list

```python
    L = [0,1,2,3,4,5,6,7,8,9]
    print(min(L)) # prints '0'
```

max() returns the largest items in a list

```python
    print(max(L)) # prints '9'
```