



MTU

Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

COMP7004 - Systems Scripting

Lecture 13: List

Dr. Vincent Emeakaroha

vincent.emeakaroha@mtu.ie

Semester 2, 2025

www.mtu.ie

List Data Type

- A list is a value that contains multiple values in an ordered sequence.
 - The term “list” refers to itself and not to the values inside of it
 - It can be stored in a variable or passed to a function like any other values
 - It is comparable to array in bash scripting but they are not the same.
- Since list can contain multiple values, it makes it easier to write programs that handle a large amount of data.
- List can contain other lists so it can be used to arrange data into hierarchical structures.

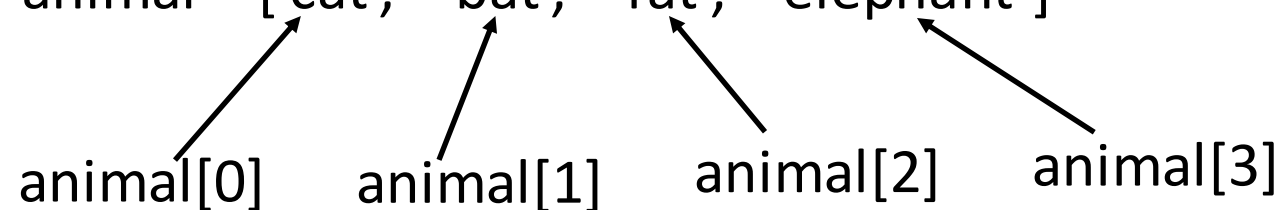
List Data Type

- A list value looks like this: ['cat', 'bat', 'rat', 'elephant']
- Similar to the denotation of string values that starts with a quote character and ends with a quote character, a list starts with an open square bracket and ends with a closing square bracket.
- Values inside a list are called items.
 - Items are separated with commas. We refer to them as comma-delimited.
- A list value is stored in a variable for easy reference.

Accessing List Values with Indexes

- Values in a list are indexed and can be individually accessed based on their index value and the variable name.
- Example:

```
animal = ['cat', 'bat', 'rat', 'elephant']
```



```
animal[0]  animal[1]  animal[2]  animal[3]
```

- The integer inside the square brackets that follows the list variable name is called an index.

Accessing List Values with Indexes

- Interactive shell example

```
>>> animal = ['cat', 'bat', 'rat', 'elephant']
>>> print(animal[0])
cat
>>> print(animal[1])
bat
>>> print('Hello ' + animal[0])
Hello cat
>>> print('The ' + animal[0] + ' ate the ' + animal[2] + '.')
The cat ate the rat.
```

- Python will give you an IndexError message if you use an index that exceeds the number of your list values.
 - Indexes can only be integer values and not float.

Nested List: Accessing Values

- List can contain other list values. The values in these lists of lists can be accessed using multiple indexes.

- Example:

```
>>> animal = [['cat', 'bat'], [10, 20, 30, 40, 50]]
>>> print(animal[0])
['cat', 'bat']
>>> print(animal[0][1])
bat
>>> print(animal[1])
[10, 20, 30, 40, 50]
>>> print(animal[1][3])
40
```

- The first index dictates which list value to use and the second identifies the value within the list value.

Negative Index

- While indexes start from 0 and go up, you can use negative integer for the index.
- For example

```
>>> animal = ['cat', 'bat', 'rat', 'elephant']
>>> print(animal[-1])
elephant
>>> print(animal[-3])
bat
>>> print('The ' + animal[-1] + ' is afraid of the ' + animal[-3] + '.')
The elephant is afraid of the bat.
```

- The integer value -1 refers to the last index in a list, the value -2 refers to the second to the last index, and so on.

Indexes and Slices

- Just like an index can get a single value from a list, a slice can get several values from a list, in the form of a new list.
- A slice is typed between square brackets like an index, but it has two integers separated by a colon.
- Examples:
 - `animal[3]` is a list with an index (one integer)
 - `animal[2:4]` is a list with a slice (two integers)

Slices

- Interactive shell example:

```
>>> animal = ['cat', 'bat', 'rat', 'elephant']
>>> print(animal[0:4])
['cat', 'bat', 'rat', 'elephant']
>>> print(animal[1:3])
['bat', 'rat']
>>> print(animal[0:-1])
['cat', 'bat', 'rat']
```

- A shortcut can leave out one or both of the indexes on either sides of the colon in the slice.

```
>>> animal = ['cat', 'bat', 'rat', 'elephant']
>>> print(animal[:2])
['cat', 'bat']
>>> print(animal[1:])
['bat', 'rat', 'elephant']
>>> print(animal[:])
['cat', 'bat', 'rat', 'elephant']
```

List Length

- The `len()` function will return the number of values that are in a list value that is passed to it, just like it can count the number of characters in a string value.
- Example:

```
>>> animal = ['cat', 'bat', 'rat', 'elephant']  
>>> print(len(animal))  
4  
...
```

Changing List Value with Indexes

- Normally a variable name goes on the left side of an assignment statement, like
 - `eggs = 24`
- However, you can also use an index of a list to change the value at that index. For example, `animal[1] = 'Zebra'` means “Assign the value at index 1 in the list `animal` to the string ‘Zebra’.”

```
>>> animal = ['cat', 'bat', 'rat', 'elephant']
>>> animal [1] = 'Zebra'
>>> print(animal)
['cat', 'Zebra', 'rat', 'elephant']
>>> animal [2] = animal [1]
>>> print(animal)
['cat', 'Zebra', 'Zebra', 'elephant']
```

List Concatenation and List Replication



MTU

Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

- Concatenation

- The + operator can combine two lists to create a new list value in the same way it combines two strings into a new string value.

- Replication

- The * operator can be used with a list and an integer value to replicate the list. Note that this is not a normal integer arithmetic.

```
>>> [1, 2, 3, 4] + ['A', 'B', 'C', 'D']  
[1, 2, 3, 4, 'A', 'B', 'C', 'D']  
>>> animal = [1, 2, 3, 4]  
>>> animal = animal + ['A', 'B', 'C', 'D']  
>>> print(animal)  
[1, 2, 3, 4, 'A', 'B', 'C', 'D']  
>>> [1, 2, 3] * 3  
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Removing List Values: del Statement

- The del statement will delete value at an index in a list. Remaining values in the list after delete action will be moved up one index.

```
>>> animal = ['cat', 'bat', 'rat', 'elephant']
>>> del animal [1]
>>> print(animal)
['cat', 'rat', 'elephant']
>>> del animal [2]
>>> print(animal)
['cat', 'rat']
```

- The del statement can also be used on a simple variable to delete it, as if it were an “unassignment” statement. If one tries to use the variable after deleting it, the program will throw a NameError error because the variable no longer exist.

Working with List

- When you first begin writing programs, its tempting to create many individual variables to store a group of similar values. For example, if I want to store the names of cats, I might be tempted to write a code like:

```
catName1 = 'Zophie'  
catName2 = 'Pooka'  
catName3 = 'Simon'  
catName4 = 'Sara'  
catName5 = 'Jasmine'  
catName6 = 'Rally'
```

- This is a bad way to write such a code. For one thing, if the number of cats changes, your program will never be able to store more cats than you have variables.
- This type of code has a lot of duplicates codes. Consider this example:

Working with List: allMyCatList.py



MTU

Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

```
print("Enter the name of cat 1:")
catName1 = input()
print("Enter the name of cat 2:")
catName2 = input()
print("Enter the name of cat 3:")
catName3 = input()
print("Enter the name of cat 4:")
catName4 = input()
print("Enter the name of cat 5:")
catName5 = input()
print("Enter the name of cat 6:")
catName6 = input()

print("The cat names are: ")
print(catName1 + " " + catName2 + " " + catName3 + " " + catName4 + " " + catName5 + " " + catName6)
```

- Instead of using multiple repetitive variables, you can use a single variable that contains a list value. An improved version of this code is provided in the next example.

Working with List: allMyCatList2.py



eolaíochta na Mumhan
nological University

```
catNames = []
while True:
    print("Enter the name of cat " + str(len(catNames) + 1) + " (Or enter nothing to stop.): ")
    name = input()
    if name == "":
        break
    catNames = catNames + [name] # list concatenation

print("Cat names are:")
for cName in catNames:
    print(" " + cName)
```

- The benefit of using a list is that now your data is in a structure and your program is much more flexible in processing the data.

Output:

```
Enter the name of cat 1 (Or enter nothing to stop.):
Zophie
Enter the name of cat 2 (Or enter nothing to stop.):
Pooka
Enter the name of cat 3 (Or enter nothing to stop.):
Simon
Enter the name of cat 4 (Or enter nothing to stop.):
Sara
Enter the name of cat 5 (Or enter nothing to stop.):
Jasmine
Enter the name of cat 6 (Or enter nothing to stop.):
Rally
Enter the name of cat 7 (Or enter nothing to stop.):
```

```
Cat names are:
Zophie
Pooka
Simon
Sara
Jasmine
Rally
```

List: Looping with Index Using for Loop

- Technically a **for** loop repeats a code block once for each value in a list or list-like value. For example:

```
for i in range(4):  
    print(i)
```

```
for i in [0, 1, 2, 3]:  
    print(i)
```

- Both of the code has the same output because Python considers the return value from **range(4)** to be similar to **[0, 1, 2, 3]**.

List: Looping with Index Using for Loop

- A common Python technique is to use `range(len(someList))` with a for loop to iterate over the indexes of a list. For example:

```
>>> supplies = ['pens', 'staplers', 'flame-throwers', 'binders']  
>>> for i in range(len(supplies)):  
    print('Index ' + str(i) + ' in supplies is: ' + supplies[i])
```

```
Index 0 in supplies is: pens  
Index 1 in supplies is: staplers  
Index 2 in supplies is: flame-throwers  
Index 3 in supplies is: binders
```

- Using `range(len(supplies))` will iterate through all the indexes of supplies, no matter how many items it contains.

The in and not in Operators

- The **in** and **not in** operators enable determining whether a value is in a list or not.
- Like other operators, **in** and **not in** are used in expressions and connect two values:
 - A value to look for in a list
 - The list where it may be found.
- The expression will evaluate to a Boolean value.

```
>>> 'pencil' in ['pens', 'staplers', 'flame-throwers', 'binders']  
False  
>>> 'pens' in ['pens', 'staplers', 'flame-throwers', 'binders']  
True  
>>> 'pencil' not in ['pens', 'staplers', 'flame-throwers', 'binders']  
True  
>>> 'pens' not in ['pens', 'staplers', 'flame-throwers', 'binders']  
False
```

Example: myPets.py

```
myPets = ['Zophie', 'Pooka', 'Fat-tail']  
print("Enter a pet name:")  
name = input()  
if name not in myPets:  
    print("I do not have a pet named " + name)  
else:  
    print(name + " is my pet.")
```

Output

```
Enter a pet name:  
Bekky  
I do not have a pet named Bekky
```

Multiple Assignment Trick

- The multiple assignment trick is a shortcut that lets you assign multiple variables with the values in a list in one line of code. For example:

```
>>> cat = ['fat', 'black', 'loud']  
>>> size = cat[0]  
>>> color = cat[1]  
>>> disposition = cat[2]
```

Single assignment

```
>>> cat = ['fat', 'black', 'loud']  
>>> size, color, disposition = cat
```

Multiple assignment tricks

- The number of variables and the length of list must be exactly equal, or Python will throw a `ValueError` error.

Augmented Assignment Operators



Augmented assignment statement	Equivalent assignment statement
var += 1	var = var + 1
var -= 1	var = var - 1
var *= 1	var = var * 1
var /= 1	var = var / 1
var %= 1	var = var % 1

- The += operator can also do string and list concatenation, and the *= can do string and list replication. Here are example:

```
>>> var = 'Hello'
>>> var += ' world!'
>>> print(var)
Hello world!
```

```
>>> bacon = ['Zophie']
>>> bacon *= 3
>>> print(bacon)
['Zophie', 'Zophie', 'Zophie']
```


Python Methods

- A **method** is very similar to a **function**, except it is “called on” a value.
- For example, if a list value were stored in **bacon**, you would call the **index()** list method on that list like so:
 - **bacon.index('hello')**
 - The **method part comes after the value separated by a period.**
- Each data type has its own set of methods. The list data type, for example, has several useful methods for finding, adding, removing, and otherwise manipulating values in a list.

List index() Method

- List values have an **index()** method that can be passed a value, and if that value exists in the list, the index of the value is returned. If the value is not in the list, then Python throws a ValueError error.
- Example:

```
>>> cat = ['fat', 'white', 'loud', 'heyas']  
>>> print(cat.index('loud'))  
2  
>>> print(cat.index('fat'))  
0
```
- Where there are duplicate of the value in the list, the index of its first appearance is returned.
- If the value is not in the list, a ValueError will be thrown.

Adding Values to List: `append()` and `insert()`

- To add new values to a list, use the `append()` and `insert()` methods.

- For example:

```
>>> cat = ['fat', 'white', 'loud', 'heyas']
>>> cat.append('hairy')
>>> print(cat)
['fat', 'white', 'loud', 'heyas', 'hairy']
```

- The `insert()` method can insert a value at any index in the list. The first argument to the `insert()` method is the index for the new value, and the second argument is the new value to be inserted.

- For example:

```
>>> cat = ['fat', 'white', 'loud', 'heyas']
>>> cat.insert(1, 'mice')
>>> print(cat)
['fat', 'mice', 'white', 'loud', 'heyas']
```

Adding Values to List: `append()` and `insert()`

- Notice that the code is `cat.append('hairy')` and `cat.insert(1, 'mice')`, **not** `cat = cat.append('hairy')` and `cat = cat.insert(1, 'mice')`.
- Neither `append()` nor `insert()` method gives the value of `cat` as its return value. In fact their return value is `None`. Rather the list is modified *in place*.
- Methods belong to a single data type. The `append()` and `insert()` methods are list methods and can be only called on list values, not on other values such as integers or strings.

Removing Values from List: remove()



- The **remove()** method removes a value passed as argument to it from the list it is called on.
- For example:

```
>>> cat = ['fat', 'white', 'loud', 'heyas']  
>>> cat.remove('heyas')  
>>> print(cat)  
['fat', 'white', 'loud']
```

- Attempting to remove a value that does not exist in the list will result to ValueError error.
- If the value appear multiple times in the list, only the first instance of the value will be removed.

References

- We use variables to store strings and integer values.
For example:

```
>>> animal = 45
>>> cat = animal
>>> animal = 500
>>> print(animal)
500
>>> print(cat)
45
```

- This is because **animal** and **cat** are different variables and store different values.

References

- But **list** does not work this way. When you assign a list to a variable, you are actually assigning a list **reference** to the variable.
- A reference is a value that points to some bit of data, and a list reference is a value that points to a list.
- Example:

```
>>> animal = [1, 2, 3, 4, 5]
>>> cat = animal
>>> cat[1] = 'Hello'
>>> print(animal)
[1, 'Hello', 3, 4, 5]
>>> print(cat)
[1, 'Hello', 3, 4, 5]
```
- When **animal** is copied to **cat**, only a new reference was created and stored in **cat**, not a new list.

References

- Python uses references whenever variables must store values of mutable data types, such as list or dictionaries. For values of immutable data types, such as strings, integers, or tuples, Python variables will store the value itself.

Passing References

- References are particularly important to understand how arguments get passed to functions. When a function is called, the values of the arguments are copied to the parameter variables.
- For list (and dictionary, to be discussed later), this means a copy of the reference is used for the parameter.

- For example:

- [PassingReference.py](#)

```
# passing reference
```

```
def pets(someParameter):  
    someParameter.append('Dog')
```

```
animal = [1, 2, 3]  
pets(animal)  
print(animal)
```

Output

```
[1, 2, 3, 'Dog']
```

- Notice that when `pets()` was called, a return value was not used to assign a new value to `animal`. Instead it modifies the list in place directly.

copy() and deepcopy() Functions

- In some cases you want to keep original list values and work on a copy of it instead. Python provides a module named **copy** that provides both **copy()** and **deepcopy()** functions.
- The **copy.copy()** function can be used to make a duplicate copy of a mutable value like list and dictionary, not just a copy of a reference.
- For example:

```
>>> import copy
>>> animal = [1, 2, 3, 4, 5]
>>> cat = copy.copy(animal)
>>> cat[2] = 'Sunny'
>>> print(animal)
[1, 2, 3, 4, 5]
>>> print(cat)
[1, 2, 'Sunny', 4, 5]
```
- If the list you need to copy contains lists, then use the **copy.deepcopy()** function instead of **copy.copy()**. The **deepcopy()** function will copy these inner lists as well.

List-like Types: Strings and Tuples

- List is not the only data type that represent ordered sequence of values. Strings are actually similar, if you consider a string to be a “list” of single characters.
- Many of the things you can do with lists can also be done with strings:
 - Indexing, slicing, using in for loops, with len(), and with in and not in operators

- Example:

```
>>> name = 'Zophie'
>>> name[0]
'Z'
>>> name[0:4]
'Zoph'
>>> 'Zo' in name
True
>>> 'p' not in name
False
```

Mutable and Immutable Data Types

- List and strings are different in an important way. A list value is a mutable data type:
 - It can have values added, removed, or changed.
- However, a string is immutable:
 - It cannot be changed.
 - Trying to reassign a single character in a string results in a `TypeError` error.
 - For example:

```
>>> name = 'Zophie a cat'  
>>> name[7] = 'the'
```

```
Traceback (most recent call last):  
  File "<pyshell#151>", line 1, in <module>  
    name[7] = 'the'  
TypeError: 'str' object does not support item assignment
```

Mutable and Immutable Data Types

- The proper way to “mutate” a string is to use slicing and concatenation to build a new string by copying from parts of the old string.
- Example:

```
>>> name = 'Zophie a cat'
>>> newName = name[0:7] + 'the' + name[8:12]
>>> print(name)
Zophie a cat
>>> print(newName)
Zophie the cat
```
- We used [0:7] and [8:12] to refer to the characters that we do not wish to replace.
- Notice that the original string “Zophie a cat” is not modified because strings are immutable.

The Tuple Data Type

- Tuple data type is almost identical to the list data type, except in two ways.
- First, tuples are typed with parentheses, (and), instead of square brackets, [and].
- Example:

```
>>> eggs = ('hello', 42, 0.5)
>>> print(eggs[0])
hello
>>> print(eggs[1:3])
(42, 0.5)
>>> print(len(eggs))
3
```


The Tuple Data Type

- Second, the main way that tuples are different from lists is that tuples like strings, are **immutable**.
- Tuples cannot have their values modified, appended, or removed.
- Consider the output of the following code:

```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[1] = 99
```

```
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    eggs[1] = 99
TypeError: 'tuple' object does not support item assignment
```

The Tuple Data Type

- If you have only one value in your tuple, you can indicate this by placing a trailing comma after the value inside the parentheses. Otherwise Python will think that you have just typed a string value inside a parentheses.
- The `type()` function enable identifying a value data type.
Example:

```
>>> type(('hello',))  
<class 'tuple'>  
>>> type(('hello'))  
<class 'str'>
```

- You can use tuples to convey to anyone reading your code that a sequence of value is not intended to be changed.

Converting Types: list() and tuple() Functions

- Just like how `str(42)` will return '42', the string representation of the integer 42, the function `list()` and `tuple()` will return list and tuple versions of the value passed to them.
- For example:

```
>>> print(tuple(['cat', 'dog', 5]))
('cat', 'dog', 5)
>>> print(list(('cat', 'dog', 5)))
['cat', 'dog', 5]
>>> print(list('hello'))
['h', 'e', 'l', 'l', 'o']
```
- Converting a tuple to a list is handy if you need a mutable version of a tuple value.

