Lecture 7 (Week 8): Advanced Lists, Tuples and Mutability. Errors and Debugging.

Dr Simon D'Alfonso

School of Computing and Information Systems
Faculty of Engineering and Information Technology

Lecture 6 Challenges

- Convert the code on the Nested for Example 1 slide to a functionally equivalent piece of code that uses *while* loops instead of *for* loops.
- Modify the Exercise 1 Solution so that the function instead returns a list containing all characters that don't appear in the string. If the string is a pangram, then return an empty list.
- Write a program that uses a while loop to print out each individual character in a given string line by line.

Continue

- We have seen the *break* statement for loops in previous weeks.
- One other statement to briefly mention is *continue*, which stops the current iteration of a loop and continues with the next.

```
#print out only odd numbers
numbers = [0,1,2,3,4,5,6,7,8,9]
for x in numbers:
    if x % 2 == 0:
        continue
    print(x)
```

Today

1. Sequences and collections
   1. Lists Comprehensions
   2. Tuples
   3. Dictionaries (brief introduction)
2. Mutability
3. Copying lists
4. Types of errors: syntax, runtime and logic
5. Debugging your code and error handling

Advanced Lists

- By now we are quite familiar with lists:
  - They are *collections* – collections of things
  - One list can hold a variety of element types
  - They are sequences (ordered collections) - the order in which we input the items will be the same when we access them.

Lists

Examples:

```
["head", "tail", "tail"] # list of strings
[5, 5, 30, 10, 50]   # list of ints
[1, 2, "greetings", 3.0, 4.0, False] # all sorts
[[1,2],[3,4]] #lists within lists
```

As with all types, we can assign a list to a variable:

```
>>> fruits = ["orange", "apple", "apple"]
>>> fruits[0][1] = ?
```

List methods:
https://www.w3schools.com/python/python_lists_methods.asp

Exercise 1

Write a function middle() that returns the middle element(s) from a given input list:

```
>>> lst = [1, 2, 3]
>>> middle(lst)
2


>>> lst = [1, 2, 3, 4]
>>> middle(lst)
[2, 3]
```

List comprehensions

- Suppose we have a list of integers lst1 and want to create a corresponding list of integers lst2 such that each item of lst1 is doubled in lst2. For example, if lst1 = [1, 2, 3, 4] then lst2 = [2, 4, 6, 8].
- Based on what we have learnt so far, we could use a *for* loop to achieve this:

```
lst1 = [1, 2, 3, 4]
lst2 = []
for item in lst1:
    lst2.append(item * 2)
```

List comprehensions

- A *for* loop is a perfectly fine way to do it. However, Python provides another, elegant way to achieve this, called list comprehensions.
- List comprehensions are of the form:

  [expression for item in list]

- An example which is equivalent to our *for* loop code in the previous slide:

```
lst1 = [1, 2, 3, 4]
lst1_duplication = [item for item in lst1]
lst2 = [(item * 2) for item in lst1]
```

List Comprehensions

- It is also possible to add conditionals to list comprehensions.
- Get numbers between 0 and 20 into a list:

```
number_lst = [x for x in range(21)]
```

- Get *even* numbers between 0 and 20 into a list:

```
number_lst_even = [x for x in range(21) if x % 2 == 0]
```

- Nested loops, nested conditionals and other sophisticated comprehension constructs are possible, though beyond the scope of this lecture.

Exercise 2

Write a function palindromes() that receives a list of strings as input and prints a list of all those strings which are palindromes using list comprehension (a palindrome is a word that is the same in reverse order, e.g., "dad").

```
>>> words = ["dad", "hello", "goodbye", "mom",
"rotator"]
>>> palindromes(words)
["dad", "mom", "rotator"]
```

Tuples

- Another way to store multiple things is in a tuple
- A tuple is also a sequence (ordered collection)
- In Python tuples are written with round brackets:
  - (1, 2, 3)
  - (37.8136, 144.9631)
  - ('red', 'Fred', 67)

Accessing values in a tuple

You can access tuple items by referring to the index number, inside square brackets (same as lists)

```
>>> my_tuple = ('red', 'Fred', (67, 8), -6.88)
>>> my_tuple[2]
(67, 8)
>>> my_tuple[:2]
('red', 'Fred')
>>> my_tuple[[1][:2]]
'Fr'
>>> 'red' in my_tuple
True
```

Lists versus Tuples

- Individual list items can be changed (mutable), whereas individual tuple items cannot (immutable).

```
>>> my_list = [1, 2, 3]
>>> my_tuple = (1, 2, 3)
>>> my_list[1] = 6
>>> my_list
[1 , 6, 3]

>>> my_tuple[1] = 6
TypeError : 'tuple' object does not support item assignment
```

- You cannot change, delete or add items to a tuple once it is defined.

Tuple methods

Given that they are immutable, tuples don't have many of the methods that lists have:

- tuple.index(x) - return index in the tuple of the first item whose value is equal to x.
- tuple.count(x) - return the number of times x appears in the tuple.
- tuple.append(x) - ?

Object Identity

- When we get a literal or construct an object and assign it to a variable, the variable is simply assigned the identity of the new object.
- Therefore, when we assign a variable to a new variable, the new variable is simply given the identity of the existing object.

```
>>> int1 = 90059
>>> int2 = int1
>>> int2 is int1 #same object?
True
>>> int2 = 90059
>>> int2 is int1 #same object?
False
```

Mutability

- When you pass a mutable object (list here) to a function and locally alter an element within the function, the change is preserved in the global object
- Take the following functions and the next slide:

```
def change_list(lst):
    lst = []
    return lst

def change_list_item(lst):
    lst[0] = "changed"
```

Mutability

```
>>> my_list = [1, 2, 3]
>>> change_list(my_list)
[] #empty list returned according to function
>>> my_list
[1, 2, 3] #but my_list itself does not change


>>> change_list_item(my_list)
>>> my_list
['changed', 2, 3] #the list item itself
changes
```

Mutability

When one list variable is assigned to another variable, changing one changes the other, since they are the same thing:

```
>>> list1 = [1, 2, 3]
>>> list2 = list1
>>> list1 is list2
True
>>> list2[0] = "changed"
>>> list2
['changed', 2, 3]
>>> list1
['changed', 2, 3]
```

Copying things

Making copies of things is something that you might have to do. However there are several ways to do it in Python, and things can get tricky.

'Copying' Lists with =

```
#copying using =
old_list = [1, 2, 3, 4, 'a']
new_list = old_list

new_list[4] = 5
new_list.append(6)

print('old_list:', old_list)
print('ID of old_list:', id(old_list))
print()
print('new_list:', new_list)
print('ID of new_list:', id(new_list))
```

Copying Lists with copy()

```
#copying using copy()
import copy

old_list = [1, 2, 3, 4]
new_list = copy.copy(old_list)
old_list[3] = 'four'
new_list.append(5)

print("old_list:", old_list)
print(id(old_list))
print()
print("new_list:", new_list)
print(id(new_list))
```

Copying Lists with copy()

```
#copying using copy()
import copy

old_list = [123, ['four', 5, 6], [7, 8, 9]]
new_list = copy.copy(old_list)
old_list.append([10])
new_list.append([11])
old_list[1][0] = 4
old_list[0] = 321

print("old_list:", old_list)
print(id(old_list))
print()
print("new_list:", new_list)
print(id(new_list))
```

Copying Lists with copy()

- With the code in the previous slide, why do old_list and new_list both end up with the element [4, 5, 6], even though we first did a copy then just changed the ‘four’ to 4 in old_list?
- old_list and new_list give the same result here, because the change of old_list[1][0] = 4 is being made to a list ([‘four’, 5, 6]), which is a *mutable* type, a mutable part of old_list / new_list.
- Hence if two lists each contain an element that points to the same mutable instance, changing that element via one of the list variables changes the value for both.
- On the other hand, with an immutable item such as the 123 at position [0], changing element [0] for old_list does not affect element[0] for new_list.

Copying Lists with deepcopy()

To ensure that all items, even mutable ones, are completely duplicated, we can use deepcopy()

```
#copying using deepcopy()
import copy
old_list = [[1, 2, 3], ['four', 5, 6], [7,
8, 9]]
new_list = copy.deepcopy(old_list)
old_list.append([10])
new_list.append([11])
old_list[1][0] = 4

print("old_list:", old_list)
print("new_list:", new_list)
```

Copy Conclusion

- Shallow copy of some structure, that is structure.copy(), makes new copies of all the immutable elements into a new structure but does not copy the mutable elements and will instead reference them.
- Deep copy on the other hand completely makes a new copy of all immutable and mutable elements.

Exercise 3

Write a function **word_count** that takes as input a string of **text** and a **word** and returns the number of times that word appears in the text.

```
>>> word_count("Word, WORD! and word.", "word")
3

>>> word_count("one thousand two thousand three thousand four thousand one", "thousand")
4
```

Dictionaries

- Another data structure used for storing collections of items is the *dictionary*.
- Dictionaries are collections but not sequences.
- Dictionaries store items as **key: value** pairs just like a book dictionary stores a collection of **word: meaning** pairs:

```
>>> australian_capitals = {'VIC': 'Melbourne', 'NSW': 'Sydney'}
>>> cars = {'1MN 3JK': 'Mazda 6 sedan', '5SD 2WE': 'Ford Ranger’}


>>> australian_capitals['VIC']
'Melbourne'
>>> cars['1MN 3JK’]
'Mazda 6 sedan'
```

Dictionary count keeping

Following on from Exercise 3, how could we write a function word_count2() that returns a count for each word in the text.

For example:

```
txt = "one thousand two thousand three thousand
four thousand one"
word_count2(txt)
```

Count of ‘one’ is 2

Count of ‘two’ is 1

Count of ‘three’ is 1

Count of ‘four’ is 1

Count of ‘thousand’ is 4

We will look at dictionaries in greater detail next week to achieve this type of thing.

Bugs

- A (software) “bug” is an error/flaw in a piece of code that leads to a malfunction
- According to Steve McConnell’s book "Code Complete, the industry average is about 15-50 errors per 1000 lines of delivered code.
- Error Types:
  - syntax errors = incompatibility with the syntax of the programming language
  - run-time errors = errors at run-time, causing the code to crash
  - logic errors = design errors, such that the code runs but doesn’t do what it is supposed to do

Tips for debugging

- Using modular programming techniques (functions for each sub-task)
- Coding and testing one task at a time
- Diagnostic print statements
- Using a tool to trace program execution
  - https://docs.python.org/3/library/trace.html
  - https://towardsdatascience.com/3-tools-to-track-and-visualize-the-execution-of-your-python-code-666a153e435e

We will now also look at:

- Assertions
- Catching or handling error exceptions

Errors when coding

Syntax errors can be detected before your program begins to run. These types of errors are usually typing mistakes, but more generally it means that there is some problem with the structure of your program.

Runtime errors occur as your program executes. Since Python is an interpreted language, these errors will not occur until the flow of control in your program reaches the line with the problem.

Whenever a **run-time** error occurs in Python, it takes the form of an exception being raised.

Syntax errors

Syntax errors occur when the Python interpreter attempts to convert the program text into **machine code**.

You can think of these errors like “spelling and grammar” errors. As you become more practiced at typing Python code you will find syntax errors easier to detect and fix.

```
>>> a = 2 +
SyntaxError: invalid syntax
```

Runtime errors and exceptions

Python has many in-built exceptions, and the names are usually self-explanatory. Some of the more common ones:

ValueError – the value of an object is invalid for that type

```
>>> a = int("a")

Traceback (most recent call last):
File "<stdin>", line 1, in <module>

ValueError: invalid literal for int() with base 10: 'a'
```

NameError – an undefined variable has been used

```
>>> b = a

Traceback (most recent call last):
File "<stdin>", line 1, in <module>

NameError: name 'a' is not defined
```

Common exceptions cont.

IndexError – an out-of-range list or tuple index has been used

```
>>> a = [1,2,3]
>>> a[3]

Traceback (most recent call last):
File "<stdin>", line 1, in <module>

IndexError: list index out of range
```

KeyError – a non-existent dictionary key has been used

```
>>> a = {1:2}
>>> a[2]

Traceback (most recent call last):
File "<stdin>", line 1, in <module>

KeyError: 2
```

UnboundLocalError - referencing a local variable inside a function become variable assignment statement

```
>>> def funct(lst):
        for x in range(len(lst)):
            val = lst[i]

UnboundLocalError: local variable
'i' referenced before assignment
```

TypeError – an operation has been attempted which is invalid for the type of the target object or object type combination

```
a = 1 + "2"

Traceback (most recent call last):
File "<stdin>", line 1, in <module>

TypeError: unsupported operand
type(s) for +: 'int' and 'str'
```

Logic Errors

Logic errors are not errors to the computer at all. They occur when the program runs without crashing but produces an incorrect result. Thus, the only way you can identify logic errors is by the output produced:

```
if x < 5 and x > 7: #there is no such x

if x % 2 == 0: #condition for an even number
    print("x is odd")
else:
    print("x is even")
```

To find and eliminate logic errors programmers should test code thoroughly with a range of test cases and trace code execution.

Assertions

To date, we have perhaps tended to assume well-behaved inputs to our functions and such, and lived with the fact that ill-behaved inputs will cause a logic or run-time error:

```
def withdraw(amount, balance):
    if balance < -100:
        print("Insufficient balance")
        return(balance)
    else:
        print("Withdrawn")
        return(balance - amount)

>>> print(withdraw(100, 0))
Withdrawn
-100

What about print(withdraw(100, '0'))?
```

Assertions

One way to ensure that the inputs are of the right type is with *assert*:

```
def withdraw(amount, balance):
    assert type(balance) == int, "balance should be an integer"
    if balance < -100:
        print("Insufficient balance")
        return(balance)
    else:
        print("Withdrawn")
        return(balance - amount)

>>> print(withdraw(100, '0'))
AssertionError: balance should be an integer
```

- Assertions are used when debugging code and are a quick way to check/ensure that something is as expected
- They can be used with Exception Handling too (next slides)
- Can alternatively use an explicit *if* statement if the result is important to the logic of the code

Exception Handling

It is possible to handle error exceptions within your code using Python Try Except:

```
try:
    code block
except ErrorType1: #optional
    code block
except ErrorType2: #optional
    code block
except:
    code block
```

- *try* attempts to execute its block of code, and passes off to the *except* handlers (which are also tested in linear order) only if an exception is raised during the execution
- There are also *else* and *finally* options: https://www.w3schools.com/python/python_try_except.asp

Exception Handling: Example 1

This code is not the best as it does not tolerate non-numerical inputs:

```
x = "not a number"
while type(x) != int:
    x = int(input("Please enter a number: "))
```

This version is better:

```
while True:
    try:
        x = int(input("Enter a number: "))
        break
    except ValueError:
        print("Try again")
```

Exception Handling: Example 2

```
try:
    x = 0
    y = 5
    #a = b
    print(y/x)
    lst = [1, 2, 3]
    print(lst[5])
except ZeroDivisionError:
    print("You cannot divide by zero")
except IndexError:
    print("Item must exist at index")
except:
    print("Some other error")
```

Summary

Today we covered:

- List comprehensions
- Tuples – immutable sequences/collections
- Mutability, the ability to change elements after the object is assigned a value. Lists are mutable sequences/collections.
- Copying lists
- Dictionaries very briefly – key: value collections
- A bit about bugs and debugging
- Three types of errors:
  - Syntax – you got the “grammar” wrong
  - Runtime – errors with the running of the program
  - Logic – the code runs but doesn’t (always) do what it should
- Assertion and Error Handling

Lecture 7 Challenges

- Write a function that devowels and returns an input string using list comprehension. Note that you will want to convert the input string to a list to do this, then reconstruct the devoweled string from the list comprehension result before returning it.
- Write a function that takes one tuple as input, returning True if all tuple items are of the same type, and False otherwise.

Lecture Identification and Acknowledgement

Coordinator / Lecturer: Simon D'Alfonso

Semester: Semester 1, 2022

These slides include materials from 2020 - 2021 instances of COMP90059 run by Kylie McColl or Wally Smith