INFO1110 Notes

Week 1 – Introduction to Programming

- Program: set of instructions that were written in such a way that can be converted to instructions that a computer can understand and then execute.
- Programming in a text file must follow the syntax of the programming language → a compiler processes the text into a primitive language and turns it into an executable program the computer/interpreter can run.

What we have for all of these programs is something like the following:

```
<load things we need>    # "optional"
<begin a method>         # explicit or implicit, "main"
<print the string>       # definitely needed
<end the printed line somehow>
<end the method>         # as in line #2
```

- Or more clearly:

```
import sys

sys.stdout.write("Hello World")
sys.stdout.write('\n')
sys.stdout.flush()
```

Week 2 – Variables and Expressions

- Everything in a computer system is stored in binary digits, known as bits → True = 1, False = 0 (base 2).
- 8 bits is a byte, 256 combinations of bits in one byte → the total value of a byte is found with this general formula:

Bits

1s and 0s are easy to store as "on" and "off" so binary has become the universal way to store all electronic information.

If the bits are

$$d_7 d_6 d_5 d_4 d_3 d_2 d_1 d_0$$

then the total value is

$$d_7 2^7 + d_6 2^6 + d_5 2^5 + d_4 2^4 + d_3 2^3 + d_2 2^2 + d_1 2^1 + d_0 2^0$$

which is equal to

$$128d_7 + 64d_6 + 32d_5 + 16d_4 + 8d_3 + 4d_2 + 2d_1 + d_0$$

E.g.,

$$01101010 = 0 + 64 + 32 + 0 + 8 + 0 + 2 + 0 = 106$$

Hexadecimal

- 2 bytes would go from 0 to 2^16 – 1.

Hexadecimal

Sometimes it's handy to have a more compact way to store numbers in a range that's a power of two.
Noting that 2^4 is 16, we use the first 6 letters of the alphabet to represent 10, 11, . . . , 15.

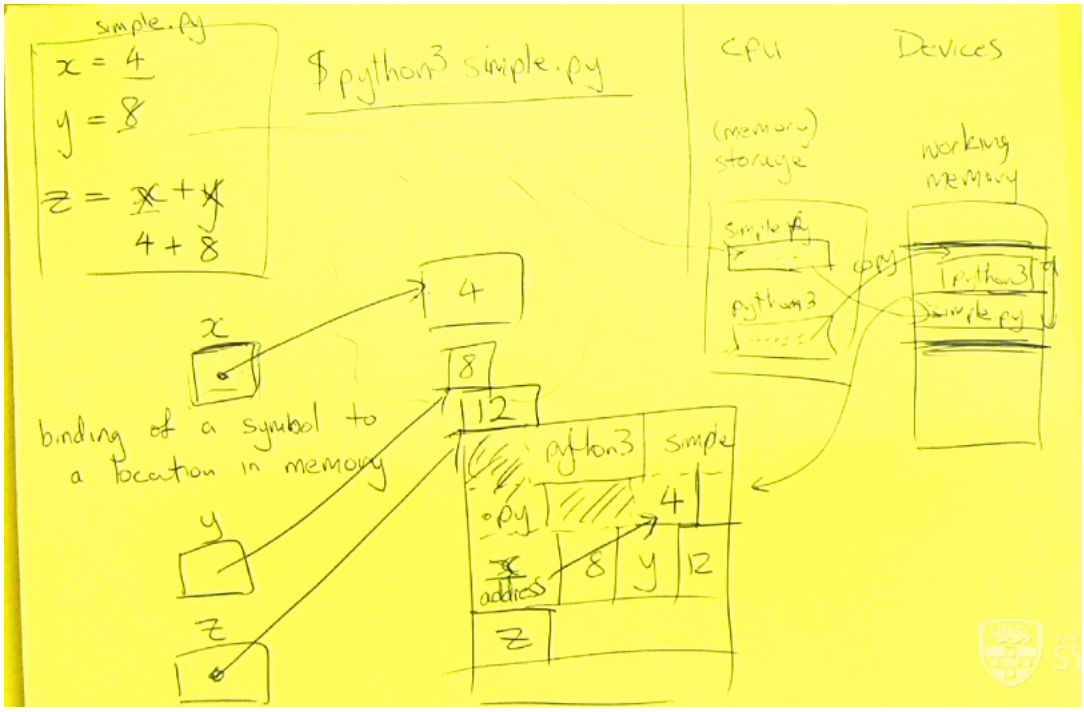
This is more compact: a byte just takes 2 characters: 00 (0) to FF (255).

To indicate that a number is to be interpreted in this way we put 0x in front: 0x00 and 0xFF.
Two-byte numbers need four such characters, so look like this: 0x0000 to 0xFFFF (65535).

binary	hexadecimal	decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Variables

- Variables are pieces of information stored in memory so we can access it by giving its name later.
- Memory (as well as the CPU, devices etc.) is found in the computer → it is here that programs are stored such as python programs and the python3 program itself.
- When a python program is run, both the program and python3 is copied across to the working memory, where it can be used.
- <u>In terms of variables, symbols are bound to some location in memory by creating a memory address to that location (the expression is created first, then the symbol is bound).</u>

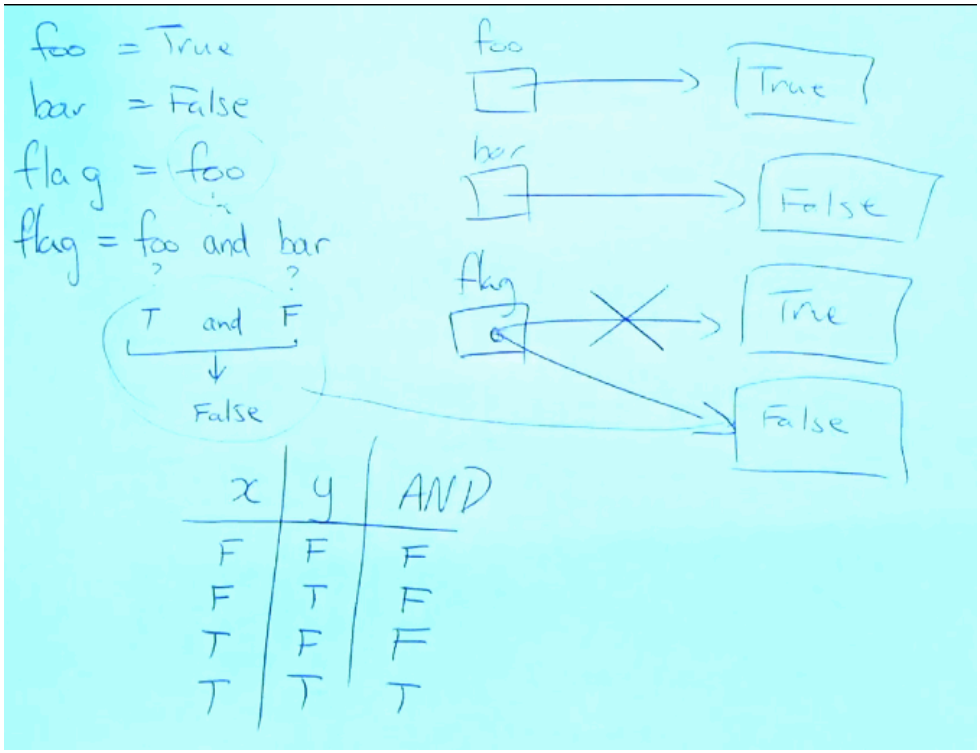


- Primitive types of variables use a fixed amount of memory e.g. an integer.

- Objects are blocks of memory that contain related data and associated operations for that data that use a variable amount of memory.
- How to use a variable → declare to the compiler what variables exist (and what data type it is), initialise what the variable contains, then use the variable as you wish.
- Within python, the first step is done for you (which can be an issue sometimes).
- Note: variables have a lifetime which it can be used for.

Using Boolean Operations

- Boolean operations are True and False.



- Other Boolean operations include OR, NOT, XOR (exclusive or)

Other Types of Object

- Strings (collection of characters, see below).
- Integers (exact number, no rounding)
- Floats (decimal numbers) → greater accuracy requires more decimals since it uses base 2 (constrained by memory, hence is often an approximation.
- Characters → human defined symbol (a graphic) that is used to represent a number e.g. A = 65, a = 97.

Accuracy requires integers since they are exact, however, when range is needed, floating point types should be used.

Expressions, Command Line Arguments, Conventions and Pseudocode

- Different computer languages have different main functions which is the entry point to the program → for Python, the code begins from the beginning of the .py file.
- Within the command line, the sys.argv list is read as STRINGS → starting from index 0 where 0 is often the .py name.

Operators

- In general x ()= y, it is equivalent to x = x () y e.g. x += y is equal to x = x + y.
- Left value == right value would give you a Boolean expression.
- msg1 is msg2 would not compare the contents of each variable but the memory address → DO NOT USE FOR STRINGS, since string variables are a memory address.
- ! or not, is how to negate e.g. not (x == y), or (x != y).
- 'And' operation is only true if all conditions are true, 'Or' is true if at least one condition is true.
- Most programming languages follow the BODMAS order → when programming, using brackets is essential.

<u>Comparing Floating Point Numbers</u>

- Comparing floating point number for equality is bad idea since they are stored to finite precision.
- One way to get around this is to subtract the two numbers from each other, take the absolute value, and compare it to a determine 'small' number.
- Hence abs(f1 – f2) < epsilon → when this is true, you can say that the two floating point numbers are the same.
- Another way to circumvent the issue is to convert to an integer e.g. $3.50 = 350c.

Week 3 – Control Flow 1: Branching and Looping

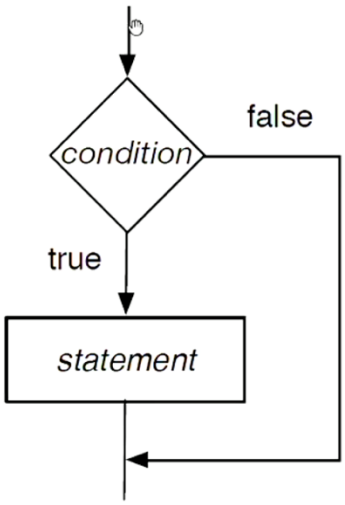
Boolean Expressions

Operation	*Meaning*
x and y	true if both x and y are true, false otherwise
x or y	true if both x or y or both are true, false otherwise
not x	true if x is false
x == y	true if x and y are both true or both false, false otherwise
x != y	true if x is true and y is false, or x is false, false otherwise

Casting Primitive Types

- Casting can change the data type e.g. from integer to a string (allows the programmer to explicitly tell the compiler to treat a variable, or expression, as another type).

If Statements



- Used to have different behaviours in a program based on whether something is true or false (kind of control statement).
- if condition :

  statement
- The indentation must be required when writing if statements.
- The condition is a Boolean expression that evaluates to True or False.
- The statement is one computer instruction or sequence of computer instructions within the same indentation block.

If…else

- The alternative to writing out both cases if to use an else statement, can be used as a temporary aside.
- if condition:

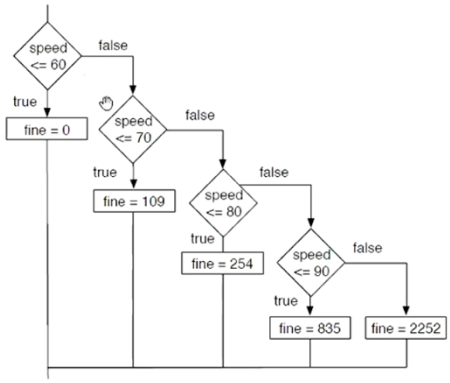
  statement
- else :

  alternative statement

if…elif…elif…else

- You can have separate conditions checked in sequence → if statements with the alternative statement elif (else if).

An example of checking conditions in a specific order




```
1  if speed <= 60:
2    fine = 0
3  elif speed <= 70:
4    fine = 109
5  elif speed <= 80:
6    fine = 254
7  elif speed <= 90:
8    fine = 835
9  else:
10   fine = 2252
```

Deskcheck

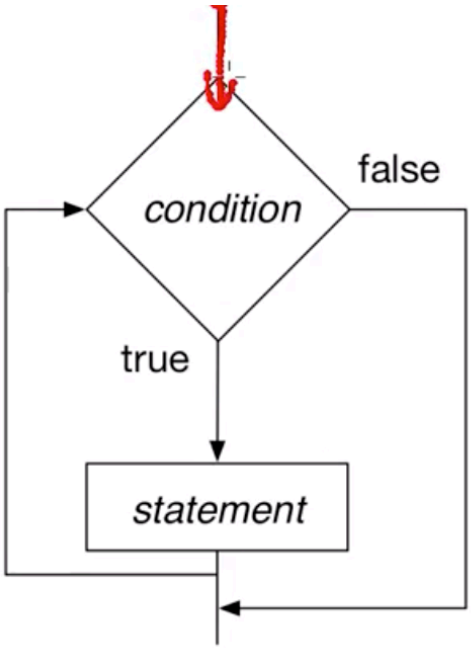
- Control statements like if can lead to different statements being executed, code paths.

- The variable will have different values depending on the current conditions.
- Desk checks follows your code through and is used to check whether your code has any logical/execution errors.
- Do deskchecks on normal, abnormal and boundaries.

While Loop

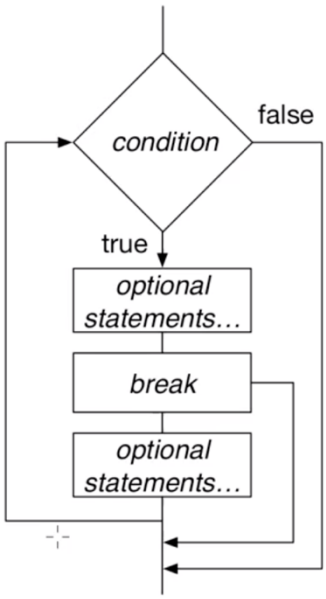
- While the condition is true, repeat what's in the statement.
- While (condition) do

  Statement
- Condition is checked, if true then go on to Step 2.
- Statement is executed, then go onto Step 1.
- The while loop continues until the condition is checked at Step 1 is false → the condition must be a variable.



1. begin with the *condition*
2. check the *condition*
3. if it's true, execute the *statement*; otherwise, exit the loop
4. go back to the *condition*

- Iterations = doing something again and again in a systematic way.
- Break is a special reserved word that breaks from the current iteration of the loop.



Read from File

- It is very useful to work with files to read in large amounts of data as input.
- UNIX environment is all about files and this will also be important for the upcoming assignment.
- The instructions will only be executed when the file was opened successfully/when they are completed, the filed will be closed.
- myfile is a variable.

Our sample.txt

```
This is my file, the first line
I have written this on the second line
last line!
```

```
with open('sample.txt', 'r') as myfile:
    print ( myfile.readline() )
    print ( myfile.readline() )
    print ( myfile.readline() )
```

Will print each line in the file

How do we read all lines in the file? Loops of course!

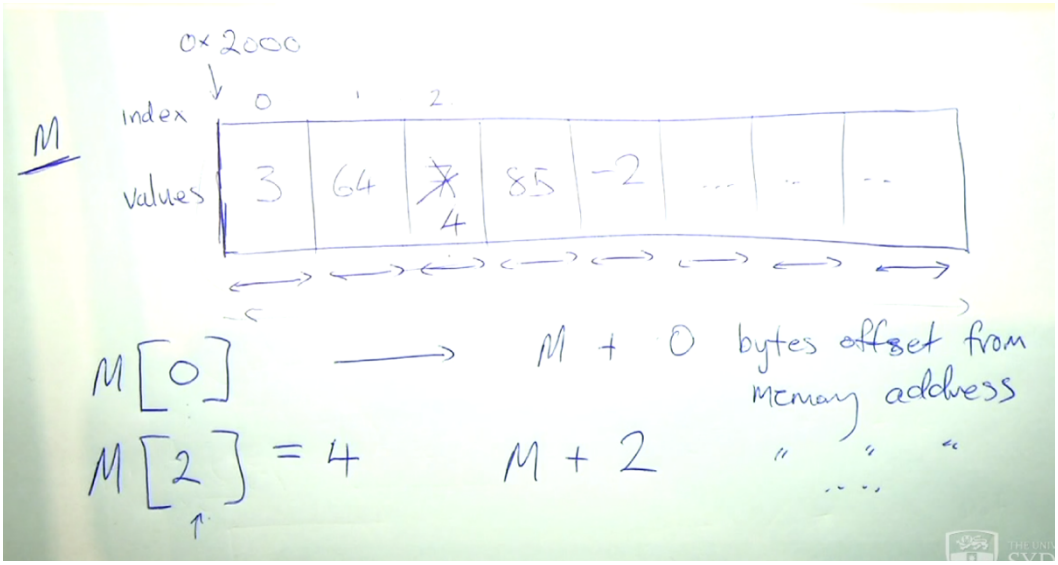
```
with open('sample.txt', 'r') as myfile:
    while True:
        line = myfile.readline()
        if line == None:
            break
        print(line, end='')
```

Week 4 – Arrays, Lists and Iteration

- print(a, b, c) → a b c

Arrays

- Arrays are a contiguous block of memory containing multiple values of the same type (all elements of the array must be the same width/bits in memory).
- An array solves the problem of having many variables e.g. you don't have to declare 100 int variables.
- A contiguous block of memory is one contained area of memory in which data can be stored within the data addresses.



Lists are not Arrays, Arrays are not Lists – Python

- Python presents the usage of a list as an array; however, lists have different operations and internal memory organisation.
- Arrays refer to the general data structure that applies to all programming languages whilst Python lists are built upon this concept.

Creating Arrays

- An empty array is created like this: [ ]
- A single element array: [47]
- Elements can be accessed by placing the element within square brackets (starts from 0).

Arrays — creation (cont.)

A single element array

```
1 x = [ 47 ]
```

Accessing the only element in the array

```
1 first_element = x[0]
```

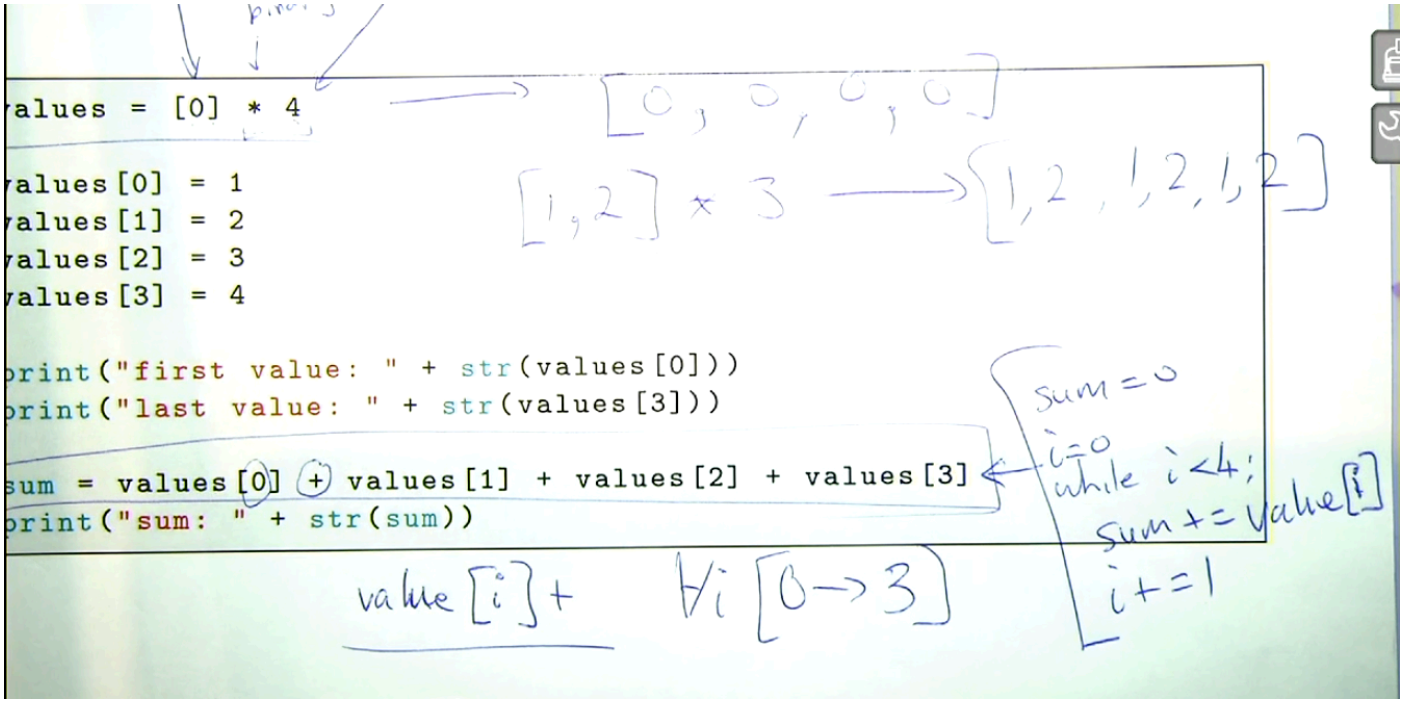
Printing the array

```
1 print("the array is:" + str(x))
2 print("the length of this array is: " + str(len(x)))
3 if len(x) > 0:
4    print("the first element of this array is: " + str(x[0]))
```

oops!

```
1 second_element = x[1]
```

- An empty array will have length 0, another appended on with add one to the length.
- A multi-element array is defined as comma separated objects of the same type within the square brackets.



- Values = [0] * 4 → the binary operator * will extend the array to a certain length.
- The len operator is useful when accessing the last index (note that last line).

len: Number of items in a Python object

There is a very nice feature of the data structures in Python that is built-in: for any list *x* that you define, the value `len( x )` is the length of the list.

That means if you declare an array of String items like this:

```
1 fruits = [ 'Apple', 'Banana', 'Cherry' ]
```

the length of `fruits` is 3.

Accessing the last element

```
1 fruits = [ "Apple", "Banana", "Cherry" ]
2 print("first fruit is " + fruits[0] )
3 print("second fruit is " + fruits[1] )
4 print("last fruit is " + fruits[ len(fruits) - 1] )
```

- Command line arguments are just arrays of strings, with index 0 being the name of the program.
- When you attempt to run a program that assumes you have some String arguments but don't give it any, you will get an Exception called IndexError.

ArrayOfStrings.py

```
1  import sys
2  i = 0
3  while i < len(sys.argv):
4      argument = sys.argv[i]
5      print(argument)
6      i = i + 1
```

Running this gives the following behaviour:

```
~> python ArrayOfStrings.py one two
ArrayOfStrings.py
one
two
```

Arrays of Objects

You can have arrays of *anything*: we have used `int` type and `String` type, but consistently.

As the programmer, you are deciding what memory you need to have and then how it is processed.

- print an array of integers
- print an array of strings
- print a mixed array of integers and strings...oh wait...does this work?

Mixing different objects into the same array can become problematic. There are quick and dirty solutions to this, however, they make both testing and debugging much harder.

Use the same type of object in your arrays!

- Within some types of arrays, a fixed amount of memory is used to store the same object type → however within this, the object type can be directions to another object/piece of memory – in doing so, an array can hold many types of data type (each fixed block directs you to another part of memory that stores a string, integer, float etc.)

Lists

- Lists are data structures that represent a collection of items (lists are commonly used in Python, and they are not arrays).
- A multi-element array is defined as comma separate objects of the same type within the square brackets [].

The list data structure has different characteristics than an array

- The size of the list is not fixed, it is mutable
- Any type of object can be be included
- Elements can be inserted or removed in any position
- Lists can be concatenated together
- A subset of the list can be created from the list
- Very different memory structure

- Searching is the most fundamental tool of any programmer, since we want to find a particular value among a collection of items (if it exists, where it is, how many exist).
- A search requires reading each element of the collection until the search criteria has been satisfied.
- Within lists, the keyword "in" can be used to find elements within a list.

The `in` keyword can be used to test if an object can be found within a list.

boolean expression := Object in List

```
1  mylist = [ "sand", 45, 3.14, "teacake" ]
2
3  if "teacake" in mylist:
4      print("found teacake!")
5  else:
6      print("oh no. no teacake")
7
8  if "almond" in mylist:
9      print("found an almond!")
10 else:
11     print("oh no. no almond")
```

In our case for a list, it will search for the equivalent value OR the object reference.

- In can only be used to find an object in list, not the other wat around.
- In can only find within one collection, it cannot look for objects within another collection. E.g. if ticket 1 has number 7, and lottery contains lottery 1, 2, 3 etc, you cannot use in to find if 7 is contained within lottery.

List operation – insert

Insert new elements at a given position.

list.insert(index, object) where 0 <= index <= len(list)

Insert at the end of the list

```
1 countries = [ "Malaysia", "Mozambique", "Benin", "Ukraine" ]
2 countries.insert(len(countries), "South Sudan")
3 print(countries)
```

Insert as the very first element of the list

```
1 letters = [ "b", "c", "d" ]
2 letters.insert(0, "a")
3 print(letters)
```

- When inserting in the middle, make sure you are careful about the index that you do place it into.
- Inserting at the end can also be done with list.append.

Searches the list and removes the *first* object matching the value

list.remove(object)

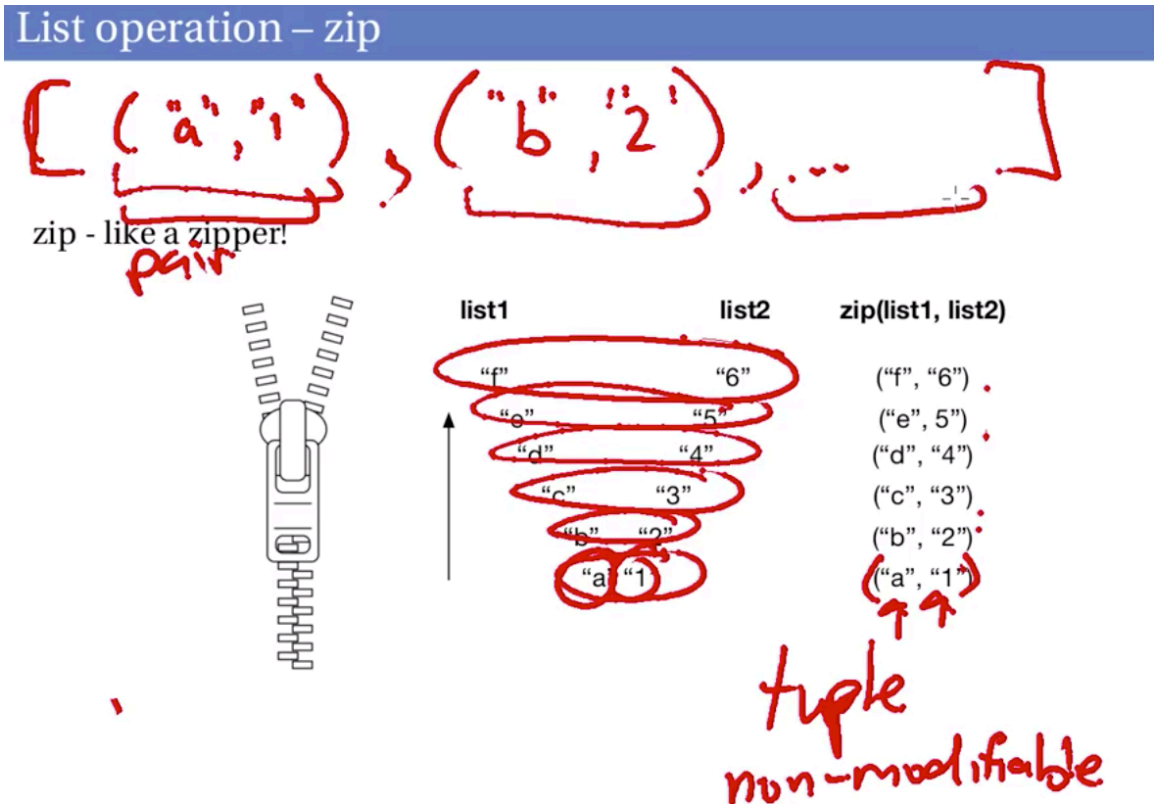
Only one! and it is the first one

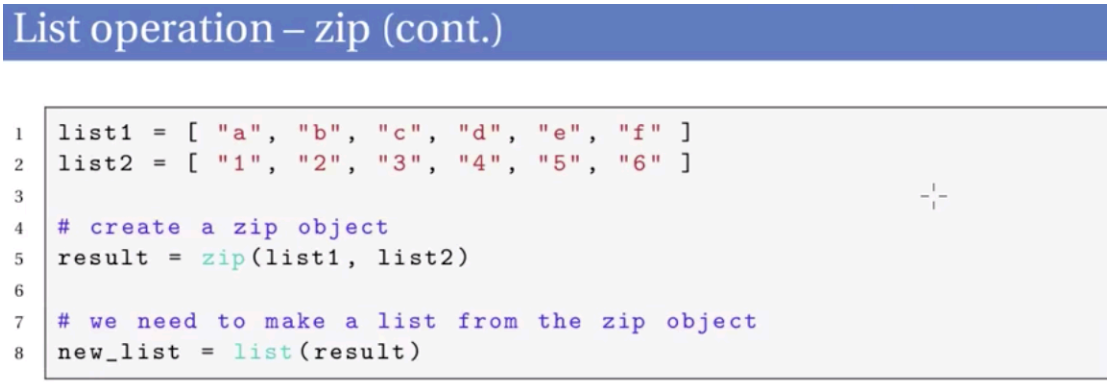
```
1 numbers = [ 7, 10, 01, -7 ]
2 numbers.remove(7)
3 print(numbers)
```

- When looking for the first object that matches the VALUE, Python will only remove the value that matches the same data type (if “700” is your first element, removing 700 will not take “700” away – HOWEVER removing 700 will also remove floats like 700.00).
- Lists can be concatenated together like strings. Adding an empty list will have no effect, as expected.

Zip and Tuples

- The zip operation pairs two lists in parallel, taking each respective element and placing them together in a TUPLE (new data type), which is a non-modifiable list of the same length.





How to describe zip?

- make a list of pairs
- merge two lists in sequence
- create a list that is the union of two lists, where the union operation is only for the first k elements of each list, where k is the smaller of the two list lengths

List operation – zip (cont.)

zip can help define an association between related objects

```
1  name_list = [ "Sunset", "Amber Jewel", \
2                "Teagan", "Queen Garnet" ]
3  weight_list = [ 0.872, 0.972, 0.797, 1.332 ]
4  unit_cost_list = [ 2.99, 6.99, 3.99, 17.99 ]
5  price_list = [2.61, 6.79, 3.18, 23.96]
6
7  # present invoice of minimum information
8  simple_bill = list(zip(name_list, price_list))
9  print(simple_bill)
10
11 # present invoice of all information with extra markup
12 itemised_bill_full = list( \
13     zip(name_list, weight_list, ["kg @ $"] *4, \
14         unit_cost_list , ["/kg = $"] *4, price_list))
15 print(itemised_bill_full)
16
17 # just one item
18 print(itemised_bill_full[1])
```

('Amber Jewel', 0.972, 'kg @ $', 6.99, '/kg = $', 2.61)

- The list operation (in blue) will form a new list of tuples.

Split

- The split operation will split the string into a list of smaller strings based on the delimiter sep.
- When splitting, the delimiter is NOT included e.g. the ‘on’ will be excluded.

List operation – split

A String operation. Split the string into a list of smaller strings based on the delimiter sep

str.split(sep=None, ...) where sep represents the delimiter found

We have seen this in week 1. There is a list returned from this function. Each element of the list is a string. We can choose how the string is broken up.

```
1  text = "The unit of measure, a zeptonewton, \
2  is equal to one billionth of a trillionth of a newton"
3  words = text.split(sep=' ')
4  words2 = text.split(sep='on')
5  words3 = text.split(sep='ll')
```

List operation – split (cont.)

More practical with structured text data

```
places="50°04′N   19°56′E; Kraków, Poland; 56°07′N   101°36′E; Bratsk, Russia; 4
6°49′N   71°13′W; Quebec City, Canada; 67°34′S   68°08′W; Rothera, United Kingdo
m; 22°17′N   114°10′E; Hong Kong,China; 24°34′N   81°47′W; Key West, United Stat
es; 43°50′N   87°36′E; Ürümqi,China; 7°12′S   39°20′W; Juazeiro do Norte, Brazil
; 14°33′N   121°02′E; Makati, Philippines"
```

```
1  # each place is *two* strings
2  place_list = places.split(';')
3
4  # search for Canada in the list. Use a loop!
5  canadian_place_coord = places_list[ ? ]
6  canadian_place_name = places_list[ ? ]
7
8  # split this up as coordinate, city, country
9  latitude = canadian_place_coord.split('   ')[0]
10 longitude = canadian_place_coord.split('   ')[1]
11
12 city = canadian_place_name.split(',')[0]
13 country = canadian_place_name.split(',')[1]
```

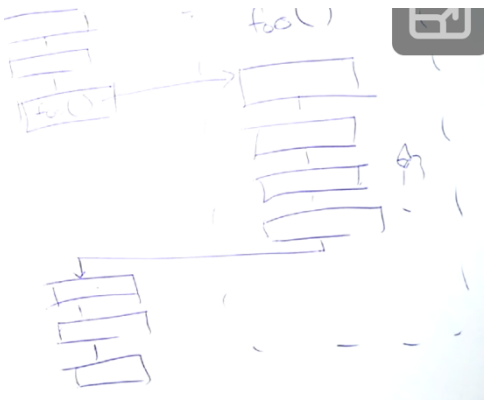
More on Lists and Loops – continue, for loop, range, list comprehension

- Continue is a special reserved word meaning “skip the rest of this iteration and go on to the next one”
- It is only allowed inside the body of a loop.

Week 5 – Function, Tuples

Functions

- A function is a separate part of a program that performs some operations, which can be invoked from somewhere else in the program.
- A function is a set of instruction that will produce an output based on a number of inputs.
- Functions can have many inputs and many outputs.
- An argument is something that is passed to a function, the parameter is something that is used by a function e.g. foo(x) where x is the parameter, res = foo(3), 3 is the argument.



def *name of function* (parameter inputs datatypes): → delimited/end of function prototype, referring to the colon.

Indented function body.

Why Use Functions?

Answer 1: It's tidy and easier to understand:

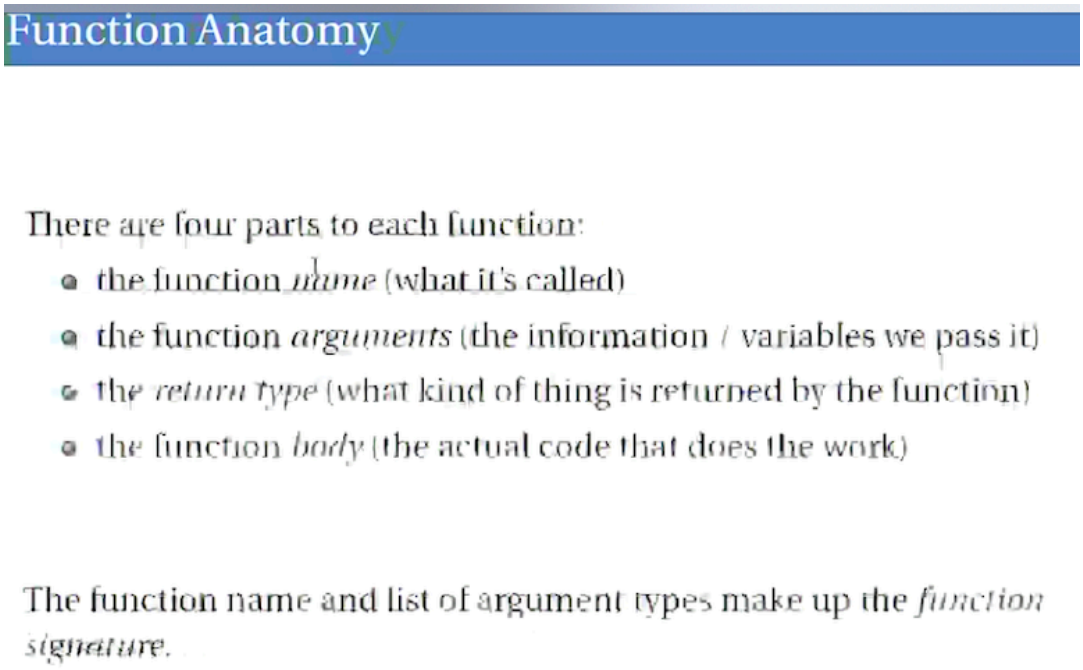
- Even slightly complex programs contain nested loops, conditions, and many variables.
- It's easier to check/test individual components, instead of searching through code for each instance of the same function.

Answer 2: it allows for code re-use:

- Once defined and abstracted, it can be used over and over again.
- E.g. if you have written a function for sine, you won't have to rewrite it every time.
- Calculate it again, search for it again, read from input again, displaying output again.

Answer 3: to reduce the chance of error:

- So you don't have to repeat the same code, functions are used to minimise the change of human error.
- If upgrades and updates are madem, functions allow for it to be changed immediately, instead of searching through and finding every instance.



Returning

- When you call a function you can give it information to process e.g. a calculation requires x.
- You also have the option of getting something back from that function e.g. a message to say it's a prime or the square root → you can return multiple items (using commas), tuples (x,y), lists [x, y] etc.
- If your code is intended to return something then you should NOT just print it out – this is horribly incorrect.
- Functions will automatically return None data type if no other return is specified.

Calling a function:

- Get together information I want the function to handle.
- Invoke the function by using its name and supplying the information as arguments.
- Do something with the returned item.

Function Local Variables

- Function arguments are new variables that exist only for the code block of the function → local variables.
- They exist ouside of the Main and are all removed when the function ends, except for the returned item.

Dictionaries

- Dictionaries store a collection of values. A value is added, updated, removed from the dictionary by searching for the value using a key.

Using built in functions

abs(x) Return the absolute value of a number. $|x|$

```
x = 5
y = -4
result = abs( x - y ) + y + abs(x)
```

pow(x, y) Return x to the power y x^y

```
x = 2
result1 = pow(x, 2)
result2 = pow( pow(x, 2) , 2 )
```

round(number[, ndigits]) Return number rounded to ndigits precision after the decimal point. If ndigits is omitted or is None, it returns the nearest integer to its input.

```
pi = 3.141592365358979
```

Using Python Standard library: random

`random.random()`
Return the next random floating point number in the range [0.0, 1.0)

```
# Returns a double value with a positive sign,
# greater than or equal to 0.0 and less than 1.0.
import random
i = 0
while i < 100:
    value = random.random()
    print("random value = " + value)
    if value < 0.1:
        print("winner ")
    i = i + 1
```

Week 6 – Exceptions

- An exception is an object that signals the occurrence of an unusual event during the execution of a program → an error is thrown, and can be caught by something else (e.g. another part of the code like a try block).

Using exceptions

Exception mechanism:

When an (exceptional) error is encountered

1. *Construct* an exception Object
   (which contains information about the error);
2. *Throw* (also called "raise") the exception;
   This stops normal flow of control.
3. The exception may be *caught* somewhere and dealt with; otherwise the execution stops.

Exceptions syntax

`try` a piece of code that contains a potential rare error

if the rare error occurs a new exception is generated. `raise`

The exception is caught using `except` and handled

Following the `try`, error or not, more code can be executed with a block `finally`.

- Exception is a class in Python, where you need to make an Exception object that can be passed around.
- Exception itself however, is very general and doesn't indicate what the specific issue is.
- Specificity in describing the error is used to allow another person to handle the problem and know where to look.

When to Raise an Exception

- An error or a situation occurs that makes the normal flow of processing unsuitable.
- Someone else should handle the error (passed up the chain, Python will crash the program if the error is not dealt with).

Catching and Dealing with Exceptions

- When the code catches an exception, the error can be dealt with and excecution continues OR the program cleans up the damage and exits.

Exceptions — structure

Exception-handling code

- always begins with a `try` block, which should surround all the code you think should normally work – but which could raise an exception;
- must be followed by one or more `except` blocks *in increasing order of generality*: that is, from the most specific to the most general;
- may be followed by a `finally` clause, which contains code that will always be executed if it's there;
- should not be used for control flow!

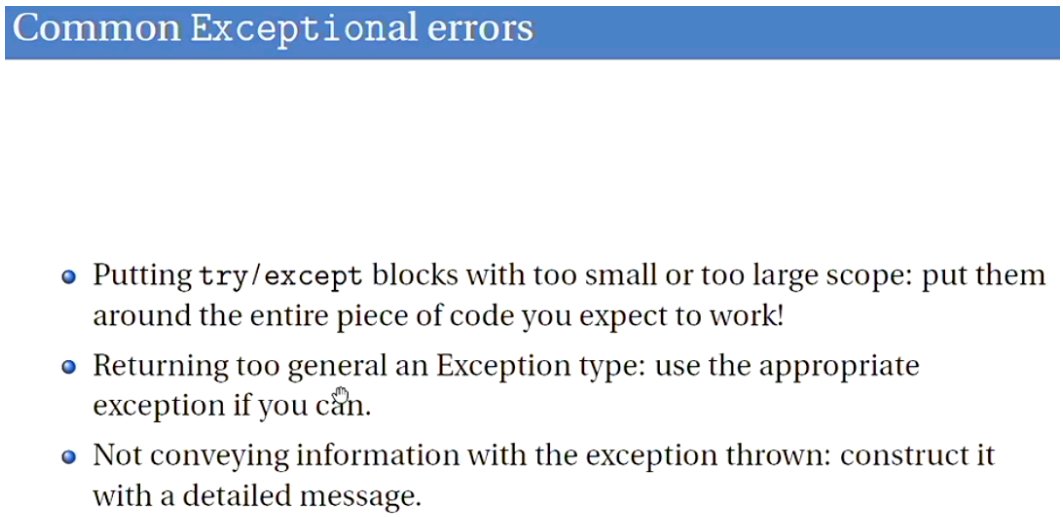
Some exception code

Separate the "normal case" code (in the `try` block) from the code for weird situations (the "`except`" block(s)).
You may have several *different* except clauses, depending on kind of exception thrown in the `try` block.
The optional `finally` block is always executed at the end of the `except` region:

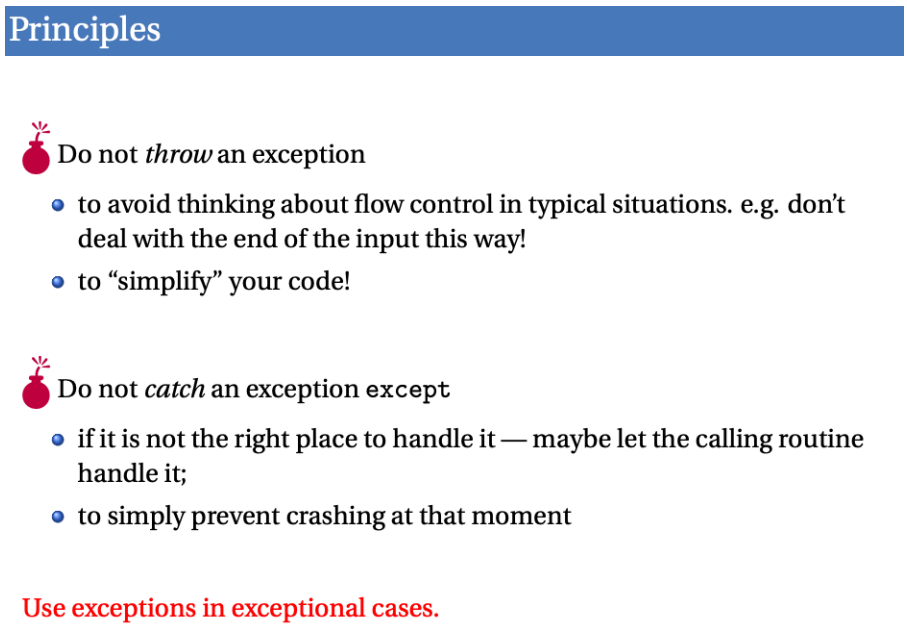
- after the `try` block completes successfully,
- or after a `except` block is executed.

```
try:
    # do something
    # which could throw
    # Exceptions
except EOFError:
    # do something to deal
    # with the situation
except FileNotFoundError:
    # do something else
except Exception:
    # do something general
finally:
    # code that should happen
    # after normal/weird case
```

- If an exception occurs in a method constructed and thrown explicitly, or raised by some other method that is called, and if it cannot be handled then that method will terminate and throw the same exception to the caller.
- The exception will be passed up until someone can handle it.



Common Exception Use Errors



- In terms of flow control, if you can use if…elif, that is the best way to deal with an issue.
- If you can handle an exception, do not use an exception.

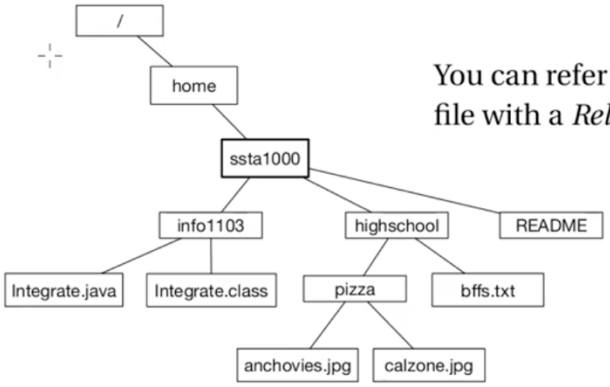
Files

- Files are an idea that makes information storage simple for users.
- E.g. hello.py → .py is an extension/suffix that gives an idea of what the file contains for human users.
- There are no rules about what information is stored in a file, and the suffix has no meaning to the computer.
- The linus tool *file* can scan the content of a file and determine its type.

Where are Files Stored?

- Files are stored in directory structures for easy human usage so that we can see where files exist and the path to it.

There is a *path* associated with files



You can refer to the
file with a *Relative* path, or *Absolute* path

- Absolute paths begin fom the root directory, whilst relative paths begin from another file/directory e.g. home.

Dealing with Files

To read from or write to from a file you need several things

1. The file has to be there
2. The file has to be available — it must be opened
3. You must have access to it
4. You must know what to read/write

Once you've finished with a file you should *always* close it.

- We need to first create a File Object: an abstract representation of a file e.g. file_variable = open("README", "r") → gives a handle over the file.

`readline()`: A method to read exactly one line of text that is delimited by a new line character (default). A String object is returned. Where the string object is empty, there is no more data in the file

```
1 file_variable = open("README", "r")
2 one_line = file_variable.readline()
3 print("first line of file is : " + one_line)
```

- However, do not forget error handling e.g. FileNotFoundError, IndexError

Array bounds checking

Switch the flag to open the file in write mode

```
1 the_file = open(sys.argv[1], 'w')
```

Expecting file to exist

Expecting data to be there

`f.write(string)` writes the contents of string to the file, returning `None`. Don't forget the new line character!

```
1 the_file = open(sys.argv[1], 'w')
2 the_file.write("10")
3 the_file.close()
```

Always expect integer

Didn't close file

Potential data loss - If the file exists then it will be truncated to zero size; otherwise, a new file will be created.

Week 7 – Idioms

- A programming idiom is a commonly used method to solve a small problem, related to searching.
- Infinite looping until a goal is satisfied is best done with while True.
- Square brackets mean inclusive, round brackets do no include that index.

Reverse Looping

```
1 i = N - 1
2 while i >= 0:
3     print(i)
4     print(numbers[i])
5
6     i = i - 1
```

Creates sequence for values of `i` as `N-1, N-2, N-3, ... 3, 2, 1, 0`

Prints the values of `i` with first value `N-1` Prints the values in the array in reverse order, last element first, index $N-1$

N – i – 1 can be used to create a reverse index (e.g. for bouncy strings).

Same. print values at indices `2, 3, 4, 5, 6`

Version 3 - we know it is 5 times, make a variable for offset from beginning

```
1 start = 2
2 i = 0
3 while i < 5:
4     print(numbers[i + start])
5
6     i = i + 1
```

This method can be used in sequence control flow so that we know how many iterations there are. Another method is using start = 2, end = 7, I = start, while I < end.

Obligatory python slice :

Used in indices of array operators, return a new collection that is a subsequence (not necessarily smaller)

```
numbers = [ 0, 1, 2, 3, 4, 5, 6 ]
subnum = numbers[1:3]
print(subnum)
```

```
[1, 2]
```

Read as `array[start:end]`, where the indices are [*start*, *end*)

Convention of reading sequences as `[0,N-1]` reigns supreme!

Other tricks with slicing

Copy sequence all values starting from index *start* up to and including $N-1$

```
numbers = [ 0, 1, 2, 3, 4, 5, 6 ]
subnum = numbers[3:] # [3, N )
```

Copy sequence all values starting from index 0 up to and including *end* – 1

```
numbers = [ 0, 1, 2, 3, 4, 5, 6 ]
subnum = numbers[:3] # [ 0, 3 )
```

Copy entire sequence starting from index 0 up to and including $N-1$

```
numbers = [ 0, 1, 2, 3, 4, 5, 6 ]
subnum = numbers[:] # [ 0, N )
```

SLICING CANNOT BE USED IN ASSESSMENTS, AS THIS DOES NOT APPLY TO ALL PROGRAMMING LANGUAGES.

Idioms with arrays

Programs tend to work with same data but different size. e.g. sum of N numbers, where N could be small, or really big

The data is stored in some kind of memory, could be on disk as a file, or could be in main memory

To solve the problem, we have to search. We need to **visit** each number that is somewhere in memory/file

An array is the most typical way to represent N pieces of data, where each piece is the same data type (e.g. Integer)

The idiom is to scan the entire array and *do something* with each value

Search for Minimum of N Numbers

It would make sense to start defining what happens where there is no minimum. e.g. an empty list.

We have some design decisions here:

- return None
- return the largest possible value
- throw an exception
- change the function prototype and return an error code

```
def find_minimum(numbers):
    if numbers is None: # or (not (numbers is <expected type>))
        # do special thing
        # return ...
        # raise ...
    N = len(numbers)
    if N == 0:
        # do special thing
        # return ...
        # raise ...

    min = sys.float_info.max # largest possible number value
    i = 0
    while i < N:
        if min > numbers[i]:
            min = numbers[i]

        i += 1

    return min # always correct, even if largest
```

- Largest possible value may not always be the best, returning None may be more effective.

Idiom: general idea with arrays

- A search task
- input is some array
- there is some criteria for the search
- there is a failed search outcome that needs to be considered

```
input: array[N]
output: result

LOOP over each element of the array
    if search criteria:
        update result

if search failed criteria
    error handling/reporting

return result
```

Using Objects and Methods

Modules

- A single file which stores the related definition, functions and code that can be used to aid organisation of code (Python also has specific mechanisms associated with the modules).
- Modularity allows you to break a problem down into smaller, more logical and easily dealt with parts.

Break up code into *developer defined categories*

e.g.
User interface subsystem
Stock tracking subsystem
Item scanning subsystem
Cash Payment subsystem
Electronic Payment subsystem
Discount subsystem

- A module is a .py file containing source code.

Using a Module

- *import* is a keyword that will open another file and insert all the code there.
- All code is executed once when import is used for a module. That code is executed as if it were in the correct place.
- HOWEVER, Python keeps the module contents out of current scope – all symbols visible to the current execution environment.
- The code imported in stored in an Object, that has related data and functions → functions and variables from an object can be accessed with a . (dot) notation.

ModuleTest2.py

```
1 import NumberOperations
2
3 x = 2
4 print( NumberOperations.get_square(x) )
5 print( NumberOperations.get_min(29, 5) )
```

- For clarity, objects should be assigned to another variable name.
- Multiple imports can become tricky, since we have to observe the hierarchy of import.

For ModuleA to use a function in ModuleC, we have two ways of going about this

ModuleA.py

```
1 # go straight to the source
2 from ModuleB import ModuleC
3 ModuleC.callme()
4
5 # use a series of . operators
6 import ModuleB
7 ModuleB.banana()
8 ModuleB.ModuleC.callme()
```

Objects

Strings Methods

`str.find(sub[, start[, end]])` Return the lowest index in the string where substring sub is found within the slice `s[start:end]`. Optional arguments start and end are interpreted as in slice notation. Return -1 if sub is not found.

```
msg = "hello world"
print(  msg.find('o') )
```

`str.replace(old, new))`

Return a copy of the string with all occurrences of substring old replaced by new.

```
msg = "hello world"
print(  msg.replace('o', 'a') )
```

`str.strip()`

Return a copy of the string with the leading and trailing characters removed.

```
msg = "   hello world   "
print( msg.strip() )
print( msg )
```

`str.title()`

Return a titlecased version of the string where words start with an uppercase character and the remaining characters are lowercase.

```
String msg = "   hello world   "
String msg_title = msg.title()
print( msg )
print( msg_title )
```

- Multiple methods can be called on strings, although note that order does matter.

Functions vs Methods

The difference! what is it!

Function	Method
`find_value(msg, 'a')`	`msg.find('a')`
`replace_c1_with_c2(msg, 'a', 'b')`	`msg.replace('a', 'b')`

A function has no notion of existing data. It operates on inputs and produces outputs

Methods are functions operating on objects. The calling of a method will use information within the object to calculate, produce, a result

python uses functions and presents the reserved keyword `self` to refer to functions and data within the object for use as part of the function (more on this later when discussing classes).

Week 8/9 – Testing

- A test gets some input to a program and then investigates the output (what is the input, what is the expected outcome, what is the actual outcome).
- To test, we need to inspect the contents of the memory, and see if the values of what we compute matches with what we expect.
- An oracle is some external 'thing' that we can ask for the right answer e.g. customer who knows what they want → we don't have an oracle, hence we must make our own tests and choose the precision etc needed.
- Code coverage in testing is the proportion of lines of code that you have tested/covered/used → consider every execution path (route of the flow of the program).

Tracing Code

- Tracing through the code itself by writing out the state of variables as you go through loops etc is an effective way to test.

However, it is not scalable:

- Although the chance of identifying a problem is very high, the chance of making a mistake in code tracing increases if not careful.
- Extremely time consuming in large amouts of code.

Test-Driven Development

- This is a way of software development that requires you to write the tests for a given function first, before you even write the function!
- This has a number of advantages: it makes you think exactly what you want your functions to do in terms of:
    - What the normal functionality should be
    - What should happen given 'bad input' (e.g. passing a negative number into something that only accepts positive numbers.)

Failing is a good thing

- The test has brought your attention to a specific case where things should work.
- The time spend on making the test, for a small part of a program, is much less than bug hunting an entire program.

Types of Testing

- Black Box Testing: treating the program as a kind of a 'black box' which simply processes input and output, with no reference to how it works.
- White Box Testing: looks insde the function in detail, enabling you to test different and extreme kinds of input and different execution paths.
- Testing by printing:

This is a very common function and also has its (limited) use:

Print out a running log of what's going on, while you're developing code, and print out error messages if things go wrong.

This is fine up to a point but *don't do it all the time!*

After finding the problems, you have to go back and comment out all those print statements, essentially all your checking/testing is never used again.

It's *far*, ***far*** better to have your testing separate from the rest of the code!

- Regression Testing: running ALL your tests again whenever you make a significant change to your code, to make sure you haven't broken code that used to work.

Corners and Edges

- An edge case is where one dimesion of a problem is at an extreme e.g. when the load on a bridge is maximum.
- A corner case is where more than one dimension is at an extreme, such as when a bridge has maximum load and it's being hit with highest possible winds and it's an earthquake.

Assertion

- An assertion is a condition (Boolean expression) written at some point in a program.
- We assert that the condition must be true whenever the program reaches that point.

- Assertions are designed to stop execution.

Assume there was mismatch of expectations

```
1
2  def get_percentage( value, largest_value ):
3      ''' guarantees to return a float type that is within the
           range of [0,1]
4          to represent 0% to 100%'''
5
6      return value / largest_value
7
8
9  percent = get_percentage( 13, 71 )
10
11 assert percent >= 0 and percent <= 1
```

Assertions in Python are handled as Exceptions. You can catch them, and then decide what you could possibly do in that situation of unexpected values. (it may not make sense to use assertion)

- Pre and postconditions are assertions placed before and after some piece of code that are considered by a programmer whilst coding e.g. for a square root function, the precondition is that the number is greater than 0, the post condition is that the result is greater than 0.
- The code will always produce a true postcondition if the precondition is true.

Software Design Process

- The software design process focusses largely on gathering requirements, and then coding for that.
- When given a description, attempt to determine capabilities/abilities/commands, conditions, starting conditions, variables needed, inputs, outputs.

What is the output?

What information needs to be stored?

What is being processed/updated?

What is the input?

- Following this, consider what can be tested e.g.
    - repeated input
    - what if the input is faulty (wrong type, range of values etc)
    - how is data treated (float, integer, limits, initial values)
    - is there an order to input and expressions
    - what are the combinations of these?
    - What is the precedence of the conditions, will one have to be passed before other conditions can be tested?
    - What is the format of inputs and outputs, what are adjustments made (rounding?)

Week 9 Lecture – Classes and Objects

- Variables in programming are used all the time, but there are cases when we need types which:
  - → control access to the value
  - → control the operations possible with the value
  - → contain more than one kind of value (not an array, a composite of other types).
- An object is a thing that may have data or particular methods, or BOTH → constructed differently.
- The type of an object variable is it's class, objects are instances of things of a particular class → P = Point(1, 0), P is the variable, Point is the class type, Point(1, 0) is the object.

Constructing a Class

class NameofDataType:

definition of data type

….

….

Constructors

- The constructor does the work of building an instance of the class.
- It creates space in memory for the object when it is called, so that a programmer can then initialise/set the contents of the objects memory.

- A constructor for a class is a plan, or blueprint, to make an instance of the class.
- When you make classes, new data types, you often need to make instances of them.
- *Self* used to access the instance variable of the class.

```
1  class PositiveInteger:
2
3      # constructor method for class
4      def __init__(self):
5          # things to do when creating
6          # a new instance of class PositiveInteger
7          self.number = 0
```

`self` is a keyword to describe the instance of the class (object). [1]

There is always a default constructor if none was defined

We could have a very simple constructor for our new `PositiveInteger` class, like this:

```
class PositiveInteger:
    def __init__(self):
        # this constructor does nothing!
        pass
```

or this:

```
class PositiveInteger:
    def __init__(self):
        self.number = 1
        # this constructor creates and sets a new instance variable
```

or this:

```
    def __init__(self, initial_value):
        self.number = initial_value
        # this constructor creates and sets a new instance variable
            to the parameter
```

When you do that,

- space is made for the object of type `PositiveInteger`
- the instance variable of the class is set to some value
- variables `water` and `kelvin` are ready for use.

What do you notice?

- no return type — it returns an object;
- `PositiveInteger()` is an expression. It will still create an object, even if it is not assigned.

```
PositiveInteger(35) # fine, but does nothing
print("some number is: " + str(PositiveInteger(35).number))
```

With an Object variable we use the dot . to refer to instance variables or methods e.g.

```
text = "beetle"
uppertext = text.upper()
```

We can do the same with instance variables, without parentheses

The instance variable value is visible from any part in the program with that object reference.

public instance variables are not helpful if we want to restrict read/write access

currently, any part of the program can modify the value without checking it is *valid*[2]

```
import PositiveInteger
num = new PositiveInteger(34)
num.number = -470 # <--- should not be allowed!
print( "num.number: " + str(num.number) )
```

By design, any object of type `PositiveInteger` should contain only positive integer values

- We need to create our own function calls/operations.

We can create many instances of the same class, but they each have their own memory

We can define a method of a class. A method is a function that associates with the memory of the object

```
class PositiveInteger:
    def __init__(self, initial_value):
        self.number = initial_value

    def add_value(self, value):
        self.number += value

x = PositiveInteger(10)
x.add_value(4)
y = PositiveInteger(10)
y.add_value(4)
print(x.number)
print(y.number)
```

```
# function
def calc_total(values):
    ...

# method
def calc_total(self, values):
    ...
```

The set method is one of the preferred ways the value can change from anywhere in the program

By forcing usage in this way, we can define restrictions on what possible values it can have

```
# allowable number: zero or greater
def set_number(self, newnumber):
    if newnumber >= 0:
        self.number = newnumber
```

Week 10 Lecture – Recursion, Further Classes, Methods, WorldPoint

Recursion

- A technique to solve a problem → an arbitrary sized problem can be solved by first solving a smaller version of that problem.
- Recursion has two essential ingredients:
    - → a base case (somewhere to stop)
    - → the recurrence relation (a way to continue)
- The base case is also known as the trivial case and is the case where we don't need to use the recurrence relation to get the answer.
- The recurrence relation tells you how to get from a more complex case to a single one.
- Without the recurrence, you can't proceed, without the base case, you can't stop.

Recursion with Matryoshka Doll

Familiar example

Define a class MatryoshkaDoll (Russian Doll)
A Matryoshka Doll contains zero or one Matryoshka Doll
A Matryoshka Doll has a size, 0 being the smallest

Task 1: For a given MatryoshkaDoll: count the number of dolls inside

Task 2: For a given MatryoshkaDoll: add up all the sizes of all dolls inside

Task 3: For a given MatryoshkaDoll: increase the size of the inner most doll by one, then increase the outer doll size by one (inner must be first!)

Task 4: For a given MatryoshkaDoll: increase the size of the inner most doll by `n`, then increase the outer doll size to at most `n + 1`

Something that allows you to define something in terms of itself, e.g.,

- *factorial*: $n! = n \cdot (n-1)!$ (a function)
- *Fibonacci sequence*: $F_n = F_{n-1} + F_{n-2}$ (a sequence of numbers)
- *list*: a list either *empty*, or is an *item* followed by a *list* (a definition)
- *rooted binary tree* (a definition)
    - T = *root* {+ left subtree TL} {+ right subtree TR}
    - where TL and TR are rooted binary trees

Writing a recursive function, you must:

- Know what the recurrence relation and base cases are
- Check whether the base case has been reached
- If it hasn't been reached, call the same function with different arguments

- recursive functions have a base case and a recurrence relation;
- recursion is never necessary: the alternative is iteration
- recursion can be very inefficient
- the recursion tree indicates approximately how much time is taken and how much space is required for a recursive function

Each node in the tree is a function call.

The number of nodes in the tree is the number of *function calls*, so the amount of time taken to work through the whole tree is proportional to *the number of nodes in the tree.*

The *height* of the tree is the maximum number of copies of the function in memory at any time, so this corresponds to the maximum memory requirement of the recursion.

Recursion can be very useful:

- code is usually clean
- easy to write/read/understand,
- some problems are much harder to solve in other ways.

But recursion is *not always ideal*:

Recursive code may be very inefficient (and this can be hard to spot).

Recursion is *never essential*: with enough effort, it can be replaced by *iteration.*

- When calling a function, we copy the value to be used in the function → the reference value is copied.
- When creating an Object, we have a variable that stores a reference, the value of memory address.

Reference value (cont.)

More bad news

We need to also know if the arguments to the function are modified in the first place. That is, if the data type is mutable, it can be modified. Otherwise we cannot know the outcome of a function call on that object.

```
1 s = "hello {}"
2 s.format(123)
```

returns a new string, it does not modify the existing string.

```
1 x = 1
2 x.__add__(1)
```

returns a new int, it does not modify the existing int

- When calling a method, the reference value is copied.

Understanding references and function calls (cont.)

We want to know the state of the program at any given time

We use the idea of a desk check to understand the changes from one instruction to the next

Problem: we don't know if after calling foo(a,b) what the state is.

To confirm what is the state of the program after a function call that uses objects we need more information:

1. We need to know if the data type mutable or not
2. If it is mutable, we need to know if function `foo()` calls methods on mutable object
3. If it is mutable, and that object has a method called upon it, we need if that method modifies the state of the object

class variables

Variables that belongs to the `class`.

Instance variables belong to *one* instance, whereas class variables are common to *all* instances.

What is common to all objects? Identifiers, global values, shared settings

```
class Student:
    # global counter for ID
    sid = 0

    def __init__(self, name):
        self.name = name
        self.id = Student.sid
        Student.sid += 1

students = [Student("Kerry"), Student("Xixi"), Student("Alice")]
for student in students:
    print(student.id)
```

static and class methods

static or class methods can always be called without any objects ever being created.

instance methods with `self` must be associated with the memory of an instance.

static or class methods can be used to operate on class variables, or they can perform operations related to the class similar to a function (input, process, output)

```
class WorldPoint:
    ...

    # Latitude/Longitude converted to read as xxx S/N  yyy W/E
    # returns a list of 2 strings. Each is positive num
    # 1st element has S/N, 2nd element W/E
    def get_human_readable(worldpoint):
        ...
    def get_human_readable(lattitude,longitude):
        ...
```

Review

class	the type of an object, e.g., the class `str`; to do with the type of an object, e.g. class variable or class method;
`object`	the most basic class in `Python`;
instance	to do with a single copy or case of, e.g. "piano" is an *instance* of the type "MusicInstrument"
method	a separate block of code that can be called, e.g., `s.format(...)` or `s.__len__()` for a string `s`;
class/static variable/method	in the current context, applying to the whole class, as variables, e.g. `sys.argv`), or as methods, e.g., `str.split("simple word", " ");`

Week 11 Lecture – Iterators and Generators

- An iterator (a way to move between different elements of data) is an object representing a stream of data
  → could be finite or infinite.
- The iterator deals with how to navigate data, and has two fundamental operations:
  → return the next element in the stream of data.
  → test if the end of the stream has been reached.
- Given a collection, traverse through each item: (the way that a collection is traversed is depending on circumstance).
  → in the order they are stored
  → in sorted ascending order
  → in order of addition
  → in order of importance
  → selectively
  → in random order

Existing Iterator: list

You have already been using iterators. List objects support the iterator functionality.

```
1 strings = ["Do", "we", "sell...", "French...", "fries?" ]
2 p = iter(strings)
3 print(type(p))
4 print(p.__next__())
5 print(p.__next__())
6 print(p.__next__())
7 print(p.__next__())
8 print(p.__next__())
9 print(p.__next__())
```

- Suppose we have a collection of items that we wish to traverse in a certain order.

  → the iterator class represents how the traversal is performed.

  → the iterator object represents the current state of the traversal (position of the pointer).

Make your own Iterator

Example. We want to create an iterator for a collection of people's names. The iterator would give us the same sequence of names as they are presented in the collection.

We first need to define this new data type for a collection of names

```
1 class People:
2     def __init__(self, names):
3         self.names = names
```

Iterator requirement 1: The data type has to support a method `__iter__()` to return an iterator object

```
1 class People:
2     def __init__(self, names):
3         self.names = names
4
5     def __iter__(self):
6         return PeopleIterator(self.names) # what is this?!
```

Make your own Iterator (cont.)

`PeopleIterator` is our iterator object, we need a new class

Iterator requirement 2: The iterator object is also new data type and supports the `__next__()` method

```
1 class PeopleIterator:
2     def __init__(self, names):
3         # initialised with some data necessary for the iterator
4         self.names = names
5
6     def __next__(self):
7         # returns the next name in the collection
8         return ???
```

Make your own Iterator (cont.)

Iterator requirement 3: The iterator class needs to define how the iterator state is updated, what element is to be returned on each call to `__next__()` method and raise the exception `StopIteration`

Make your own Iterator (cont.)

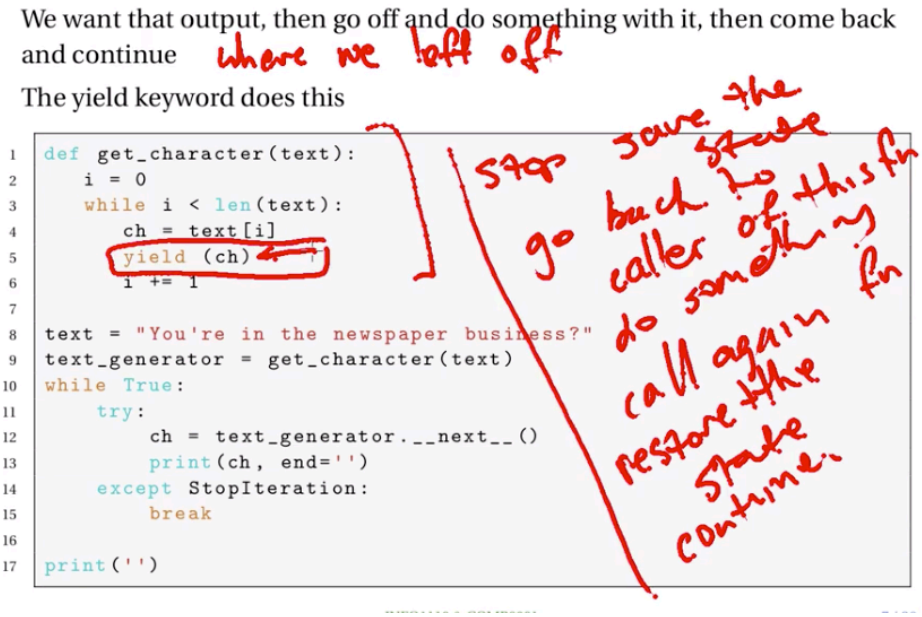
```
class PeopleIterator:
    def __init__(self, names):
        self.names = names
        # the state of the iterator is defined as the index of the
            list
        self.cursor = 0

    def __next__(self):
        # if the end of the list reached, raise exception
        if self.cursor >= len(self.names):
            raise StopIteration("Reached end")

        # get the next element to return
        ret_val = self.names[self.cursor]

        # update the iterator state to the next element
        self.cursor += 1

        # return the value
        return ret_val
```

Generators – Producing a Stream of Data

- Produces a stream of data → creating them is easy if you have the freedom to define new types and can save the state of the iterator object.
- What if you already had a function which produced the __next__() equivalent?
- The function itself already has state built in. When you have local variables, and they possess values, that reflects the state of execution.

Yield

- When we want that output, then go off and do something with it, then come back and continue where we left off, the yield keyword does this.



That was easy!

With a generator, we also have an iterable. Meaning we can use for loops

```
text = "You're in the newspaper business?"
for ch in get_character(text):
    print(ch, end='')

print('')
```

Any function containing a yield keyword is a generator function

Generators are identified during the compilation process

Generators

1. Suspend execution of function, preserving the state of all local variables, and the position of last instruction executed
2. Return the object/value to the caller of the function
3. Upon being called again, restore the state of the function and resume execution

---

Built-In Functions with Iterators

There are already built in functions in Python that can act on iterable objects. These can be iterators, generator iterators, or other collection types (supported by Python)

`sum(iterable[, start])` - Sums start and the items of an iterable from left to right and returns the total.

`len()` - Return the length (the number of items) of an object.

`max(iterable, *[, key, default])` - Return the largest item in an iterable.

Filter

`filter(function, iterable)` - Construct an iterator from those elements of iterable for which function returns true.

Example: return all the numbers that are in the range [0,50)

```
def is_0_to_50(n):
    if n >= 0 and n < 50:
        return True
    return False

nums = [0, -1, 4, 55, 25, 49, 50]
result = list(filter(is_0_to_50, nums))
print(result)
```

Map

Programming idiom transform *each input element* in a way defined by function `f()` and produce a *new collection*

Example: add one to each value

```
def add_one(n):
    return n + 1

nums = [0, 1, 2, 3, 4]
nums2 = list(map(add_one, nums))
print(nums2)
```

Reduce

Programming idiom taking two parameters to be *accumulated* to one. When applied on a collection of *n* elements, it will operate a function on pairs of elements to return *a single element.*

The `sum()`, `max()` and `min()` functions are examples of reduce operations

Example: the average value. Has multiple input values, output is one number.

> `reduce(function, iterable[, initializer])` *Apply function of two arguments cumulatively to the items of iterable, from left to right, so as to reduce the iterable to a single value.*

There is not always a solution for many to one. In this case, thinking about this carefully, the average is `sum(numbers) / len(numbers)`. That is one reduce operation performed already.

Reduce style calculation with our own sum

```
from functools import reduce
def mysum(x,y):
    return x+y

numbers = [0, 1, 2, 3, 4]
print(reduce(mysum, numbers)/len(numbers))
```

Functional Programming

- Perform all operations as functions using only input and output.
- Do not store any state information outside the function call (side effects).
- The goal is to transform the input data to output using a well chosen set of functions.
- Many benefits: security, testing, readability, scalability (complexity).

Functional (pure)

```
1 print(*list(map(len, [ "I", "set", "the", "toaster", "to", "three
      ", "-", "medium", "brown." ] )))
```

There are 4 functions used to solve this problem.

- Each is very simple.
- Does one thing well.
- Has input, has output.
- There is no state stored inside the functions being used
- There are no temporary variables needed to transfer the output of one function as the input to another
- The entire program is one function call and this will produce a result.

A Small Note on Code

Small can be good. Not always.

Optimising code for performance is almost always a wasted effort in early stages of software development. It is dangerous for readability and maintainability of software

Optimising code for readability is king. Almost all industry wants maintainable software products with large development teams.

Small code can be good if it really simple and comprehensible. For example `x += 1` is `x = x + 1`. `for` loop vs a `while` loop, list comprehensions can easily become obscure.

Lambda Functions

lambda functions are a way to define a function without a formal declaration. It is also known as an anonymous function, it does not have a name, and is defined in the place where it is used

Example

```
def x_mult_y(x,y):
    return x * y

z = x_mult_y(3, 5)
```

This is most useful with our reduce operation

```
from functools import reduce
z = reduce(lambda x,y: x*y, [3, 5])
```

Week 12 Lecture – More Data Types and Arrays

Python has seemingly simple types (string, int, float) and hide the memory details

We need to understand the representation of data types and their limits generally.

The data type and its representation is very important to define clearly.

There are cases that we need to be careful about which data type we choose.

numpy is a module for python that is specifically targeting scientific applications. We will be using this module.

```
import numpy
# we have access to the numpy module using numpy.<something>

import numpy as np
# we have access to the numpy module using np.<something>
```

Signed integers can be both positive and negative

Any bit pattern can be associate with any symbol. There are no real restriction of defining this mapping:

bits	value
000	-3
001	1
010	-4
011	-2
100	0
...	...

e.g. someone decided
the letter `A` will be `0100 0001`, and
the letter `a` will be `0110 0001`.

However, if we want arithmetic operators on bit patterns to produce results consistent with mathematical + - / *, then we need a method having both:
1) Specific bit patterns to map to signed and unsigned numbers and ;
2) Specific sequence of bitwise operations for said operators to be performed correctly.
Two's complement is the most common method.

bits	value	1st bit
000	0	positive
001	1	positive
010	2	positive
011	3	positive
100	-4	negative
101	-3	negative
110	-2	negative
111	-1	negative

type	bits	range
uint8	8	[0, 255]
int8	8	[-128, 127]
uint16	16	[0, 65535]
int16	16	[-32768, 32767]
uint32	32	[0, 4294967295]
int32	32	[-2147483648, 2147483647]
uint64	64	[0, 18446744073709551615]
int64	64	[-9223372036854775808, 9223372036854775807]

- Integer overflow occurs when we try to represent values outside the allowable range.

Example of integer overflow

```
1  import numpy as np
2
3  for i in range(20):
4      x = i + 240
5      y = np.uint8(x)
6      print(y, end=' ')
7  print("")
```

Expect:

```
240 241 242 243 244 245 246 247 248 249 250
251 252 253 254 255 256 257 258 259
```

Actual:

```
240 241 242 243 244 245 246 247 248 249 250
251 252 253 254 255 0 1 2 3
```

Floats

To represent numbers using a decimal point e.g. 3.14, 0.0023, -6.21×10^3

`float` is referred to as *floating point* types.

Typical `float` type uses 4 bytes of memory, but Python 3+ has arbitrary precision.

The range and the accuracy of floating point types depends on how many bits are used in the *coefficient*, and the *exponent*. Python will manage this automatically.

Errors with floating point numbers

```
1 import numpy as np
2
3 y = np.float16(1)
4 y = y / 0
5 print("still going...")
6 if np.isinf(y):
7     print("we found an infinite number, but keep going")
```

```
divzero.py:4: RuntimeWarning: divide by zero encountered in true_divide
   y = y / 0
still going...
we found an infinite number, but keep going
```

...we don't stop. Allows for error handling at a later time by the programmer.

Fixed Bit Width Data Types

- Why do we care about knowing fixed bit width data types?
- Are you alone? Probably not.
- Is the data stationary? Probably not.
- Will the program run for a really long time? Maybe. How long is really long?

Lists

- The list has been used to describe a collection of ojects.
- The list is a kind of data type. It is a more complicated data structure and can do further operations on collections. Automatically resize, insert in the middle, delete from the middle.
- It relies on an iterator to get from one element to the next. This is because the elements are not necessarily adjacent in memory.

Array

- Array is the simplest form of representing a collection of a single data type in a contiguous area of memory.

- Contiguous memory: that is, if we look at how bits are laid out in memory, they are all adjacent and have no gaps.
- Python arrays exist, largely based on the C programming language types. However, we will focus on numpy arrays and fixed width data types.

Further information for Numpy

Make your own types with dtype (there are no new classes to define)

Reading real data from file directly into array. Big topic, not covered. (.NPY as binary storage really fast!)

Special values for missing data `NA` are also considered with statistical packages within numpy as to not skew the results. `NaN`'s can be filtered and also replaced with other values.

Numpy is separate to Python. It is part of the Scientific Computing Tools for Python ecosystem.

Multi-Dimensional Arrays

Declaring a multi-dimensional list is easy

Each dimension has another index []

e.g. 1D `line = np.zeros((5)).astype('int32')`

e.g. 2D `grid = np.zeros((2,2)).astype('int32')`

e.g. 3D `volume = np.zeros((3,3,3)).astype('int32')`

e.g. 6D `hyperspace = np.zeros((3,3,3,3,3,3).astype('float64')`

We will keep it simple and use 2D. The same ideas apply when extending to higher dimensions

Initialise a 2D array

```
grid = np.ones((2,2)).astype('int32')
```

creates a new array of (2 × 2 =) 4 int32s

We access them in the same way as for 1-dimensional arrays:

```
grid[0][0] = 7
grid[0][1] = 7
grid[1][0] = 7
grid[1][1] = 7
```

Which we can imagine, it might look like this:

$$\begin{pmatrix} 7 & 7 \\ 7 & 7 \end{pmatrix}$$

We need a procedure to visit each element in a 2D array and print the value

Such a procedure is termed a *traversal*

```
# initialise
grid = np.zeros((2,2)).astype('int32')

# our traversal to assign the initial value
for x in range(2):
    for y in range(2):
        grid[ x ] [ y ] = 1

# our traversal to display values
for x in range(2):
    for y in range(2):
        print(grid[x][y], end=" ")
    print("")
```

- 5 rows, 10 columns.
- When using the len function, it will return the size of the 1st dimension (len(grid)), and the size of the second dimension (len(grid[2]).

Arrays of arrays

To understand what's going on you need to know an important fact: multidimensional arrays are stored as arrays of arrays.

An array of k dimensions is stored as a 1-dimensional array of arrays of $(k-1)$ dimensions each.

```
grid = np.zeros((5,10)).astype('int32')
print("dim 1:   " + str(len(grid)))
print("dim 2:   " + str(len(grid[0])))
```

So in the example above, `grid.length` is the length of the first dimension: it's the number of 1-dimensional arrays

`grid[0][0]...grid[0][9]` is the first "row", `grid[1][0]...grid[1][9]` the second, etc.

- Theres an exception thrown when python tries to access an index that is out of range (IndexError).

Summary: Multidimensional Arrays

- The ordering of rows, columns, and other dimensions should be very carefully considered.
- Traversals through the multi-dimensional array should always be the same for the same effect.
- Arrays have strict types for element, some of which are strict bit width (you can still have strings as types).
- Arrays are best used for achieving memory efficiency.

Procedural Programming

Object Oriented Programming – Classes, Objects, Methods

Functional Programming – includes iterators and generators.

What does _ _ mean?