IT@JCU

CP1401 Module 8
Lists and Tuples

JAMES COOK UNIVERSITY
AUSTRALIA

Learning outcomes – you will be able to:

- Choose appropriate data structures
- Use lists and tuples to store sequential data

- MANY programming problems require dealing with data containing multiple parts/elements, and so require lists

Do this now

- Write a program to ask the user for how many hours of study they did each day last week. Display the total, average, minimum and maximum hours as well as a nice text 'table' that shows how many hours they studied for.

Hmmm…

We know we can use a definite loop, but how do we store 7 different values? Surely not with 7 different variables…
What if it were 7000?

Day	Hours of study
1	2
2	1.5
3	1
4	3
5	0
6	2
7	1.5

Storing data in Python variables

- We know how to store individual pieces of data in Python

```
age = 23
study_time = 2.5
```

- Sometimes we want to group data together,
  like 7 different study_times, or 7000 samples of air quality readings, or…?

Don't repeat yourself
Don't repeat yourself

- Having variables for each value in a sequence, ***can't be right!***

day1=2, day2=1.5, day3=1, day4=3, day5=0, day6=2, day7=1.5

- That's a lot of variables to keep track of
  - What if we needed to save 365 or 700000 days of data?

- **Remember:** any time you find yourself repeating yourself you need to ask yourself if you're repeating yourself and then there's probably a better way to avoid repeating yourself!

Some variable types store **collections** of data

- List
- Tuple
- Dictionary
- Class
- …

- We will learn about lists and tuples (focusing on lists) now, and leave dictionaries and classes for Programming 2.

In the Repetition Structures lecture, we saw lists briefly...

(Because lists and looping go together so well)

Here are some simple for loop examples

```
for subject in ["CP1401", "CP1404", "CP2406"]:
    print("I like", subject)
```

```
I like CP1401
I like CP1404
I like CP2406
```

```
for number in [2, 4, -8, 99]:
    print(number, end=" ")
```

```
2 4 -8 99
```

```
for character in "Python":
    print(character, end="-")
```

```
P-y-t-h-o-n-
```

The first two use lists, which we will learn later in the subject…

Nested loops

- How would we print a line or something after each round (not each person?)

- Note the location: *after* the inner loop is finished, but *inside* the outer loop

```
for round_number in range(1, 4):
    for name in ["Miles", "Ella", "Chet"]:
        print("Round", round_number, "-", name)
    print("--------------------")
```

```
Round 1 - Miles
Round 1 - Ella
Round 1 - Chet
---------------
Round 2 - Miles
Round 2 - Ella
Round 2 - Chet
---------------
Round 3 - Miles
Round 3 - Ella
Round 3 - Chet
---------------
```

Lists are useful to store collections of data

- A list is an object that contains multiple data items
  - Each item in a list is called an **element**
  - Lists can hold items of different (any) types

```
friends = []  # empty list
subjects = ["CP1200", "CP1030", "MA1000"]
scores = [18, 25, 96]
things = [True, 1.2, "Good", [1, 10]]
```

[Square brackets]

Lists, tuples and strings are ordered sequences

- A sequence is an object that contains multiple items (elements) of data stored in sequence (*ordered*) one after another
- Python sequences include lists, tuples and strings
  - lists are mutable
    - you can modify individual elements
  - tuples and strings are immutable
    - to change a tuple or string you must create a new one

You can iterate over a list using a for loop

- Lists and loops go together a lot
- Since lists contain a definite number of items, you would use a definite loop to print all items

```
subjects = ["CP1401", "CP1404", "CP2406"]
for subject in subjects:
    print(subject)
```

Use **plural** names for sequence/composite types

- Lists are collections – they contain multiple elements, so…
- **Name them with plural terms**
- E.g., a list containing numbers could be… `numbers`
- A list of subjects is called… `subjects`
- A list of names is… `names`
- What about?
  - ages
  - countries
  - countries that are members of the Commonwealth

Use **plural** names for sequence/composite types

```
subjects = ["CP1401", "CP1404", "CP2406"]
for subject in subjects:
    print(subject)
```

- Notice how nice this is. Each element is a… "subject",
  So a list of "subjects" is called… "subjects"

```
for singular in plural
```

is a very common naming pattern.
If yours doesn't match, check it.

Each item in a list has an index

```
subjects = ["CP1401", "CP1404", "CP2406"]
```

Index	Element
0	"CP1401"
1	"CP1404"
2	"CP2406"

- Index of the first element in the list is 0,
  second element is 1, and
  nth element is n-1
- Negative indexes identify positions relative to the end of the list (not available in many other languages)
  - -1 identifies the last element, -2 identifies the second-last element, etc.

Index	Element
-3	"CP1401"
-2	"CP1404"
-1	"CP2406"

Use indexing to access elements in a sequence

```
subjects = ["CP1401", "CP1404", "CP2406"]
print(subjects[2])  # Prints "CP2406"
```

- Use the index in square brackets to access individual elements

What is the output of this code?

```
numbers = [10, 20, 30]
print(numbers[1], numbers[-1], numbers[-3])
```

Beware of invalid indexes

- An IndexError exception is raised if an invalid index is used

```
things = ["one"]  # only one element, index = 0
print(things[1])  # accessing non-existent element
```

```
IndexError: list index out of range
```

IT@JCU

List methods, functions, statements, operators

JAMES COOK UNIVERSITY AUSTRALIA

The len function returns the length of a collection

- The `len` function returns the length of any collection type
  - list, string, tuple, dictionary, set…
- The last element in a list is always at index: length - 1

```
number_of_things = len(things)
length_of_first_thing = len(things[0])
```

Note: You never need to store the length of a list in a separate variable. In general, never store *derivable* data.

Use Python's built-in **functions** with collections

- **min**, **max** and **sum** functions: built-in functions that do as you'd expect - the collection is passed as an argument

```
average = sum(scores) / len(scores)
```

- **del** statement: removes an element from a specific index in a list. E.g., to delete the 2nd element in scores, use:

```
del scores[1]
```

These are not list **methods**.
min and max are **functions**.
del is a **statement**.

Lists (and most other types) have **methods**

- We have seen methods before:

```
input("Enter choice: ").lower()
```

  - See how the dot separates the **object** (left) from the **method** (right)

Some useful list methods:

- **append**(item): used to add item to the end of the existing list
- **sort**(): used to sort the elements of the list in ascending order
- **reverse**(): reverses the order of the elements in the list

Use the **append** method to add elements to a list

```
scores = [18, 25, 33]
scores.append(55)  # scores now has 4 items
new_score = int(input("Score: "))
scores.append(new_score)

# scores[5] = 66  # This would NOT work
scores[4] += 1  # This does work (modify existing item)
print(scores)
```

Use "**in**" to determine if an item is in a list

- The **in** and **not in** operators can be used for searching a collection

```
if "CP1401" in subjects:
    print("You are cool")

if value not in values:
    print(value, "is not found")
```

Strings are sequences too

- Much of what works with lists also works with strings (but not modifying)

```
string = "Hello"
print(string[0])  # 'H'

for character in string:
    print(character.upper(), end="-")
print(len(string))
# Prints H-E-L-L-O-5
```

Do this now

- Complete this code that asks the user for scores until they enter a negative, adds them to the list, then prints the maximum score.

```
scores = __
score = int(input("Score: "))
while ___________

    ____________________________
    score = int(input("Score: "))
print("Your highest score is", ____________)
```

Do this now

- Complete this code that asks the user for scores until they enter a negative, adds them to the list, then prints the maximum score.

```
scores = []
score = int(input("Score: "))
while score >= 0:
    scores.append(score)
    score = int(input("Score: "))
print("Your highest score is", max(scores))
```

Pseudocode for lists

Very similar to actual code.
Don't go out of your way to make it harder to write.

```
prices = empty list
repeat 5 times
    get price
    add price to prices
prices[0] = 0.0
print prices[0]
```

Look for collections during design

- Look for situations where you need to store multiple related items - "plural" sounding names, as well as words like:
  - list, collection, group, set, etc.

Problem Description

Julie is planning a holiday and wants to record all the places she goes in the order she visits them. Once she returns, she wants to be able to display the list of places and which place had the longest name (because she likes that sort of thing).

Possible pseudocode for solution

```
places = empty list
get place
while place is not empty
    add place to places
    get place
for place in places
    print place
```

(We'll leave the longest name as a challenge)

Code from that pseudocode

```
places = []
place = input("Place: ")
while place != "":
    places.append(place)
    place = input("Place: ")
for place in places:
    print(place)
```

Lists and functions

- Mutable objects (lists) are handled differently than immutable ones (like numbers, strings) when passed to functions
- You do not need to return a list to modify it

```
def main():
    numbers = [1, 2, 3]
    add_offset(numbers, 2)
    print(numbers)  # prints [3, 4, 5]

def add_offset(elements, offset):
    for i in range(len(elements)):
        elements[i] += offset
```

IT@JCU

Tuples

Like lists, but different

JAMES COOK UNIVERSITY AUSTRALIA

Tuples are immutable sequences, similar to lists

- Tuples are like lists, but immutable
- Once created, cannot be changed
- Example:

(Round brackets)

```
date_of_birth = (13, 11, 1945)
```

- Tuples support many of the same operations as lists, but not those that modify them
  - Indexing using square brackets, e.g., `date_of_birth[1]  # is 11`
  - in, +, and * operators
  - Built in functions such as len, min, max (not del statement)

Use tuples for sequential data that won't change

- Tuples do not support the modifying methods:
  - append, reverse, sort, etc.
- Store and access values, but no modifying
  - You can always create a new tuple

```
date_of_birth = (13, 11, 1945)
lucky_year = random.randint(1900, 2022)
if date_of_birth[2] <= lucky_year:
    print("Congratulations, you are old enough!")
```

Lastly, how did we go with...

Write a program to ask the user for how many hours of study they did each day last week. Display the total, average, minimum and maximum hours as well as a nice text 'table' that shows how many hours they studied for.

Now do these next steps

- Practise writing algorithms and programs that use lists
- "Play" in the Python console with lists, practising with the things you've learned and experimenting with other things you can find
  - E.g., can you add two lists together?
    What about a list * a number?
    How could you create a 2nd list of numbers that was the same as the first, but each number was squared?
    Or…?