Data Structures

Python includes built-in variables types for a number of fundamental data structures, including lists, tuples, sets, and dictionaries (maps).

Data Structures: Lists

A *list* is an ordered collection of other variables. These variables can have different types. Lists definitions are enclosed within square brackets [and]

```
In [ ]: mylist = []                          # an empty list
        numbers = [12, 108, 21]              # a list of 3 integers
        somedata = ["text", 7, 0.34, True]  # a list containing 4 different variables of different types
```

Values in a list are accessed by specifying the *index* in square brackets - i.e. the position of the value in the list. Note: We always count from 0 in Python, so the first value in a list has index 0.

```
In [ ]: values = [34, 9, 12, 34]
        values[0]
```

We can count from the end of the list backwards by using negative index values. Index -1 is the last value, index -2 is the second last value, and so on.

```
In [ ]: values[-2]
```

If we try to access a value to an index that is beyond the length of the list, we will get an error message.

```
In [ ]: values[50]
```

Nesting: Lists can also be contained within other lists, which allows the construction of hierarchical data structures.

```
In [ ]: child1 = [12, 108, 23]
        child2 = [99, 4]
        child3 = ["a","b","c"]
        parent = [ child1, child2, child3 ]
        print(parent)
```

Values in nested lists can be accessed using multiple indexes in square brackets:

```
In [ ]: parent[0][2]
```

```
In [ ]: parent[2][1]
```

Slicing: Lists can also be *sliced* to access subsets of that list. The notation is [i:j], where *i* is the start index inclusive and *j* is the end index exclusive. Remember that we always count from index 0.

```
In [ ]: fulllist = [9, 12, 23, 18, 21]
        fulllist[0:2] # start at 1st item, end before 3rd item
```

```
In [ ]: fulllist[0:3]  # first three items
```

When slicing, the default for i is 0, default for j is the end of the string.

```
In [ ]: fulllist[1:] # all items from the 2nd one onwards
```

```
In [ ]: fulllist[:4] # start at 1st item, end before 5th item
```

Modifiying lists: Values in a list can be changed after the list is created by specifying the index and performing assignment.

```
In [ ]: values = [34, 9, 12, 34]
        values[2] = 5000
        print(values)
```

If we try to assign a value to an index that is beyond the length of the list, we will get an error message.

```
In [ ]: values[99] = 343
```

Instead, we can add a value to the end of a list using the *append()* function:

```
In [ ]: values.append("extra")
        print(values)
```

We can also concatenate two or more lists together using the plus + operator:

```
In [ ]: values + [11, 27]
```

```
In [ ]: ["A","B"] + ["Y","Z"]
```

Membership operators: The special 'in' keyword can be used to test if a value is contained in a list.

```
In [ ]: mylist = [3,6,9,12]
```

```
In [ ]: 3 in mylist
In [ ]: 27 in mylist
```

The logical 'out' operator can be used to test if a value is missing from a list.

```
In [ ]: 27 not in mylist
```

Related functions: A variety of built-in functions can be used with lists.

We can check the length of a list using the built-in *len()* function:

```
In [ ]: len(values)
```

We can sort the items in a list by a calling the *sort()* function on the list. Note that this sorts the list "in place" - i.e. the list itself is modified, rather than copied.

```
In [ ]: letters = ["b","d","a","c"]
        letters.sort()
        print(letters)
```

Data Structures: Tuples

Tuples are like lists but are "immutable" - this means that once they are created, they cannot be modified. Tuples are created using parenthesis notation (and).

```
In [ ]: suits = ("hearts", "diamonds", "spades", "clubs")
        suits
```

Values in tuples are also accessed using the same square bracket index notation that we saw for lists.

```
In [ ]: suits[0]
```

```
In [ ]: suits[-1]
```

Like lists, different types of variables can be contained within the same tuple.

```
In [ ]: t = (123, True, "UCD", 123.23)
        t
```

However, unlike lists, we cannot modify the tuple once it has been defined. If we try to assign a new value to an index in the tuple, we will get an error message.

```
In [ ]: t[3] = 3435
```

Data Structures: Sets

Sets are unordered lists which contain no duplicate values. They can be created from lists, strings or any other iterable value, using the *set* function. Sets do not have an order, so we cannot index into them by position.

```
In [ ]: mylist = [1,3,1,4,3,6,8,1,4,4]
        set(mylist)
```

```
In [ ]: set("abcddabcdaacbcc")
```

The 'in' membership operator also works on sets:

```
In [ ]: names = set(['Bill','Lisa','Ted'])
```

```
In [ ]: 'Bill' in names
```

```
In [ ]: 'Sharon' in names
```

We can then calculate unions, intersections and differences between pairs of sets.

```
In [ ]: x = set([1,2,3,4])
        y = set([3,4,5])
```

```
In [ ]: x.intersection(y)  # what values are in both x and y?
```

```
In [ ]: x.union(y)   # what are values are in either x or y, or both?
```

```
In [ ]: x.difference(y)    # what values are in x but not in y?
```

```
In [ ]: y.difference(x)    # what values are in y but not in x?
```

We can convert a *set* back to a *list* by calling the built-in *list()* function:

```
In [ ]: list(x)
```

Data Structures: Dictionaries

A *dictionary* (sometimes called a *map*) is a data structure containing an unordered set of *(key,value)* pairs. Each *key* is linked to a *value*. The keys and values can be any basic Python variable.

Dictionaries can be created using curly bracket notation { }, and can either be initially empty or populated with one or more pairs.

In [ ]:
```
d0 = {}                                           # create an empty dictionary
d1 = {"Ireland":"Dublin", "France":"Paris"}    # create a dictionary containing two pairs
d2 = {"age": 22, "name": "alice", 100 : False} # create a dictionary containing three pairs
```

Note that types of keys and values in a dictionary can be mixed

In [ ]:
```
mixedmap = {1:"ucd",0.8:False,"b":10,"c":"d"}
```

We can access a value in a dictionary by using the square bracket notation and specifying the corresponding key:

In [ ]:
```
d1["Ireland"]
```

In [ ]:
```
d2["name"]
```

If we try to access a value for a non-existent key in a dictionary, we will get an error message:

In [ ]:
```
d1["Sweden"]
```

To avoid this type of error, check for the presence of a key in a dictionary using the **in** operator:

In [ ]:
```
"Ireland" in d1
```

In [ ]:
```
"Sweden" in d1
```

We can easily add new values to a dictionary using square bracket notation and assignment. If a does not already exist for a given key, it will be added.

In [ ]:
```
d1["Germany"] = "Berlin"
d1
```

If a value for the key exists, the previous value will be over-written.

In [ ]:
```
d1["Ireland"] = "Galway"
d1
```

Dictionaries have various associated functions to access the keys and/or values.

In [ ]:
```
d1.keys() # get only the keys from a dictionary
```

In [ ]:
```
d1.values() # get only the values from a dictionary
```

In [ ]:
```
d1.items() # get all (key,value) pairs as tuples
```

We can check the number of key-value pairs in a dictionary using the built-in *len()* function:

In [ ]:
```
len(d1)
```

Using IPython Notebooks

IPython Notebooks provide an interactive web-based environment for writing and running Python code. We will use IPython notebooks for many of the labs and assignments in COMP41680.

IPython Notebooks have two fundamental types of cells:

1. *Markdown cells*: contain text content for explaining our notebook.
2. *Code cells*: allow us to type and run Python code.

Code Cells

To get started, review the lectures notes on how to start the IPython Notebook Server and create a new notebook in your web browser.

You can then run Python code in a code cell by hitting Shift-Enter or by pressing the ‘Play‘ button in the toolbar. You can modify and re-run code cells multiple times in any order.

When a code cell is executed, the code that it contains is sent to the ‘kernel‘ associated with the notebook (i.e. the Python engine running in the background). The results that are returned from this computation are then displayed in the notebook as the cell’s output. Note that some code will not have an output.

```
In [ ]: 1 + 2   # output is the result
```

```
In [ ]: x = 25  # no output here
```

```
In [ ]: print(x)  # output is produce by the print function
```

Markdown Cells

It can often be helpful to provide explanatory text in notebooks. Markdown (http://daringfireball.net/projects/markdown/) is a lightweight type of markup language with plain text formatting syntax which can be rendered as HTML. While there is no formal definition of the Markdown language, IPython supports a set of commands which are commonly used elsewhere (e.g. Github, Reddit).

To enter a IPython Markdown cell, choose the cell dropdown list in the toolbar and set the type to ‘Markdown’.

Paragraphs: Paragraphs in Markdown are just one or more lines of consecutive text followed by one or more blank lines.

This is a paragraph of text.

Styling Text: We can create a heading by adding one or more `#` symbols before the heading text. The number of `#`'s used will determine the size of the font.

Heading 1

Heading 2

Heading 3

Heading 4

You can format text as *italic* or **bold** using one or two `*`'s respectively.

This is normal text *This text will be in italics* **And this text will be in bold**

Unordered lists: We can make an unordered (bulleted) list by prefixing lines with either the `*` or `-` character.

- My item
- Another item
- Yet another item

- My item
- Another item
- Yet another item

Ordered lists: We can make an ordered (numbered) list by prefixing lines with a number.

1. My first item
2. Next item
3. Final item

We can create a hyperlink by wrapping text in square brackets (`[ ]`) and then wrapping the link in parentheses (`( )`).

Go to the UCD homepage (http://www.ucd.ie)

More Resources

IPython Resources

- IPython Official Documentation (http://ipython.readthedocs.org/en/stable/overview.html)

Markdown Resources

- Guide to Markdown Basics (https://help.github.com/articles/markdown-basics/) from Github
- Original Markdown Syntax Specification (http://daringfireball.net/projects/markdown/syntax/) from John Gruber
- Markdown Cheat Sheet (http://nestacms.com/docs/creating-content/markdown-cheat-sheet) from Nesta CMS

Python Basics

Variables in Python

Variable: A container in memory, which has a unique name or identifier, where you can store information. Start using the variable by assigning it a value, where the = symbol denotes *assignment*.

```
In [ ]: x = 100
```

Python has a number of variable naming rules:

- Can contain both letters and numbers, but must begin with a letter.
- Can contain the underscore character.
- Must not clash with reserved keywords.

We can display the value in a variable using the *print* function - note the use of parentheses.

```
In [ ]: print(x)
```

Each variable also has a *type*, indicating the nature of the value that it stores.

```
In [ ]: type(x)
```

```
In [ ]: type("UCD")
```

Numeric data can be *integers* (whole numbers) or *floats* (real values):

```
In [ ]: a = 3
        b = -125
        fx = 0.432
        fy = -24.23
```

Boolean values can be indicated by *True* or *False* - case sensitive! Can alternatively we can use '1' and '0'

```
In [ ]: answer = True
        test_value = False
```

A *string* value containing text is enclosed within either single quotes or double quotes - make sure you end with the same character:

```
In [ ]: some_text = "hello world"
        moredata = 'university college dublin, ireland'
```

We also use a special value *None* to indicate a variable containing an empty or "null" value:

```
In [ ]: current_value = None
```

Operators and Expressions

Python can be used as a simple calculator. It supports all basic mathematical operators, such as +, -, *, /

We can use combinations of these operators and values to create *expressions*, which are the building blocks of Python code.

```
In [ ]: a = 4 * 3 + 2
        a
```

```
In [ ]: b = a - 2
        b
```

We can also use operators to perform assignment and an operation on the same variable. Note that these lines include *comments* - everything from # onwards is ignored.

```
In [ ]: a += 2   # add 2 to the value currently in a and assign it back to a
        b -= 1   # subtract 1 from the value currently in b and assign it back to b
```

Parentheses can be used to control the *order* in which operators are applied:

```
In [ ]: 4 + 10 / 2  # Division will normally be applied first
```

```
In [ ]: (4 + 10)/2  # Use parenthesis to apply addition first, then division
```

There are also other operators, such as %,// and ** (modulo, floor division and ‘to the power’)

```
In [ ]: 3**2
```

Boolean Expressions

Any value or variable in Python can be tested for a 'truth value'. These will yield a value of True or False, depending on what we are testing - e.g. equality, inequality, greater/less than

```
In [ ]: x = 75 # don't confuse assignment with equality!
```

```
In [ ]: x == 75 # test for equality
```

```
In [ ]: x == 100 # test for equality
```

```
In [ ]: x != 100 # test for inequality
```

```
In [ ]: x < 1000 # less than
```

```
In [ ]: x > 0 # greater than
```

Python contains boolean operators to create more complex boolean expressions:

- *not x*: returns False if x is True, returns True if x is False
- *x and y*: if both x and y are True then return True, otherwise False
- *x or y*: if either x or y are True then return True, otherwise False

In [ ]:
```
True and True
```

In [ ]:
```
True and False
```

In [ ]:
```
True or False
```

In [ ]:
```
not True
```

In [ ]:
```
x > 0 and x < 100
```

Data Structures

Python includes built-in variables types for a number of fundamental data structures, including lists, tuples, sets, and dictionaries (maps).

Data Structures: Lists

A *list* is an ordered collection of other variables. These variables can have different types. Lists definitions are enclosed within square brackets [and]

In [ ]:
```
mylist = []                          # an empty list
numbers = [12, 108, 21]              # a list of 3 integers
somedata = ["text", 7, 0.34, True]  # a list containing 4 different variables of different types
```

Values in a list are accessed by specifying the *index* in square brackets - i.e. the position of the value in the list. Note: We always count from 0 in Python, so the first value in a list has index 0.

In [ ]:
```
values = [34, 9, 12, 34]
values[0]
```

```
We can count from the end of the list backwards by using negative index values. Index -1 is the last value, index -2 is the second last value, and so on.
```

In [ ]:
```
values[-2]
```

If we try to access a value to an index that is beyond the length of the list, we will get an error message.

In [ ]:
```
values[50]
```

Nesting: Lists can also be contained within other lists, which allows the construction of hierarchical data structures.

In [ ]:
```
child1 = [12, 108, 23]
child2 = [99, 4]
child3 = ["a","b","c"]
parent = [ child1, child2, child3 ]
print(parent)
```

Values in nested lists can be accessed using multiple indexes in square brackets:

In [ ]:
```
parent[0][2]
```

In [ ]:
```
parent[2][1]
```

Slicing: Lists can also be *sliced* to access subsets of that list. The notation is [i:j], where *i* is the start index inclusive and *j* is the end index exclusive. Remember that we always count from index 0.

In [ ]:
```
fulllist = [9, 12, 23, 18, 21]
fulllist[0:2] # start at 1st item, end before 3rd item
```

In [ ]:
```
fulllist[0:3]  # first three items
```

When slicing, the default for i is 0, default for j is the end of the string.

In [ ]:
```
fulllist[1:] # all items from the 2nd one onwards
```

In [ ]:
```
fulllist[:4] # start at 1st item, end before 5th item
```

Modifiying lists: Values in a list can be changed after the list is created by specifying the index and performing assignment.

In [ ]:
```
values = [34, 9, 12, 34]
values[2] = 5000
print(values)
```

If we try to assign a value to an index that is beyond the length of the list, we will get an error message.

In [ ]:
```
values[99] = 343
```

Instead, we can add a value to the end of a list using the *append()* function:

In [ ]:
```
values.append("extra")
print(values)
```

We can also concatenate two or more lists together using the plus + operator:

In [ ]:
```
values + [11, 27]
```

In [ ]:
```
["A","B"] + ["Y","Z"]
```

Membership operators: The special 'in' keyword can be used to test if a value is contained in a list.

In [ ]:
```
mylist = [3,6,9,12]
```

In [ ]:
```
3 in mylist
```

In [ ]:
```
27 in mylist
```

The logical 'out' operator can be used to test if a value is missing from a list.

In [ ]:
```
27 not in mylist
```

Related functions: A variety of built-in functions can be used with lists.

We can check the length of a list using the built-in *len()* function:

In [ ]:
```
len(values)
```

We can sort the items in a list by a calling the *sort()* function on the list. Note that this sorts the list "in place" - i.e. the list itself is modified, rather than copied.

In [ ]:
```
letters = ["b","d","a","c"]
letters.sort()
print(letters)
```

Data Structures: Tuples

Tuples are like lists but are "immutable" - this means that once they are created, they cannot be modified. Tuples are created using parenthesis notation (and).

In [ ]:
```
suits = ("hearts", "diamonds", "spades", "clubs")
suits
```

Values in tuples are also accessed using the same square bracket index notation that we saw for lists.

In [ ]:
```
suits[0]
```

In [ ]:
```
suits[-1]
```

Like lists, different types of variables can be contained within the same tuple.

In [ ]:
```
t = (123, True, "UCD", 123.23)
t
```

However, unlike lists, we cannot modify the tuple once it has been defined. If we try to assign a new value to an index in the tuple, we will get an error message.

In [ ]:
```
t[3] = 3435
```

Data Structures: Sets

Sets are unordered lists which contain no duplicate values. They can be created from lists, strings or any other iterable value, using the *set* function. Sets do not have an order, so we cannot index into them by position.

In [ ]:
```
mylist = [1,3,1,4,3,6,8,1,4,4]
set(mylist)
```

In [ ]:
```
set("abcddabcdaacbcc")
```

The 'in' membership operator also works on sets:

In [ ]:
```
names = set(['Bill','Lisa','Ted'])
```

In [ ]:
```
'Bill' in names
```

In [ ]:
```
'Sharon' in names
```

We can then calculate unions, intersections and differences between pairs of sets.

In [ ]:
```
x = set([1,2,3,4])
y = set([3,4,5])
```

In [ ]:
```
x.intersection(y)  # what values are in both x and y?
```

In [ ]:
```
x.union(y)   # what are values are in either x or y, or both?
```

In [ ]:
```
x.difference(y)    # what values are in x but not in y?
```

In [ ]:
```
y.difference(x)    # what values are in y but not in x?
```

We can convert a *set* back to a *list* by calling the built-in *list()* function:

In [ ]:
```
list(x)
```

Data Structures: Dictionaries

A *dictionary* (sometimes called a *map*) is a data structure containing an unordered set of *(key,value)* pairs. Each *key* is linked to a *value*. The keys and values can be any basic Python variable.

Dictionaries can be created using curly bracket notation { }, and can either be initially empty or populated with one or more pairs.

In [ ]:
```
d0 = {}                                       # create an empty dictionary
d1 = {"Ireland":"Dublin", "France":"Paris"}    # create a dictionary containing two pairs
d2 = {"age": 22, "name": "alice", 100 : False} # create a dictionary containing three pairs
```

Note that types of keys and values in a dictionary can be mixed

In [ ]:
```
mixedmap = {1:"ucd",0.8:False,"b":10,"c":"d"}
```

We can access a value in a dictionary by using the square bracket notation and specifying the corresponding key:

In [ ]:
```
d1["Ireland"]
```

In [ ]:
```
d2["name"]
```

If we try to access a value for a non-existent key in a dictionary, we will get an error message:

In [ ]:
```
d1["Sweden"]
```

To avoid this type of error, check for the presence of a key in a dictionary using the **in** operator:

```
In [ ]: "Ireland" in d1
```

```
In [ ]: "Sweden" in d1
```

We can easily add new values to a dictionary using square bracket notation and assignment. If a does not already exist for a given key, it will be added.

```
In [ ]: d1["Germany"] = "Berlin"
        d1
```

If a value for the key exists, the previous value will be over-written.

```
In [ ]: d1["Ireland"] = "Galway"
        d1
```

Dictionaries have various associated functions to access the keys and/or values.

```
In [ ]: d1.keys() # get only the keys from a dictionary
```

```
In [ ]: d1.values() # get only the values from a dictionary
```

```
In [ ]: d1.items() # get all (key,value) pairs as tuples
```

We can check the number of key-value pairs in a dictionary using the built-in *len()* function:

```
In [ ]: len(d1)
```