Ulster University

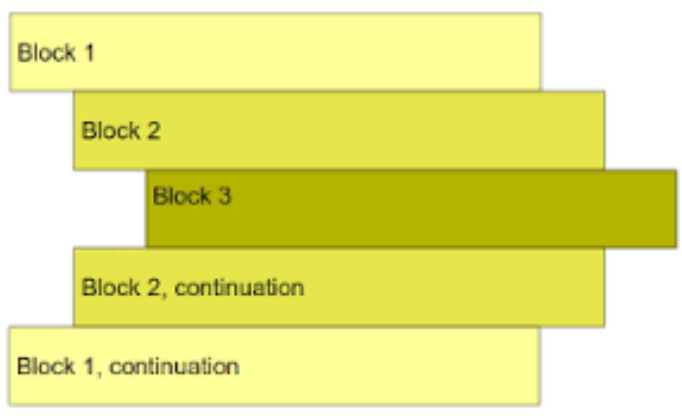
COM738 - Data Science Foundations

Week 2

Python Fundamentals (Part 2)

Module Co-Ordinator: Dr Priyanka Chaurasia

Block Nesting



Keep this figure in mind when creating Python blocks

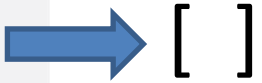
```
for i in [1,2,3,4,5]:
    print(i) #first line in 'for i' block
    for j in [6,7,8,9,10]:
        print(j)
        print(i + j) #Last line in 'for j' block
    print(i) #Last line in 'for i' block
print("end") #First line after loops.
```

- Block is a group of statements in a program or script
- Generally, blocks can contain blocks as well, so we get a nested block structure
- Python uses indentation to highlight the blocks of code
- Python code structures by indentation
- Whitespace is used for indentation in Python
- To indicate a block of code in Python, you must indent each line of the block by the same whitespace
- All statements with the same distance to the right belong to the same block of code
- If a block has to be more deeply nested, it is simply indented further to the right

Ulster University

Quick Recap

```
thislist = ["apple", "banana", "cherry"]
print(thislist[1])
```



[ ]

Access a member of List

```
thislist.append("orange")
print(thislist)
```

Insert an item as the second position:

```
thislist.insert(1, "orange")
print(thislist)
```

Remove

```
thislist.remove("banana")
print(thislist)
```

The pop() method removes the specified index, (or the last item if index is not specified):

```
thislist.pop()
print(thislist)
```

Ulster University

Quick Recap

```
thistuple = ("apple", "banana", "cherry")
```

( )

```
print(thistuple[1])
```

Access a member of Tuple

- Once a tuple is created, you cannot add items to it
- Tuples are **unchangeable**
- You **can't add a new index** or **change a value at index**

```
thistuple[1] = "blackcurrant"
```

```
thistuple[3] = "orange"
```

```
TypeError: 'tuple' object does not support item assignment
```

The del keyword can delete the tuple completely:

```
del thistuple
```

tuple() Constructor

```
thistuple =
tuple(("apple", "banana", "cherry"))
# note the double round-brackets
print(thistuple)
```

Ulster University

Quick Recap

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

{ }

- Set is a collection which is **unordered** and **unindexed**
- **So,** cannot access items in a set by referring to an index

```
for x in thisset:
  print(x)
```

Change Items

- Once a set is created, **you cannot change its items, but you can add new items**
- To add one item to a set use the add() method
  - `thisset.add("orange")`
- To add more than one item to a set use the update() method
  - `thisset.update(["orange", "mango", "grapes"])`

Remove Item

`remove()`, or the `discard()` method

```
thisset.remove("banana")
```

Note: If the item to remove does not exist, `remove()` will raise an error

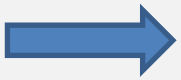
```
thisset.discard("banana")
```

Note: If the item to remove does not exist, `discard()` will **NOT** raise an error

Ulster University

Quick Recap

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```



{ } with ***Key*** : ***Value*** pair

Note: In this case Key is a string so should be within " "

Looping:

Print all *key* names

```
for x in thisdict:
  print(x)
```

Accessing Items:

```
x = thisdict[ "model" ]
```

Or

```
x = thisdict.get("model")
```

Print all *values*

```
for x in thisdict:
  print(thisdict[x])
```

Change Values:

```
thisdict["year"] = 2018
```

```
for x in thisdict.values():
  print(x)
```

Both *keys* and *values*, by using the items() function

```
for x, y in thisdict.items():
  print(x, y)
```

Ulster University

Quick Recap

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

```
1  a?
```

```
Type:          list
String form:   [1, 2, 3]
Length:        3
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
```

When you are unsure of the datatype of a variable→ use **?** with the variable name

Ulster University

Ulster University

Functions

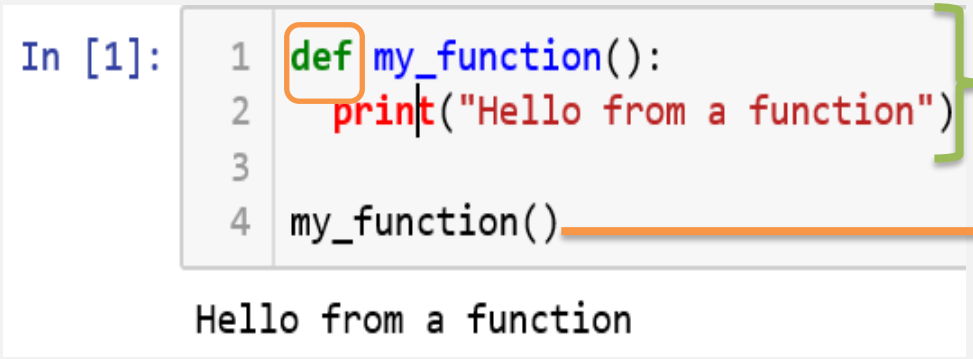
ulster.ac.uk

Python Fundamentals

- Defining and Using **Functions**
  - For ***Readability*** and ***Reusability***
  - Using Functions
    - Called using Parenthesis ( )
    - E.g. print('hello')
      - "print" is the **function name**
      - 'hello' is an **argument** (a value passed to the function for processing)

Ulster University

Python Fundamentals



First define function using **def**

Then call function using defined name

Keyword Arguments

- Specified by name
- E.g. print (address, postcode, telNum)

Ulster University

Keyword vs Non-keyword

Keyword:

- ✓ name=value instead of using positional syntax
- ✓ Keyword (named) arguments

  e.g. my_function(a=12, b="abc")

```
1  from math import sqrt
2
3  def quadratic(a, b, c):
4      x1 = -b / (2*a)
5      x2 = sqrt(b**2 - 4*a*c) / (2*a)
6      return (x1 + x2), (x1 - x2)
```

```
In [20]:  1  quadratic(a=31, b=93, c=62)

Out[20]:  (-1.5, -2.5)
```

Ulster University

NOTE: Order doesn't matter when they're passed by their name

Keyword vs Non-keyword

```
1  from math import sqrt
2
3  def quadratic(a, b, c):
4      x1 = -b / (2*a)
5      x2 = sqrt(b**2 - 4*a*c) / (2*a)
6      return (x1 + x2), (x1 - x2)
```

Non-keyword:

✓ Arguments as positional arguments

```
In [19]:  1  quadratic(31, 93, 62)
Out[19]:  (-1.5, -2.5)
```

```
In [21]:  1  quadratic(62, 93, 31)
Out[21]:  (-0.75, -1.25)
```

NOTE: Order of these arguments matters when they're passed positionally

Ulster University

Python Fundamentals

- Defining Functions

```
def fibonacci(N):
    L = []                  #Create a new empty list
    a,b = 0,1               #Create two variables to hold the starting sequence
    while len(L) < N:       #While the length of the list is less than the (hopefully) number provided
        a,b = b, a+b        #Make a = b, and b = a+b
        L.append(a)         #Append a to the list
    return L                #Once the loop is complete, return the list.
```

```
fibonacci(10)
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

- **Note:**
  - No need to declare any input or return type
  - Multiple values can easily be returned in a tuple, e.g.:
    - *return houseNumber, street, county, country, postcode*

Ulster University

Python Fundamentals

A note on assignment

```
1  a= 10
2  b= 5
3  print ('a=', a,'b=', b)
```

```
('a=', 10, 'b=', 5)
```

```
1  a, b = 10, 5
2  print ('a=', a,'b=', b)
```

```
('a=', 10, 'b=', 5)
```

Ulster University

Note: Indentation Error

```
In [7]:
1 def fibnoacci (N):
2 L= []
3 a, b= 0, 1
4 while len(L)<N:
5     a,b = b, a+b
6     L.append(a)
7 return L
```

```
  File "<ipython-input-7-b63eed196e22>", line 2
    L= []
    ^
IndentationError: expected an indented block
```

Variables L, a, and b are within the def function so they have to be indented to let the interpreter know where they belong to

```
1 def fibnoacci (N):
2     L = []
3     a, b= 0, 1
4     while len(L)<N:
5     a,b = b, a+b
6     L.append(a)
7 return L
```

```
  File "<ipython-input-10
    a,b = b, a+b
    ^
IndentationError: expected an indented block
```

Same problem here as well. The code below while has to be written after a tab, to reflect belonging to while loop

Ulster University

Correct format

```
In [12]:
1  def fibnoacci (N):
2      L = []
3      a, b= 0, 1
4      while len(L)<N:
5          a,b = b, a+b
6          L.append(a)
7  return L

  File "<ipython-input-12-3ffcae6e1102>", line 7
    return L
SyntaxError: 'return' outside function
```

Because *return* is part of the function so should be inside

```
In [11]:
1  def fibnoacci (N):
2      L = []
3      a, b= 0, 1
4      while len(L)<N:
5          a,b = b, a+b
6          L.append(a)
7      return L
```

Ulster University

Python Fundamentals

Default Argument Values

- Facilitate speed and ease of use, yet provide flexibility
- When there are values a function should use **most** times, but not always
- These **default values** do not need to be specified when calling the function, e.g.

```
def fibonacci(N, firstValue=5, secondValue=10):
    L = []                  #Create a new empty list
    a,b = firstValue,secondValue #Create two variables to hold the starting sequence
    while len(L) < N:       #While the length of the list is less than the (hopefully) number provided
        a,b = b, a+b        #Make a = b, and b = a+b
        L.append(a)         #Append a to the list
    return L                #Once the loop is complete, return the list.
```

```
fibonacci(10)
```

```
[10, 15, 25, 40, 65, 105, 170, 275, 445, 720]
```

Ulster University

NOTE: See first value only passed rest is default used when function called

Python Fundamentals

Default Argument Values

- They can be provided in order:

```
fibonacci(10, 1, 2)
```

```
[2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
```

- If they are specifically named, they can be provided in any order **(**But must be **AFTER** the non-keyword args**):**

```
fibonacci(10, secondValue=2, firstValue=1)
```

```
[2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
```

Ulster University

Python Fundamentals

Default Argument Values

- You can view the default arguments of functions
- Simply write the function call, e.g. animals. sort()
- Place the cursor within the parenthesis
- Press **Shift + Tab**

```
In [1]: animals = ["cow","pig","cat"]

        animals.sort()
```

```
Docstring: L.sort(key=None, reverse=False) -> None -- stable sort *IN PLACE*
Type:      builtin_function_or_method
```

Ulster University

- This example shows two parameters and their default values

Python Fundamentals

Flexible Arguments:

***args** and ****kwargs** → **Special syntax**

- Facilitate writing a function without knowing how many arguments a user will pass
- **Used in function definitions** to pass a variable number of arguments to a function
- The name "args" and "kwargs" are just variable names used by convention
- **Asterisks are the important part**

* before a variable = "expand this as a sequence"

** = "expand this as a dictionary"
{Because Dictionary variable are key value pair}

Ulster University

Python Fundamentals

*args→ non-keyword arguments

- Single asterisk form (*args) is used to pass a **non-keyworded**, variable-length argument list
- When not wanting to restrict the number of **non-keyword arguments** that can be accepted

**kwargs→ keyword arguments

- Double asterisk form is used to pass a **keyworded**, variable-length argument list
- When not wanting to restrict the number of **keyword arguments** that can be accepted

Ulster University

Python Fundamentals

Example:

```
def foo(*positional, **keywords):
    print("Positional:", positional)
    print("Keywords:", keywords)
```

* use→ non-keyword argument

```
1  foo('one', 'two', 'three')
```

```
Positional: ('one', 'two', 'three')
Keywords: {}
```

** use→ keyword argument

```
1  foo(a='one', b='two', c='three')
```

```
Positional: ()
Keywords: {'a': 'one', 'c': 'three', 'b': 'two'}
```

Both

non-keywords passed

keywords passed

```
1  foo('one','two',c='three',d='four')
```

```
Positional: ('one', 'two')
Keywords: {'c': 'three', 'd': 'four'}
```

Ulster University

Python Fundamentals

Python Lambda

```
1  x = lambda a : a + 10
2  print(x(5))
```

```
15
```

- Is a small anonymous function
- Syntax→ lambda arguments : *expression*
- Can take any number of arguments, but can only have one expression

```
1  x = lambda a, b, c : a + b + c
2  print(x(5, 6, 2))
```

```
13
```

- Can take any number of arguments, but can only have one expression

```
multiply = lambda x, y: x*y

multiply(4,5)
```

lambda way

```
20
```

```
def multiply(x,y):
    return x*y
multiply(4,5)
```

function way

```
20
```

Ulster University

Iterators

Python Fundamentals

Iterators

- Facilitate proceeding through a dataset and repeating similar calculations
- One task – return the "next" item in an iterable

```
for i in range(10):
    print(i, end=' ')
```

```
0 1 2 3 4 5 6 7 8 9
```

- The end=' ' is just to say that you want a space after the end of the statement (i.e. after each value) instead of a new line character

```
print(i, end = ", ")
```

```
0, 1, 2, 3, 4,
```

Value separated by commas in this case

Ulster University

Python Fundamentals

- Iterating over lists

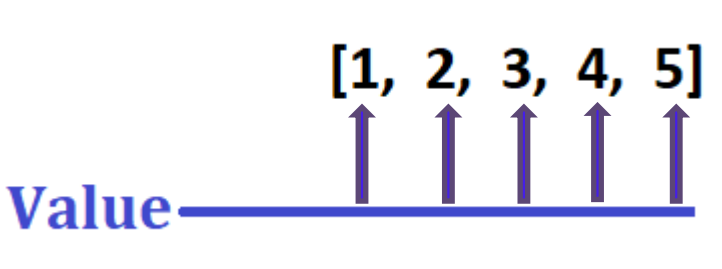
```
for value in [1,2,3,4,5]:
    print(value + 1, end=' ')
```

```
2 3 4 5 6
```

- When writing *for [value]* in *[list],* the Python interpreter checks whether it has an *iterator* interface
- This can be checked yourself by:

```
iter([1,2,3,4,5])
```

```
<list_iterator at 0x1a5bfac58d0>
```



Ulster University

Note on range ( ) function

- Returns a sequence of numbers
- Starting from 0 by default
- Increments by 1 (by default)
- **Stops before a specified number**

```
1 range(10)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

0 to 9

```
1 range (0, 10)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

0 to 9

```
1 range(1,10)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

1 to 9

```
1 range (0, 11)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

0 to 10

Ulster University

Note on range ( ) function

```
1  print(range(1,10))
```

```
range(1, 10)
```

In this case just the range function with arguments passed is printed and not the actual sequence

```
1  print(*range(1,10))
```

```
1 2 3 4 5 6 7 8 9
```

When * is used→ the sequence get **expanded** and passed element by element to the print function

NOTE: * expand the arguments as sequence

Ulster University

Python Fundamentals

- The **iterator object** provides the functionality required by the *for* loop
- It is a container providing access to the next object (**if valid**) using the **next( )** function

```
I = iter([1,2,3,4,5])
print(next(I))
print("next..")
print(next(I))
print("etc..")
```

```
1
next..
2
etc..
```

```
for i in range(10):
    print(i, end=' ')
```

```
0 1 2 3 4 5 6 7 8 9
```

So if you do this, *for* internally calls the next() and check if more values is there, it will retrieve

Ulster University

Python Fundamentals

Why is this useful?

- It allows Python to treat things as **lists** when they are not actually lists (e.g. the *range* object)
- The *range*( ) function returns a range object:

```
range(10)
```

```
range(0, 10)
```

- Like a list, the *range* object exposes an iterator:

```
iter(range(10))
```

```
<range_iterator at 0x1a5bfaa44f0>
```

- So Python can treat it as if it is a list:

```
for v in range(10):
    print(v, end=" ")
```

```
0 1 2 3 4 5 6 7 8 9
```

Ulster University

Python Fundamentals

Why is this useful?

- **The full list is never explicitly created**
- This is very memory efficient, as large lists could take up massive amounts of memory
- E.g.

```
N= 10**10 #This is 10,000,000,000
for t in range(N):
    if t >= 10: break
    print (t, end = " ")
```

```
0 1 2 3 4 5 6 7 8 9
```

- Iterators can also be infinite, with no end value
- E.g. The *count* function of the ***itertools*** library

```
from itertools import count
for i in count():
    if i >= 5: break
    print(i, end = ", ")
```

```
0, 1, 2, 3, 4,
```

Ulster University

Python Fundamentals

Other Useful Iterators

enumerate

- Helps keep track of the current index
- Instead of:

```
animals = ["cat","dog","cow"]
for i in range(len(animals)):
    print(i, animals[i])
```

```
0 cat
1 dog
2 cow
```

- len(animals)→ returns 3
- Then range(3) returns: [0, 1, 2]

- enumerate makes this much more elegant:

```
animals = ["cat","dog","cow"]
for i, val in enumerate(animals):
    print(i, val)
```

```
0 cat
1 dog
2 cow
```

Enumerate returns two values:

1. index
2. actual value in the list

Ulster University

Python Fundamentals

Other Useful Iterators

zip

- To iterate over **multiple lists simultaneously**

```
animals_farmA = ["cat","dog","cow"]
animals_farmB = ["moose","sheep","bull","mouse","lizard"]
for aVal, bVal in zip(animals_farmA, animals_farmB):
    print(aVal, bVal)
```

```
cat moose
dog sheep
cow bull
```

NOTE:

- Can be performed with **more than 2 iterables**
- The **shortest iterable determines the length**

Ulster University

Python Fundamentals

→ map

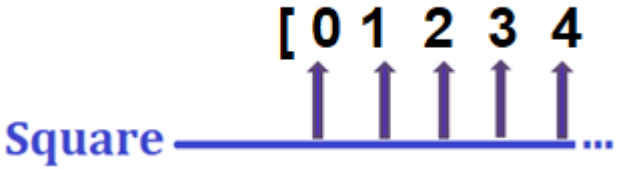
- Takes a function, and applies it to the values in an iterator
- Therefore 2 arguments:

  - function that needs to be applied

  - sequence on which it needs to be applied

```
square = lambda x: x ** 2 # The function to be used
for val in map(square, range(10)):
    print(val, end=' ')
```

```
0 1 4 9 16 25 36 49 64 81
```



Take each value from ***range*** and apply the function ***square***

→ filter

- Similar, but only passes through values for which the filter function evaluates to *True*
- Also takes 2 arguments

Ulster University

```
is_even = lambda x: x % 2 == 0
for val in filter(is_even, range(10)):
    print(val, end= ',')
```

```
0,2,4,6,8,
```

Python Fundamentals

Iterators as Function Arguments

- *args and **kwargs can be used to pass sequences and dictionaries to functions
- ***args can accept iterators**:
  - e.g.

```
print(*range(10))
```

```
0 1 2 3 4 5 6 7 8 9
```

* for list
** for dictionary

Here it means you are passing a variable length arguments to the print function

Ulster University

Python Fundamentals

Iterators as Function Arguments

What would the following output?

```
print(*map(lambda x: x**2, range(10)))
```

Ulster University

Python Fundamentals

Iterators as Function Arguments

What would the followingoutput?

```
print(*map(lambda x: x**2, range(10)))
```

Answer:

```
0 1 4 9 16 25 36 49 64 81
```

Ulster University

Python Fundamentals

Specialised Iterators: *itertools*

- The *itertools* module contains many useful iterators
- The official docs have a full list of iterators:
- https://docs.python.org/3.6/library/itertools.html
- Examples:
  - permutations
    - Iterates over all permutations (orders/arrangements) of a sequence

```
from itertools import permutations
p = permutations(range(4))
print(*p)
```

```
(0, 1, 2, 3) (0, 1, 3, 2) (0, 2, 1, 3) (0, 2, 3, 1) (0, 3, 1, 2) (0, 3, 2,
 1) (1, 0, 2, 3) (1, 0, 3, 2) (1, 2, 0, 3) (1, 2, 3, 0) (1, 3, 0, 2) (1, 3,
 2, 0) (2, 0, 1, 3) (2, 0, 3, 1) (2, 1, 0, 3) (2, 1, 3, 0) (2, 3, 0, 1) (2,
 3, 1, 0) (3, 0, 1, 2) (3, 0, 2, 1) (3, 1, 0, 2) (3, 1, 2, 0) (3, 2, 0, 1)
 (3, 2, 1, 0)
```

Ulster University

Python Fundamentals

Specialised Iterators: *itertools*

combinations

- Iterates over all unique combinations of N values in a list:

```
from itertools import combinations
c = combinations(range(4),2)
print(*c)
```

```
(0, 1) (0, 2) (0, 3) (1, 2) (1, 3) (2, 3)
```

product

- Iterates over all sets of pairs between 2 iterables:

```
from itertools import product
gender = ["male","female"]
occupation = ["data scientist","project manager","tester","developer"]
p = product(gender,occupation)
print(*p)
```

```
('male', 'data scientist') ('male', 'project manager') ('male', 'tester')
 ('male', 'developer') ('female', 'data scientist') ('female', 'project mana
ger') ('female', 'tester') ('female', 'developer')
```

- Today's practical will give you experience of these and more

Ulster University

Python Fundamentals

Specialised Iterators: *itertools*

```
from itertools import combinations, permutations, product

animals = ["cat","dog","mouse"]
size = ["small","large"]

print("Permutations: ", *permutations(animals))
print("")
print("Combinations (Groups of 2): ", *combinations(animals,2))
print("")
print("Product: ", *product(size,animals))
```

```
Permutations:  ('cat', 'dog', 'mouse') ('cat', 'mouse', 'dog') ('dog', 'cat',
 'mouse') ('dog', 'mouse', 'cat') ('mouse', 'cat', 'dog') ('mouse', 'dog', 'ca
t')

Combinations (Groups of 2):  ('cat', 'dog') ('cat', 'mouse') ('dog', 'mouse')

Product:  ('small', 'cat') ('small', 'dog') ('small', 'mouse') ('large', 'cat')
('large', 'dog') ('large', 'mouse')
```

Ulster University

Exception Handling

Python Fundamentals

Errors and Exceptions

Common **errors** include:

- **Syntax Errors** – Where code is not valid Python code
- **Runtime Errors** – Where syntactically valid code fails to execute (potentially due to user input)
- **Semantic/Logic Errors** – Code executes, but result is not as expected

Ulster University

Python Fundamentals

Syntax Errors

```
1  myList= [1, 2, 3]
2  for i in myList
3      print(i)
```

```
  File "<ipython-input-31-ca5816c11926>", line 2
    for i in myList
                   ^
SyntaxError: invalid syntax
```

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

```
1
2
3
```

Syntax errors occur when the parser detects an incorrect statement

Ulster University

Python Fundamentals

Runtime Errors

- Can happen in many different ways, including:

  Referencing an undefined variable or function:

```
pprintAll(1,"hello",a=2,b="hello2")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-19-3fef1ddcf421> in <module>()
----> 1 pprintAll(1,"hello",a=2,b="hello2")

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

- Attempting an undefined operation:

```
1 + 'test'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-20-3030544d32e1> in <module>()
----> 1 1 + 'test'

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

Ulster University

→ int and str can not be added

Python Fundamentals

Runtime Errors

- Attempting to compute a mathematically ill-defined result:

```
1 / 0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-21-b710d87c980c> in <module>()
----> 1 1 / 0

ZeroDivisionError: division by zero
```

- Accessing an out of range index:

```
animals = ['cow','dog','cat']

print(animals[6])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-24-501e9e0e281e> in <module>()
      1 animals = ['cow','dog','cat']
      2
----> 3 print(animals[6])

IndexError: list index out of range
```

Ulster University

Python Fundamentals

Exceptions versus Syntax Errors

```
1  print( 0 / 0 ))
```

```
  File "<ipython-input-34-c3931f671051>", line 1
    print( 0 / 0 ))
                  ^
SyntaxError: invalid syntax
```

- Arrow indicates where the parser ran into **syntax error**

```
1  print( 0 / 0)
```

```
ZeroDivisionErrorTraceback (most recent call last)
<ipython-input-35-b7f65c155a3b> in <module>()
----> 1 print( 0 / 0)

ZeroDivisionError: integer division or modulo by zero
```

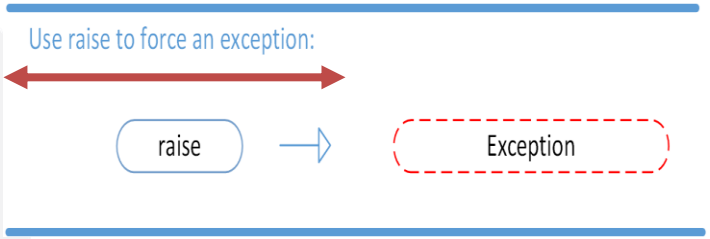
See the last line for what kind of error it throws

- **Exception error** occurs whenever syntactically correct Python code results in an error
- Last line of the message indicated what type of exception error you ran into

Ulster University

Python Fundamentals

Raising an Exception



- We can use raise to throw an exception if a certain condition occurs
- The statement can be complemented with a custom exception

```
1  x = 10
2  if x > 5:
3      raise Exception('x should not exceed 5. The value of x was: {}'.format(x))
4      print("in exception")
```

```
ExceptionTraceback (most recent call last)
<ipython-input-37-b4866de1d54d> in <module>()
      1 x = 10
      2 if x > 5:
----> 3     raise Exception('x should not exceed 5. The value of x was: {}'.format(x))
      4     print("in exception")

Exception: x should not exceed 5. The value of x was: 10
```

➢ **Program comes to a halt and displays our exception to screen, offering clues about what went wrong**

Note:

- Print statement doesn't get executed
- Because as soon as you enter if, it raises an exception

Ulster University

Python Fundamentals

Raising an Exception→ raise statement

```
1  x = 10
2  if x > 5:
3      print("in exception")
4      raise Exception('x should not exceed 5. The value of x was: {}'.format(x))
5
```

```
in exception
```

→ Print statement gets executed

```
ExceptionTraceback (most recent call last)
<ipython-input-38-6108bca6d33f> in <module>()
      2 if x > 5:
      3     print("in exception")
----> 4     raise Exception('x should not exceed 5. The value of x was: {}'.format(x))
      5

Exception: x should not exceed 5. The value of x was: 10
```

Ulster University

Python Fundamentals

Raising an Exception: Example

- Raising exceptions within your code **will help to identify the source of errors**
- For example, in our Fibonacci method, a negative number will not currently throw an error:

```
def fibonacci(N):
    L=[]
    a,b = 0,1
    while len(L) < N:
        a,b = b,a+b
        L.append(a)
    return L

fibonacci(-5)
```

```
[]
```

- **However, the output is not as we would expect (an empty list)**
- **Even there was a problem the user doesn't know what happened**
- **Just an empty list**
- **Why is this list empty?**

Ulster University

Python Fundamentals

Raising an Exception: Example

- **Resolve this by anticipating this possibility and raising an exception:**

```
def fibonacci(N):
    if N < 0:    #Check for a potential problem
        raise ValueError("N must be non-negative") #Provide an informative error message.
    L=[]
    a,b = 0,1
    while len(L) < N:
        a,b = b,a+b
        L.append(a)
    return L

fibonacci(-5)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-20-b9771790e6e2> in <module>()
      9     return L
     10
---> 11 fibonacci(-5)

<ipython-input-20-b9771790e6e2> in fibonacci(N)
      1 def fibonacci(N):
      2     if N < 0:    #Check for a potential problem
----> 3         raise ValueError("N must be non-negative") #Provide an informative error message
      4     L=[]
      5     a,b = 0,1

ValueError: N must be non-negative
```

Ulster University

Python Fundamentals

Raising Exceptions: raise

→We could handle this error in a try… except block

```
def fibonacci(N):
    if N < 0:   #Check for a potential problem
        raise ValueError("N must be non-negative") #Provide an informative error message.
    L=[]
    a,b = 0,1
    while len(L) < N:
        a,b = b,a+b
        L.append(a)
    return L

try:
    print("Trying this..")
    fibonacci(-5)
except ValueError:
    print("Bad value - must try another value")
```

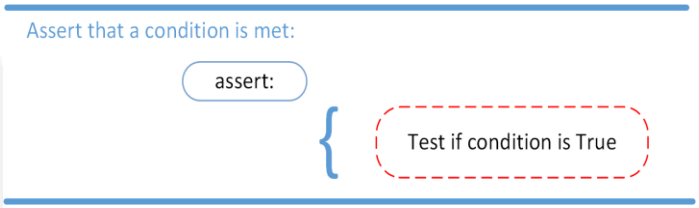
- **Put the problematic code in try block**
- **Tells where exception happened**

Jump here

```
Trying this..
Bad value - must try another value
```

Ulster University

Python Fundamentals



AssertionError Exception

- **Instead of waiting for a program to crash midway:**

  → You can also start by making an assertion in Python

- **We assert that a certain condition is met**:

  →If this condition turns out to be **True**, then program can continue

  → If the condition turns out to be **False**, you can have the **program throw an AssertionError exception**

```
1 import sys
2 assert ('linux' in sys.platform), "This code runs on Linux only."
```

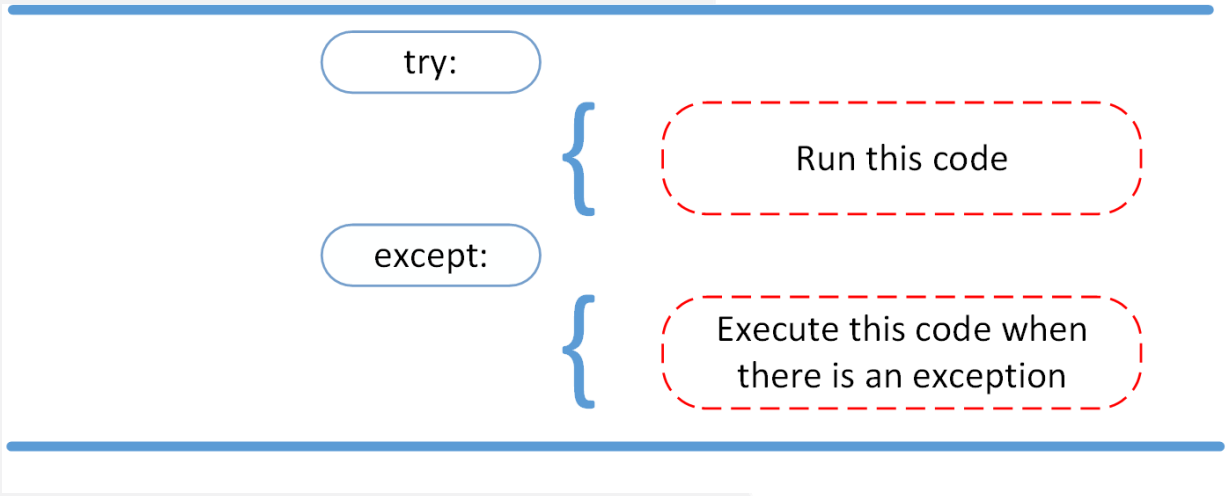
```
AssertionErrorTraceback (most recent call last)
<ipython-input-39-e31e8d82fd13> in <module>()
      1 import sys
----> 2 assert ('linux' in sys.platform), "This code runs on Linux only."

AssertionError: This code runs on Linux only.
```

Ulster University

Python Fundamentals

Catching Exceptions

- ✓ The **try block** lets you test a block of code for errors
- ✓ The **except block** lets you handle the error
- ✓ The **finally block** lets you execute code, regardless of the result of the try- and except blocks



Ulster University

Python Fundamentals

Catching Exceptions

"Try...Except" clause

```
try:
    print("Lines in this block will be executed first")
except:
    print("Lines in this block will only be executed if there is an error in the try block")
```

```
Lines in this block will be executed first
```

- try blocks will be executed up until there is an error
- Execution will then switch to the except block if exception occurs

```
try:
    print("Lines in this block will be executed first")
except:
    print("Lines in this block will only be executed if there is an error in the try block")
```

```
Lines in this block will only be executed if there is an error in the try block
```

Ulster University

Python Fundamentals

Two more clauses:

- **else** - Will only execute if the try block succeeds
  - E.g. Operations on a successfully opened file
- **finally** – Will always execute
  - E.g. Clean-up operations, ensuring files are closed or saved, etc.

- Used in this order:

```
try:
    print("Try to execute this code")
except:
    print("This block will execute if there is a problem")
else:
    print("This block will execute if there was no problem")
finally:
    print("This block will always execute, at the end.")
```

Executed only when try successfully executed

Python Fundamentals

Catching Specific Exceptions

- The previous example catches all exceptions
- It is better to catch specific exceptions so that they can be dealt with appropriately
- E.g.:

```
def safeDivide(a,b):
    try:
        return a/b
    except ZeroDivisionError:
        return 1E100

safeDivide(1,0)
```

```
1e+100
```

- This will catch all Zero-Division errors, and let other errors pass through unmodified

Ulster University

Python Fundamentals

Catching Exceptions

- Multiple exceptions can be caught by using a tuple:

```
def safeDivide(a,b):
    try:
        return a/b
    except (ZeroDivisionError, TypeError):
        return 1E100
    except(AnotherError):
        return None
```

- A full list of built in exceptions can be found at:
  - https://docs.python.org/3.6/library/exceptions.html
    - ImportError
    - IndexError
    - KeyError
    - TypeError
    - ..and many, many more!

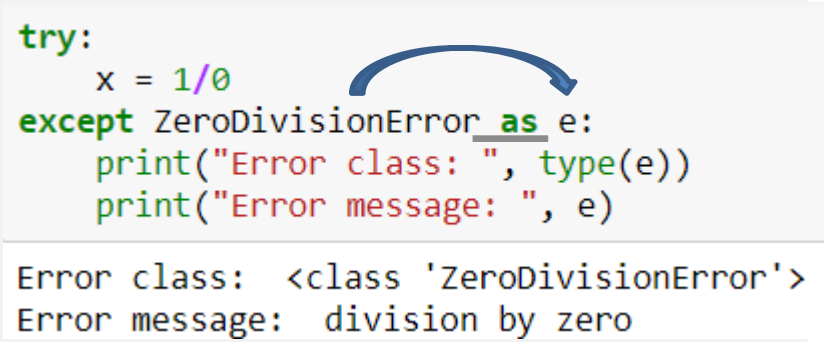
https://www.tutorialspoint.com/python/standard_exceptions.htm

Ulster University

Python Fundamentals

Exceptions in More Detail

- Accessing the Error Message
  - Use the '**as**' keyword:



```
try:
    x = 1/0
except ZeroDivisionError as e:
    print("Error class: ", type(e))
    print("Error message: ", e)
```

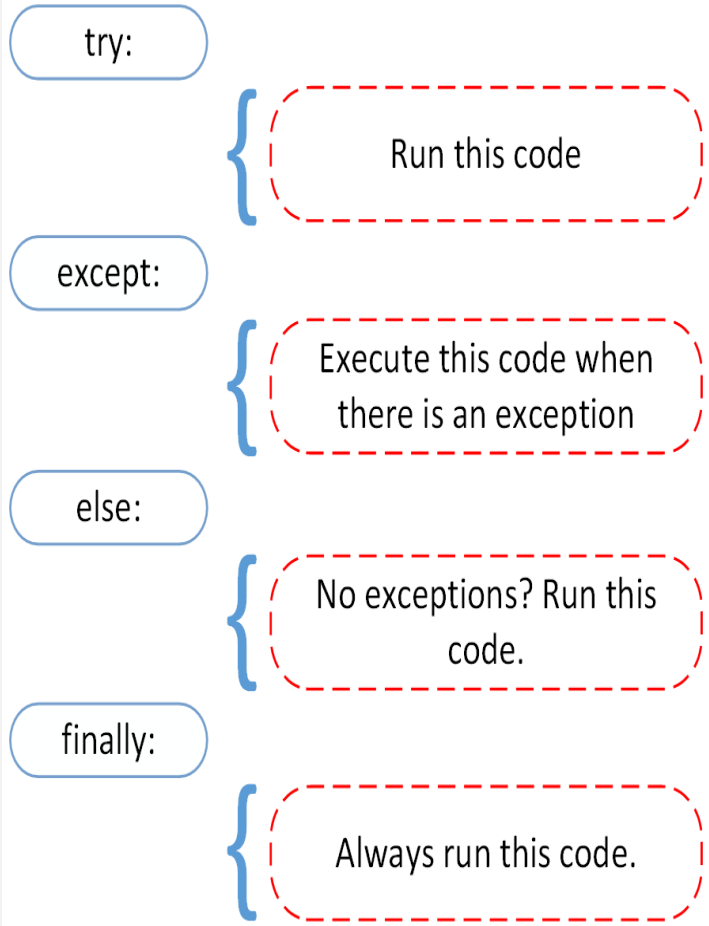
```
Error class:  <class 'ZeroDivisionError'>
Error message:  division by zero
```

Ulster University

Python Fundamentals

Summing Up

- **raise** allows you to throw an exception at any time
- **assert** enables you to verify if a certain condition is met and throw an exception if it isn't
- In the **try** clause, all statements are executed until an exception is encountered
- **except** is used to catch and handle the exception(s) that are encountered in the try clause
- **else** lets you code sections that should run only when no exceptions are encountered in the try clause
- **finally** enables you to execute sections of code that should always run, with or without any previously encountered exceptions



Ulster University

List Comprehensions

Python Fundamentals

List Comprehensions [ ]

- **Concise** way to create lists
- Common applications are:
  - ✓ To make new lists where each element is the result of some operations applied to each member of another sequence or iterable
  - ✓ To create a subsequence of those elements that satisfy a certain condition

Two expressions:

- function that needs to be applied
- sequence on which it needs to be applied

Example: **Non-concise approach**

```
1  squares = []
2  for x in range(10):
3      squares.append(x**2)
4
5  print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Concise approach

```
squares = ((lambda x: x**2, range(10)))
```

or, equivalently:

```
squares = [x**2 for x in range(10)]
```

→ **for loop in one line of code**

- **Directly assign the output to a variable**
- **List returned**

Ulster University

Python Fundamentals

List Comprehensions

- Non-succinct version:

```
L = []
for n in range(12):
    L.append(n**2)
L
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

- List Comprehension version:

```
[n ** 2 for n in range(12)]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

Read like: "**Construct a list consisting of the square of *n* for each *n* up to 11**"

- List Comprehension **Syntax**:
  - [*expression* for *variable* in *iterable*]

Ulster University

Python Fundamentals

List Comprehensions

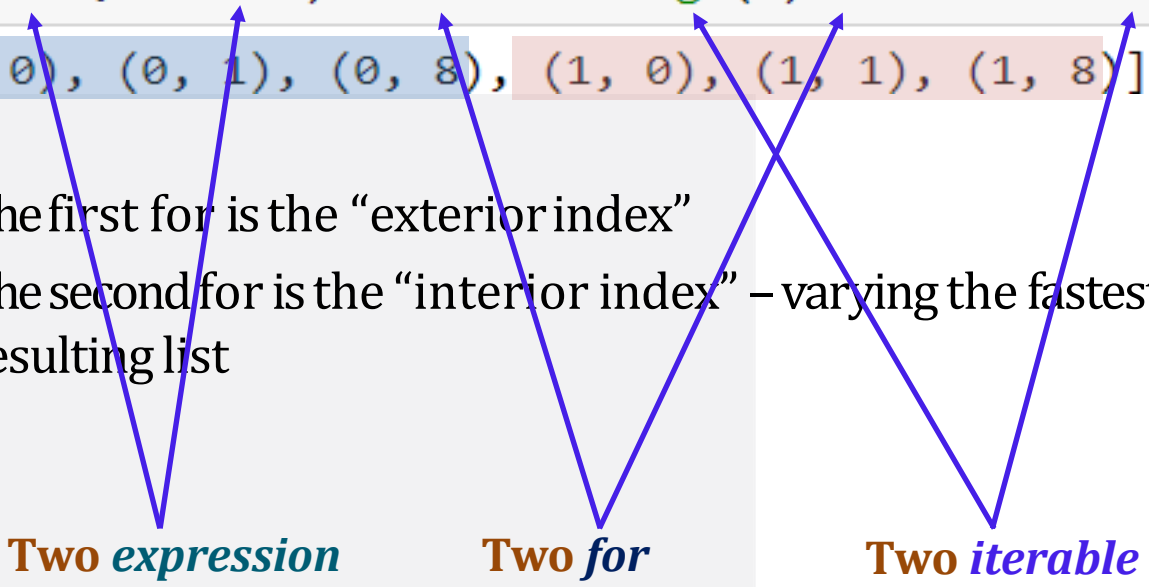
Multiple Iterations:

- To build up a list from more than one value
- Add an additional *for* expression, e.g.:

```
[(n ** 2, m ** 3) for n in range(2) for m in range(3)]
```

```
[(0, 0), (0, 1), (0, 8), (1, 0), (1, 1), (1, 8)]
```

- The first for is the "exterior index"
- The second for is the "interior index" – varying the fastest in the resulting list



Ulster University

Python Fundamentals

List Comprehensions

- Conditionals on the **Iterator**
  - Added to the end of the expression

```
[(n ** 2, m ** 3) for n in range(2) for m in range(3) if m % 2 == 0]
```

```
[(0, 0), (0, 8), (1, 0), (1, 8)]
```

- This example will only output values where *m* is an even number.
- Equivalent loop syntax is much more long winded:

```
L = []
for n in range(2):
    for m in range(3):
        if m % 2 == 0:
            n = n **2
            m = m **3
            vals = (n,m)
            L.append(vals)
L
```

```
[(0, 0), (0, 8), (1, 0), (1, 8)]
```

Ulster University

Python Fundamentals

List Comprehensions

Conditionals on the **Value**

- Syntax:
  - *a* **if** *condition* **else** *b*
  - *condition* is first evaluated, then either *a* or *b* is returned based on the Boolean value of *condition*
    1. If *condition* evaluates to **True** then *a* is returned
    2. If *condition* evaluates to **False** then *b* is returned
- e.g.

```
b = 10
[4 if b > 9 else 8]
```

```
[4]
```

- or:

```
regularCustomerThreshold = 0.7
currentCustomerVisitRate = 0.8

customerType = "regularCustomer" if currentCustomerVisitRate >= regularCustomerThreshold else "infrequentCustomer"

customerType
```

```
'regularCustomer'
```

Python Fundamentals

List Comprehensions

Conditionals on the **Value**

if val%2 else –val→ Do something

if val%3→ Do nothing

- More complex example:

```
[val if val % 2 else -val
 for val in range(20) if val % 3]
```

```
[1, -2, -4, 5, 7, -8, -10, 11, 13, -14, -16, 17, 19]
```

- Constructing a list which leaves out multiples of 3 and negate all values in the list that are multiples of 2

Ulster University

Python Fundamentals

List Comprehensions

Conditionals on the **Value**

- More complex example:

```
[val if val % 2 else -val
 for val in range(20) if val % 3]
```

```
[1, -2, -4, 5, 7, -8, -10, 11, 13, -14, -16, 17, 19]
```

- **Step 1: Use an iterator returning values from 0 to 19**

Ulster University

Python Fundamentals

List Comprehensions

Conditionals on the **Value**

- More complex example:

```
[val if val % 2 else -val
 for val in range(20) if val % 3]
```

```
[1, -2, -4, 5, 7, -8, -10, 11, 13, -14, -16, 17, 19]
```

- Step 1: Use an iterator returning values from 0 to 19
- **Step 2: Iterator conditional – only create a list of values that are not multiples of 3 (i.e. val % 3 returns a value other than 0)**

Ulster University

Python Fundamentals

List Comprehensions

Conditionals on the **Value**

- More complex example:

```
[val if val % 2 else -val
 for val in range(20) if val % 3]
```

```
[1, -2, -4, 5, 7, -8, -10, 11, 13, -14, -16, 17, 19]
```

- Step 1: Use an iterator returning values from 0 to 19
- Step 2: Iterator conditional – only create a list of values that are not multiples of 3 (i.e. val %3 returns a value other than 0)
- **Step 3: Value conditional – In values returned from step 2, return the current value if it is not a multiple of 2 (i.e. val % 2 returns a value other than 0)**

Ulster University

Python Fundamentals

List Comprehensions

Conditionals on the **Value**

- More complex example:

```
[val if val % 2 else -val
 for val in range(20) if val % 3]
```

```
[1, -2, -4, 5, 7, -8, -10, 11, 13, -14, -16, 17, 19]
```

- Step 1: Use an iterator returning values from 0 to 19
- Step 2: Iterator conditional – only create a list of values that are not multiples of 3 (i.e. val %3 returns a value other than 0)
- Step 3: Value conditional – In values returned from step 2, return the current value if it is not a multiple of 2 (i.e. val % 2 returns a value other than 0)
- **Step 4: Value conditional continued – otherwise, return every other value as a negative**

Ulster University

Python Fundamentals

Other Comprehensions

- **Set Comprehensions**
  - Use curly brackets instead
  - Remember – sets will remove all duplicates

```
{a % 8 for a in range(1000)}
```

```
{0, 1, 2, 3, 4, 5, 6, 7}
```

- **Dict Comprehensions**
  - Return 2 values separated by a colon (:)

```
{a:a**2 for a in range(5)}
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

- **Generator Expression**

```
(a**2 for a in range(5))
```

```
<generator object <genexpr> at 0x000001A5BFAD2150>
```

Ulster University