INTRODUCTION TO PROGRAMMING USING PYTHON

What do CPUs do?

- Keep track of program execution
- Execute computer program instructions
  - Native computer programs contain instructions in binary format
  - Binary instructions talk to the CPU
- Store data
- Transform data
- Store data in RAM memory
- Retrieve data from RAM memory
- Send and receive data to and from hardware devices

Program
0101010010
1010101010
01010101

CPU instructions

- Low-level instructions are specific to CPU families
- Programmers can develop in:
  - machine code - binary (base 2) or hexadecimal (base 16)
  - assembly language
- What does this code sample do…?

Machine code (in Hex)

Equivalent assembly language opcode operands

```
06B0:0100 B4 09                     MOV AH, 09
06B0:0102 BA 09 01                  MOV DX, 0109
06B0:0105 CD 21                     INT 21
06B0:0107 CD 20                     INT 20

06B0:0109 48 65 6C 6C 6F 2c 20 57 6F 72 6C 64 21 24
```

Memory address (segment:offset notation)

ASCII string (in Hex)

High and low level languages

Programming languages can be listed roughly in a ladder going from 'nearer machine code' up to 'nearer normal language'.

The higher level a programming language is, the closer it is to a natural language, eg English.

Human friendly

- **SQL**
  - SQL
  - Select customer name from orders where town = 'Leeds';
- **Python**
  - Python
  - mark = sum(marks)/100
- **Java**
  - Java
  - int var = 42;
- **C**
  - C
  - int var = 42;
- **Assembly language**
  - Assembly language
  - MOV AX, 1
- **Machine code**
  - Machine code
  - 1100101000110101

Computer friendly

Languages for different problems

Why so many different programming languages?

Low level languages may be needed for writing operating systems and hardware interfaces like device drivers.

Most high level languages are general purpose, though some are specialised for particular areas, or have useful libraries of code available – for example:

- Python and R for Data Science,
- Rust for secure coding,
- Go for efficient infrastructure,
- C or FORTRAN for speed of processing
- JavaScript for web applications.

Some become popular because of ease of use.

Some develop from academic projects to try out ideas eg Haskell

Content 1

- **What is Python?**
  - Basic facts about Python
  - Python as a programming language
  - Anatomy of a Python program
- **Using Python**
  - Interactively
  - Using IDLE
- **Python and Data**
  - Python variables
  - Python class types
  - References and mutability
- **Python Operators**
  - Different types of operators
  - Arithmetic operators
  - Relational operators
  - Logical operators
- **Python I/O**
  - Input / Output techniques
- **Python constructs**
  - Sequence
  - Selection
  - Iteration

Content 2

- **Data collections**
  - Mutability vs Immutability
  - Lists
  - Tuples
  - Dictionaries
  - Sets
  - Comprehensions
- **Python Modularity**
  - What is modularity?
  - Python built-in functions
  - Calling a function
  - Python modules and functions
  - Python user-defined functions
  - Generators
- **Testing**
  - Types of test
  - Performing a coverage test
  - Performing a unit test
- **File Input / Output**
  - Text files
  - Structured text file formats (CSV, XML, JSON)
  - Binary files
  - Pickling
  - Compression

Content 3

- **Practical OOP**
  - Classes
  - Attributes, Properties and Methods
  - Objects
  - Inheritance
  - Static and Class methods
  - ABCs (extension exercise)
- **Sockets & Network communications**
  - Connection-based communication
  - Basic Client and Server scripts
- **Command Line Interaction**
  - Parsing command line arguments
- **Creating a GUI**
  - Tkinter/TTK
- **Interfacing with Python on SCB’s**
  - Raspberry PI
  - GPIO
  - Raspbian and PIXEL
  - Connecting via VNC
  - Controlling a basic circuit

Python: The basic facts

- Python is an object-oriented scripting language.
- First published in 1991 by Guido van Rossum
- It is a powerful, but general-purpose language
- It has a rich feature set which is constant evolving
- It is free (open source: Python licence is less restrictive than GPL)

Python 3.11
is the current version;
Legacy 2.X code is still in use
- its EOL was 2020

Python as a programming language

- Case sensitive; primarily lower case
- Attempts to keep syntax simple where possible
- Uses indentation (spaces, **not** tabs) to associate blocks of program logic
- Lines of program code are terminated by a carriage return (new line)
- It is a strongly data-typed language and is dynamically typed (not statically typed)
- Has automatic memory management (garbage collection)
- Easily extended by the creation of new modules (in Python or C)
- **P**ython **E**nhancement **P**roposals (**PEPs**) give the community their technical standards and release information on new features as the language continues to evolve – Search on www.python.org

Anatomy of a Python program (script)

shebang – tells Linux which interpreter to load – benign on Windows

Comments; ignored by Python.

initialisation of variables (references)

Obtain input from standard input (STDIN, the keyboard) and store in a variable

Conditional logic (selection)

Arithmetic calculation (expression)

Messages shown on standard output (STDOUT, the screen)

```
#!/usr/bin/python

# Example Python script to calcuate gross pay
# Author: QA

salary = 0.0
hours_worked = 0
hourly_rate = 10.0

hours_worked = int(input("How many hours worked?"))

if hours_worked > 0:
    salary = hours_worked * hourly_rate
    print("Your gross pay is ", salary)
else:
    print("Sorry, you must enter your hours worked first.")
```

USING PYTHON

MODULES

Using Python interactively

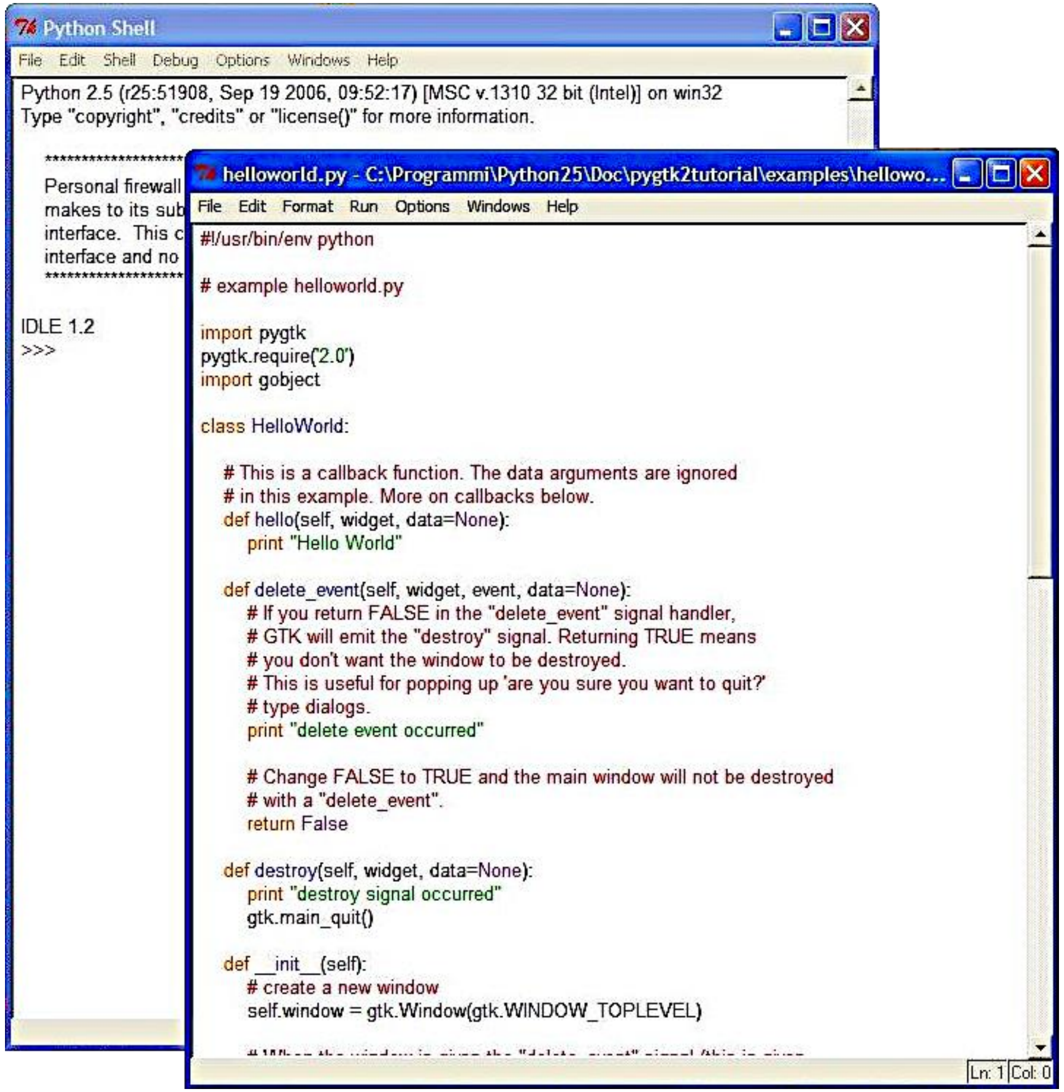
Python's shell-based **interpreter** allows the programmer to enter commands manually. A **">>>"** prompt indicates the **primary prompt** (it is awaiting a command), the **"..."** prompt indicates a **secondary prompt** (a continuation is required). This is similar

```
================================ RESTART: Shell ================================
>>> import math
>>> value = 121
>>> print ("The square root of",value,"is",math.sqrt(value))
The square root of 121 is 11.0
>>>
```

The Python interactive mode is useful for testing small snippets of code. When code gets more complex and needs to be saved for

Using Python's **I**ntegrated **D**evelopment and **L**earning **E**nvironment (IDLE)

- Coded in 100% Python
- GUI text-editor with syntax highlighting
- Debugger with breakpoint, stepping and namespace viewer
- Python shell window for execution of code; handling input and output



- Download Python 3.X (including IDLE) for Linux, Mac OS X, Microsoft Windows from:

Python variables (names that point to objects in memory)

- A reference to a specific object (or many objects stored in a specific structure)
- Case sensitive; must start with a letter or underline character, not a number and can only contain a-z, A-Z, 0-9 and _ (underscore) characters
- Can use built-in function name (monkey-patching) but cannot be a reserved keyword
- Unlike many programming languages there is no formal declaration of variable data

```
>>> my_val = 10
>>> type(my_val)
<class 'int'>
>>> my_name = "Tom"
>>> type(my_name)
<class 'str'>
>>> my_wage = 250.00
>>> type(my_wage)
<class 'float'>
>>> my_group = ['Tom', 'Alex', 'Ramiz', 'Philipe']
>>> type(my_group)
<class 'list'>
```

Creation and use of variables in the Shell.

Notes.
The "type" builtin function can be used to determine the data type of the variable. The "id" builtin function will provide its unique "identity"; for CPython its address in RAM.

INTERPRET PYTHON STATEMENTS

EXERCISE 1

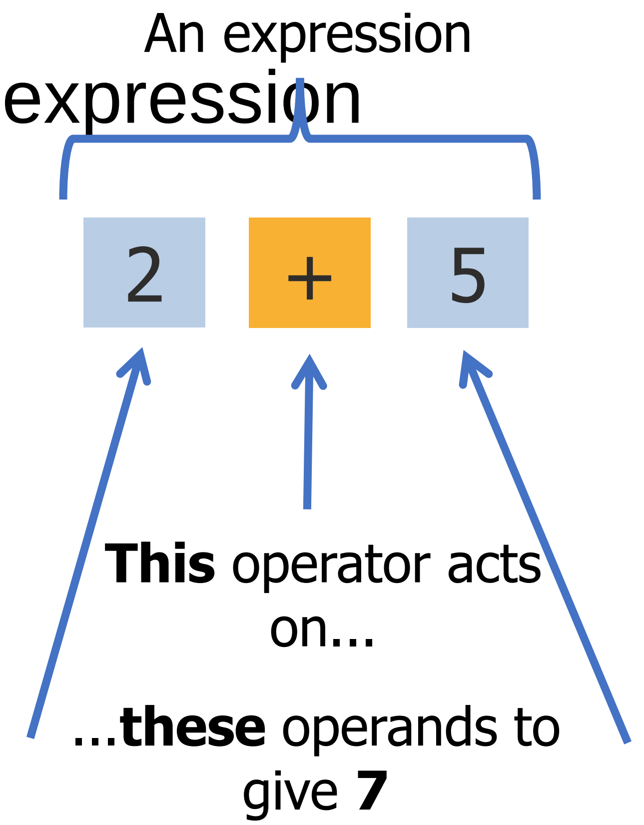
PYTHON OPERATORS

Python: Different types of operator

An operator **manipulates** operands in a legal expression

Broadly speaking, there are **7** different types:

- **Arithmetic**
- **Assignment**
- **Relational**
- **Logical**
- Bitwise
- Membership
- Identity



The ones highlighted in **green** are, broadly speaking, the ones most

Python: Operator precedence and associativity

Operators	Purpose
( )	Parentheses
**	Exponent
+x -x ~x	Unary plus, Unary minus, Bitwise NOT
* / // %	Multiplication, Division, Floor division, Modulus (remainder)
+ -	Addition, Subtraction
<< >>	Bitwise shift operators
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
== != > >= < <= is is not in not in	Comparisons, Identity, Membership operators
not	Logical NOT
and	Logical AND
or	Logical OR
:=	Walrus operator (assignment expression)

Highest precedence

Lowest precedence

Arithmetic and Assignment Operators

Arithmetic Operators		
+	Addition	10 + 30 = 40
-	Subtraction	10 – 30 = -20
*****	Multiplication	10 * 30 = 300
/	Division	30 / 4 = 7.5
%	Modulus	30 % 4 = 2
******	Exponent	10**2 = 100
//	integer division (floor)	30 // 4 = 7

Assignment Operators			
=	Assignment	num = 12	
+=	Add right operand to the left	num += 2	#num is 14
-=	Subtract right operand from left	num -= 3	#num is 11
***=**	Multiply left operand by right	num *= 4	#num is 44
/=	Divide left operand by right	num /= 4 num /= 5	#num is 11.0 #num is 2.2
%=	Calculate right operand modulus of left operand	num %= 6	#num is 2.2
****=**	Apply exponent	num **= 2	#num is 4.84
//=	Apply floor division	num //= 2	#num is 2

Relational and Logical Operators

Python operators are often used in conjunction to form more complex expressions, e.g.

Relational Operators		
==	Equal to (equality)	10==4+6 (true)
!=	Not equal to (inequality)	10!=30 (true)
>	Greater than	20 > 10 (true)
<	Less than	5 < 4 (false)
>=	Greater than or equal to	10 >= 10 (true)
<=	Less than or equal to	12 <= 10 (false)

Logical Operators	
not	Negation
and	Logical AND
or	Logical OR

```
banned_list = ['secret','password','letmein']
min_length = 6
user_password = input ("Password")

if len(user_password) >= min_length and user_password not in(banned_list):
    print("Password accepted!")
else:
    print("Password too short or just silly!")
```

SIMPLE PYTHON EXPRESSIONS

EXERCISE 2

Input and Output (standard I/O)

In Python, simple user interactions can be achieved using the standard input/output devices – the **keyboard** and **screen**. This is achieved u

Simple input (keyboard; standard input)	**Simple output** (screen; standard output)
`#basic input, no prompt` `my_text = input()`	`#basic output` `print (my_text)`
`#basic input, with optional prompt` `my_text = input("Enter your text:")`	`#basic output, with specific separator and end` `print ("Your text is", my_text, sep=":", end=".\n")`

`#basic input, explicitly converting data type`
`my_number = int(input("Enter your age (in years):"))`
	`import math`
	`#basic output, with formatted string literals`
	`print (f"Pi is {math.pi:.4f}")`

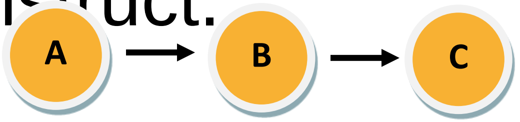
Note: The default end character output is **\n** (newline), the default separator is a single **space**.
Either pairs of single or double quotes may be used.

USING STANDARD I/O

EXERCISE 3

Sequence

In programming, a sequence occurs when a group of statements is executed individually, with none repeated and none missed. It is the most basic programming construct.



<action A>

<action B>

<action C>

```
# Simple Python script to demonstrate
# a sequence of statements
# Author: QA

name = input("Please enter your name: ")
length = len(name)
print ("There are",length,"characters in", name)
```

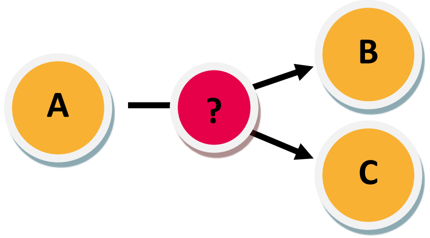
```
Please enter your name: Tommy
There are 5 characters in Tommy
```

On their own, sequences limit program logic to linear experiences.

Selection (2 choices)

Selections (or branches) are an essential aspect of Python programming, allowing us to make choices.

The “if” statement is the most well-known selection statement:



if <condition is true>:

<action B>

else:

<action C>

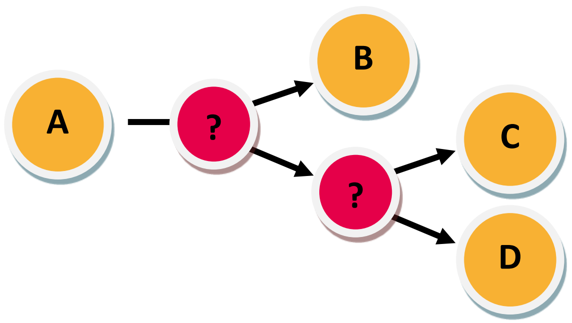
```
# Example Python script to calculate
# sum or difference of two integers
# Author: QA
result = 0
number1 = int(input("1st number:"))
number2 = int(input("2nd number:"))
choice = input("Select + or - :")
if choice == "+":
    result = number1 + number2
else:
    result = number1 - number2
#endif
print(number1,choice,number2,"=",result)
```

```
1st number:9
2nd number:10
Select + or - :+
9 + 10 = 19
```

```
1st number:9
2nd number:10
Select + or - :-
9 - 10 = -1
```

Selection (3 or more choices)

Multiple choice branches in Python can be coded using a traditional "**nested**" if statement, although neater alternatives exist:



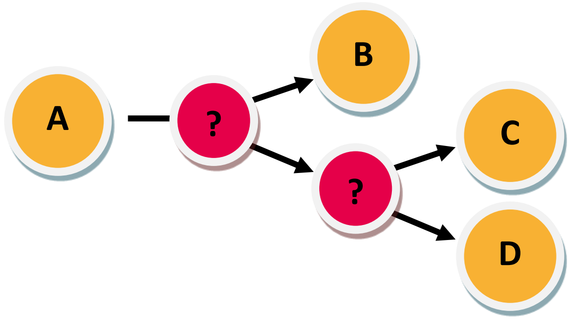

```
if <condition1 is true>:
    <action B>
else:
    if <condition2 is true>:
        <action C>
    else:
        <action D>
```

```python
# Example Python script to calculate
# sum, difference or product of two integers
# Author: QA
result = 0
number1 = int(input("1st number:"))
number2 = int(input("2nd number:"))
choice = input("Select +, - or *:")
if choice == "+":
    result = number1 + number2
else:
    if choice == "-":
        result = number1 - number2
    else:
        result = number1 * number2
    #endif
#endif
print(number1,choice,number2,"=",result)
```

```
1st number:9
2nd number:10
Select +, - or *:*
9 * 10 = 90
```

Selection (3 or more choices)

Multiple choice branches in Python can also be coded using the neater **elif** statement, which avoids excessive indentation:



if <condition1 is true>:
   <action B>
elif <condition2 is true>:
   <action C>
else:
   <action D>

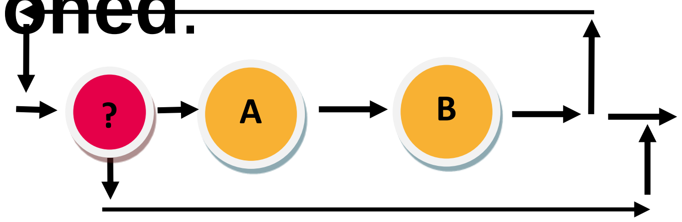
```
# Example Python script to calculate
# sum, difference or product of two integers
# Author: QA
result = 0
number1 = int(input("1st number:"))
number2 = int(input("2nd number:"))
choice = input("Select +, - or *:")
if choice == "+":
    result = number1 + number2
elif choice == "-":
    result = number1 - number2
else:
    result = number1 * number2
#endif
print(number1,choice,number2,"=",result)
```

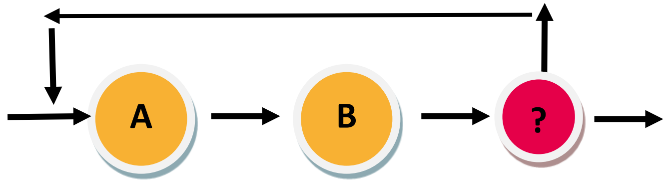
```
1st number:9
2nd number:10
Select +, - or *:*
9 * 10 = 90
```

Iterations

An iteration (or loop) allows the programmer to **repeat** a chosen block of Python code 0 or more times while a given condition is **true**. When the controlling condition becomes **false** the loop is **terminated** and normal flow is resumed.

In traditional programming, iterations can be **pre-** or **post-conditioned**.





Iterations – for loop

Use Python's pre-conditioned **for loop** when the number of iterations to be performed **is known** at the start.

Simple examples

```
#a simple counter using the range function
for counter in range(0,5):
    print (counter)
```

```
#a stepped counter using the range function
for counter in range(1,11,2):
    print (counter)
```

```
#a simple decrement counter using the range function
for counter in range(4,-1,-1):
    print (counter)
```

```
#using a string as an iterator
string = "Each letter"
for letter in string:
    print (letter)
```

Advanced examples

```
# Example Python script to process a
# list of 4 module exams re
delegate
# Author: QA

# Module scores
module_scores = [46, 88, 90, 56]

# version 1 - has problems...
for a_score in module_scores:
      print("Exam",
module_scores.index(a_score) + 1, ":",
a_score)

# version 2 - not Pythonic...
```

```
Exam 1 : 46
Exam 2 : 88
Exam 3 : 90
Exam 4 : 56
```

Iterations – while loop

Use Python’s pre-conditioned **while loop** when the exact number of iterations **may not be known** at the start.

<table>
<tr><th>Simple examples</th><th>Advanced example</th></tr>
<tr><td>

```
#Simulating a simple for loop
counter = 0
while counter < 5:
    print ("Counter is",counter)
    counter+=1

print ("Loop completed.")
```

</td><td rowspan="2">

```
# Example Python script to validate a password.
# Author: QA

#minimum acceptable length
MIN_LENGTH = 6

password = input("Enter your password: ")
length = len(password)
while (length < MIN_LENGTH):
    print ("Sorry, must be at least", MIN_LENGTH, "characters.")
    password = input("Enter your password: ")
    length = len(password)
else:
    print("Your password change has been accepted.")
```

</td></tr>
<tr><td>

```
#Simulate post-conditioned loop
while True:
    print("Do something...")
    response = input("Again (y/n)? ")
    if response =='n':
        break

print ("Finished!")
```

</td></tr>
</table>

CONSTRUCTS

EXERCISE 4

Mutable vs Immutable

- **Mutable** objects can be modified after they have been created in RAM.
- **Immutable** objects cannot be modified; another object may be
- Inspecting their "id" usually demonstrates this.

List (Mutable object)

```
>>> my_cats = ["Frank", "Jess", "Ron"]
>>> id(my_cats)
47359440
>>> my_cats.append("Phil")
>>> my_cats
['Frank', 'Jess', 'Ron', 'Phil']
>>> id(my_cats)
47359440
>>> for cat in my_cats:
        print(id(cat))
47376416
47376448
47376480
47558048
```

String (Immutable object)

```
>>> my_cat = "Phil"
>>> id(my_cat)
47558048
>>> my_cat = "Phil" + " the cat"
>>> my_cat
'Phil the cat'
>>> id(my_cat)
40897456
```

Python 3 types

- A **sequence** is the generic term for an **ordered** collection. There are several types of sequences in Python; **Strings, Lists and Tuples** are the most important
- However, Strings and Tuples are both immutable.
- Dictionaries and

Immutable

Numbers

```
3.142, 42, 0x3f, 0o664
```

Bytes

```
b'Norwegian Blue', b"Mr. Khan's bike"
```

Strings

```
'Norwegian Blue', "Mr. Khan's bike",
r'C:\Numbers'
```

Tuples

```
(47, 'Spam', 'Major', 683, 'Ovine Aviation')
```

Sequence

Mutable

Lists

```
['Cheddar', ['Camembert', 'Brie'], 'Stilton']
```

Bytearrays

```
bytearray(b'abc')
```

Dictionaries

```
{'Sword':'Excalibur', 'Bird':'Unladen Swallow'
```

Sets

```
{'Chapman', 'Cleese', 'Idle', 'Jones', 'Palin'
```

Lists

- Lists are considered the "workhorse" data container of the Python language.
- Most Python solutions will make use of list objects at some point.
- You should become confident at creating them, modifying them and at using their most popular methods.

- Try:

```
[['Phil', 128.5], ['Ron', 76.5], ['Frank', 90.0],
['Jess', 50.5]]
```

you can do with a list (methods)

```python
from operator import itemgetter

empty_list = []
mixed_list = ["Phil", 4, 128.5]
cat_list = ["Phil", "Ron", "Frank"]
list_of_lists = [["Phil", 128.5], ["Ron",
76.50], ["Frank", 90.0]]
not_numbers_list = ["4", "77", "2000", "23"]

cat_list.append("Jess")
print(cat_list)

cat_list.reverse()
print(cat_list)

print(cat_list.index("Phil"))

print(len(cat_list))

list_of_lists.append(["Jess", 50.5])
print(list_of_lists)
```

Lists – slicing

- **Slicing** is a powerful feature in Python, enabling the programmer to extract particular parts of a data structure with relative ease.
- It is a particularly useful tool when working with lists and you need to be familiar with its various

	0	1	2	3
cat_list	Phil	Ron	Frank	Jess

```
print(cat_list[0:3:2])
```

Start index

Upto index

Change value

```
# slicing
print(cat_list[0])
print(cat_list[1:2])
print(cat_list[0:3:2])
print(cat_list[::-1])
print(cat_list[::-2])
print(cat_list[0][3])

# slicing on sublists
print(list_of_lists[1][1])
print(list_of_lists[1][0][0:2])
```

Lists – other operations

- Applying scalar operations to a list of values is always useful. The **map** function is very useful and often overlooked.
- **Sorting lists** is also another typical operation.
- Many different techniques exist to handle more complex sorting when a list has to be organised on a particular sub-value.
- The **'*'** in the second example 'unpacks' the results of the map ino print function rather than creating a list from it

```
# convert to a list of numbers
numbers_list = list(map(int, not_numbers_list))
print(numbers_list)

# or to just print the values
print(*map(int, not_numbers_list))

# sorting cats based on operation costs using anonymous lambda function
sorted(list_of_lists, key=lambda cat: cat[1])
print(list_of_lists)

# sorting cats based on names (newer method)
sorted(list_of_lists, key=itemgetter(0))
print(list_of_lists)

# removing cats
homeless_cat = cat_list.pop()
print(homeless_cat)
print(cat_list)

del cat_list[0]
print(cat_list)
```

Tuples

- Tuples are immutable sequences and are typically used to store read–only data.
- They **do not** require brackets in their assignment as the comma alone is sufficient. i.e the comma is the composer of tuples.
- They are also useful for quick "swap" operations…
- As we will see many Python functions and methods return a tuple object.

```
>>> my_data = "ABC", 123
>>> print(my_data)
('ABC', 123)
>>> print(type(my_data))
<class 'tuple'>
>>> my_data2 = ("ABC", 123)
>>> print(type(my_data2))
<class 'tuple'>

>>> a = 10
>>> b = 20
>>> b, a = a, b
>>> print(a)
20
>>> print(b)
10

>>> letters, numbers = my_data
>>> print(letters)
'ABC'
>>> print(numbers)
123
```

Dictionaries

- Dictionaries store data in **key:value** pairs. A pair could be any valid object – even another dictionary.
- Each key **must** be unique and hashable (immutable).
- A KeyError will occur if an non-existent key is requested (there is a fix for this).

```
dict_cats = {"Phil": 128.5, "Ron": 76.50, "Frank": 90.0}

print(dict_cats["Phil"])

# print(dict_cats["Jess"])

print("Jess" in dict_cats)

# append using subscripted key
dict_cats["Jess"] = 50.5
print(dict_cats)

# append using update
dict.update({'Jess': 50.5})
print(dict_cats)

# get the number of objects in the dictionary
print(len(dict_cats))
```

Dictionaries - continued

- It is possible to access the dictionary's keys or values separately.

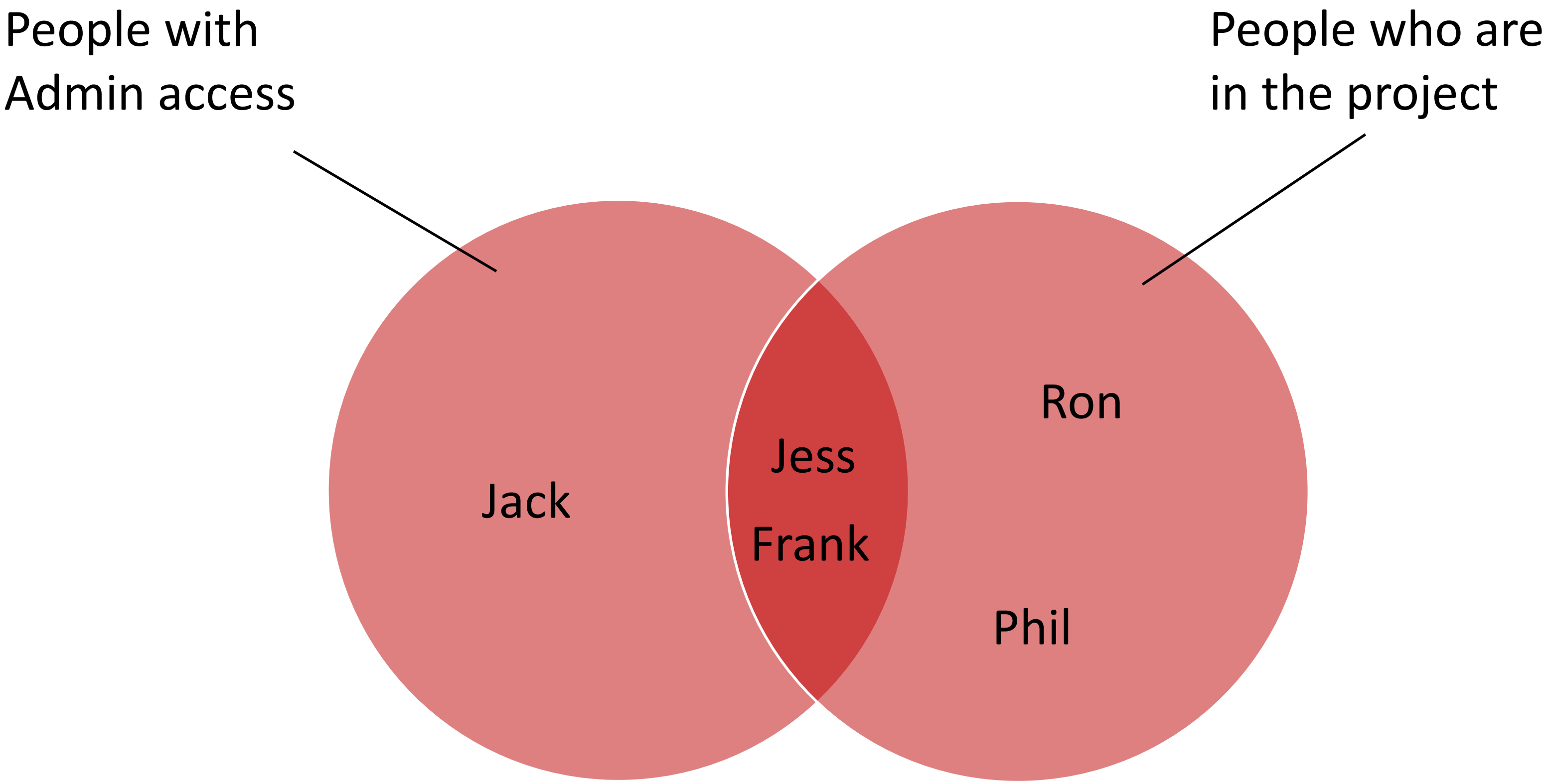
- Removing an item can be achieved a number of different ways. Del can also be used but will not return the item ("pop" does).

- Displaying all items

```
# just the keys
print(dict_cats.keys())

# just the items as iterable key:value tuples
print(dict_cats.items())

# remove an item
dict_cats.popitem()
print(dict_cats)

# remove a specific item using the key
dict_cats.pop("Frank", False)
print(dict_cats)

# looping (using best practice)
for key, val in dict_cats.items():
    print(f"{key} has {val}")

for key in dict_cats:    # Multiple lookups
    print(f"{key} has {dict_cats[key]}")

# removes all objects from the dictionary
dict_cats.clear()
print(dict_cats)
```

Sets - introduction

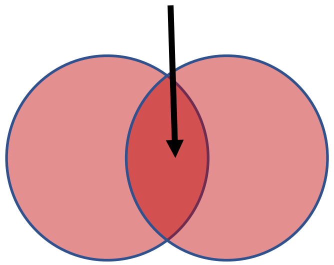
- Allow you to manage membership groups
- Venn diagrams



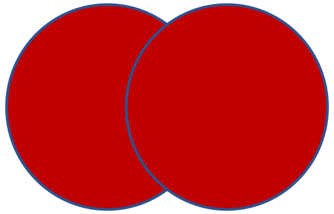

```
admin_users = {'Frank', 'Jess', 'Jack'}
project_users = {'Ron', 'Jess', 'Frank', 'Phil'}
```

Sets - Visualisations

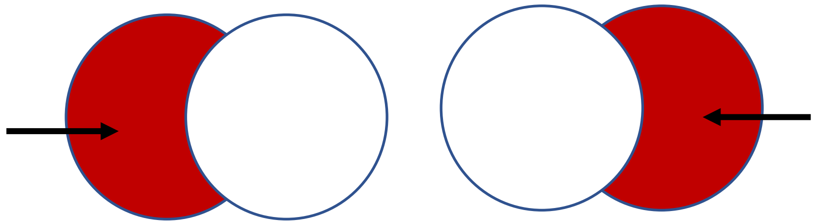
- **Intersect – '&' operator**
  - Common to both sets



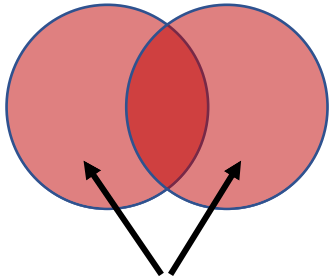
- **Union - pipe '|' operator**
  - Combination of both sets



- **Difference - minus '-' operator**
  - One set minus the common members of the other, depending on order



- **Symmetric difference – '^' operator**
  - Unique to either set



Sets – Python Methods

```
#users who are in both groups
print(project_users.intersection(admin_users))
```

{'Frank', 'Jess'}

```
#all users
print(project_users.union(admin_users))
```

{'Phil', 'Ron', 'Frank', 'Jess', 'Jack'}

```
#project users who aren't admin
print(project_users.difference(admin_users))
```

{'Phil', 'Ron'}

```
#admin users not working on the project
print(admin_users.difference(project_users))
```

{''Jack'}

```
#all users who only belong to one group
print(admin_users.symmetric_difference(project_users))
```

{'Jack', 'Phil', 'Ron'}

Comprehensions

- **Comprehensions** are a *Pythonic* way of creating populated data containers such as Lists, Sets and Dictionaries.
- The syntax is both concise and powerful and *reduces* the need for using functions such as **map** and **filter**.

```
#creates a list of squares of integers between 1 and 10
my_squares = [num **2 for num in range(1, 11)]
print(my_squares)
```

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

```
#creates a list of even number between 1 and 50

just_evens = [num for num in range(1, 51) if num % 2 == 0]
print(just_evens)
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26,
28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50]
```

```
my_cats = ["Ron", "Phil", "Jess", "Frank"]
my_cats_lengths = {name: len(name) for name in my_cats}
print(my_cats_lengths)
```

```
{'Ron': 3, 'Phil': 4, 'Jess': 4, 'Frank': 5}
```

EXTENSION ACTIVITIES

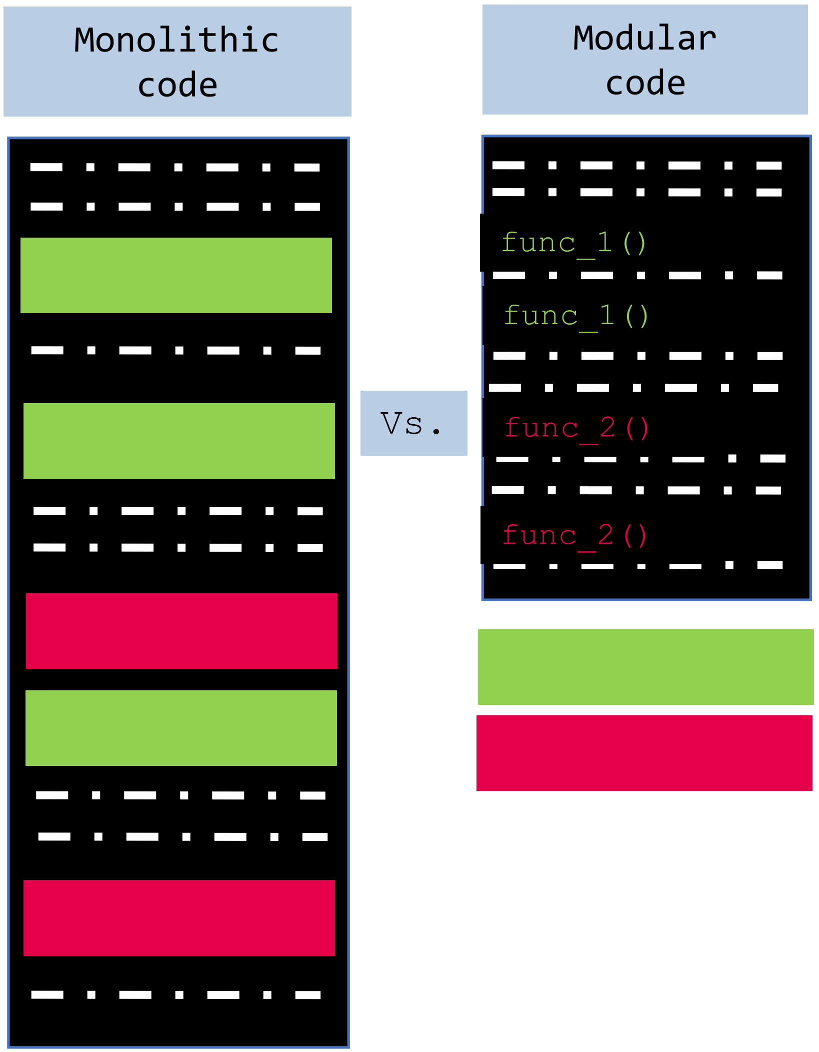
EXERCISE 5

What is modularity?

1. Create small, **reusable** blocks of code with one clear purpose (SRP)
2. Link them to form a solution

Why do this?

+Reduces complexity
+Improves readability and maintenance
+Assists debugging and testing
+Minimizes unnecessary duplication
+Encourages code reuse (DRY not WET!)
+Improves scalability and extensiveness
+Enables team-based solutions



Built-in functions ("built-ins")

Python's interpreter supports a number of different pre-written **built-in** functions. These can be used immediately.

	callable()	eval()	help()	locals()	pow()	sorted()
abs()	chr()	exec()	hex()	map()	print()	staticmethod()
all()	classmethod()	filter()	id()	max()	property()	str()
any()	compile()	float()	input()	memoryview()	range()	sum()
ascii()	complex()	format()	int()	min()	repr()	super()
bin()	delattr()	frozenset()	isinstance()	next()	reversed()	tuple()
bool()	dict()	getattr()	issubclass()	object()	round()	type()
breakpoint()	dir()	globals()	iter()	oct()	set()	vars()
bytearray()	divmod()	hasattr()	len()	open()	setattr()	zip()
bytes()	enumerate()	hash()	list()	ord()	slice()	__import__()

Correct as per Python 3.7.0

Calling a function

Python's **built-in functions** typically follow this format:

```
[return_value =] function_name([parameter_1], [parameter_2], ...[parameter n])
```

```
my_direction = "South"
print("Direction is", my_direction)
```

2 parameters

```
my_list = [20, 30, -10, 4]
biggest = max(my_list)
print("Largest is", biggest)
```

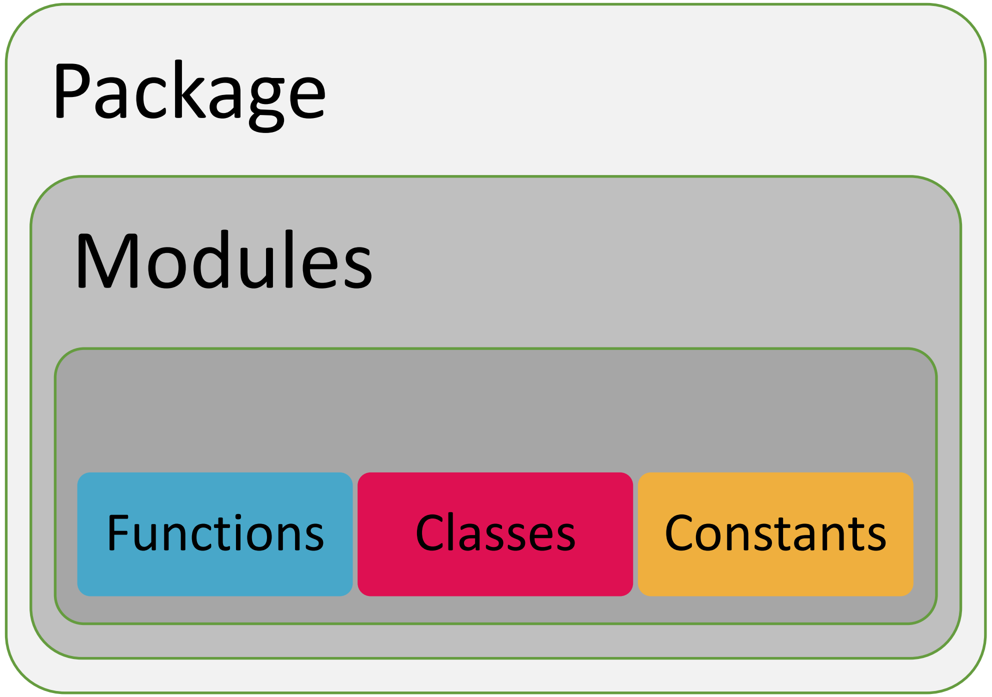
1 parameter, storing result in a variable

```
my_number = input("Enter a denary integer")
my_binary = bin(int(my_number))
print(my_number, "in binary is", my_binary)
```

Nested functions; the result of the inner function becomes the parameter of the outer

Python modules and functions

- Python has several standard modules that form its **Standard Library**
- Some modules are written in C, others are written in Python
- C modules generally provide access to file-based I/O; Python modules provide solutions to common programming problems
- Modules are imported in the Python script using the **import** statement, **one** per line
- Imports are **grouped** (standard



A package is a collection of Python modules, normally located in the same directory. Modules can contain function definitions, classes and constants.

Python modules and functions

This example shows a programmer importing a standard module from Python's Library reference and leveraging it in their solution to solve a basic problem. This is done rather than 'reinventing the wheel'.

This sample package contains a number classes, functions and constants.

```
#show a simple calendar
import calendar

start_day = input("Start of week: (M)onday or (S)unday? ")
month = int(input("Month (1-12)? "))
year  = int(input("Year? "))

if start_day != 'M':
    calendar.setfirstweekday(calendar.SUNDAY)

print ("\n",calendar.month(year,month))
```

```
Start of week: (M)onday or (S)unday? M
Month (1-12)? 8
Year? 2018

     August 2018
Mo Tu We Th Fr Sa Su
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
```

Version control source:

Python user-defined functions

It is quite straightforward to write your own module, e.g. For temperature conversion in a lab environment:

Once written, modules can be extended by adding constants, classes and functions to provide additional functionality – often by different programmers working collaboratively in a team. Caution: do **not use** the same name as an existing module.

```
# Temperature conversion module

def c_to_f(c):     #convert Celsius to Fahrenheit

    f = c * 9/5 + 32
    return f
```

User-defined module: temperature.py

Interactive use of the module and its function in the Shell

```
================================ RESTART: Shell ================================
>>> import temperature
>>> temperature.c_to_f(100)
212.0
>>> temperature.c_to_f(0)
32.0
```

Generators

- Generating large sequences of data and storing them in a list is **wasteful** of memory – particularly if the data is *only needed* on a **per-item basis** for processing.
- A **generator** creates a python-like "pez dispenser"

1
4
9
16
25
36
49
64
81
100

```
#Generates squares of integers between 1 and 10 inclusive

#Returns a list
def old_get_squares(min_val, max_val):
    squares_list = []
    for num in range(min_val, max_val + 1):
        squares_list.append(num ** 2)
    return squares_list


#Yields each square calculated
def get_squares(min_val, max_val):
    for num in range(min_val, max_val + 1):
        yield num ** 2


#Yield using the comprehension syntax (note the brackets)
def better_get_squares(min_val, max_val):
    return (num ** 2 for num in range(min_val, max_val + 1))


#Caller iterates over and prints the generated square values
for each_square in get_squares(1, 10):
   print(each_square)
```

MODULES AND FUNCTIONS

MODULE 6

Types of test

There are three types of test we may consider:

Unit Test

- Part of a test suite which examines the expected outcomes of certain use cases
- An individual test case may focus just on one function

Regression Test

- Typically used to compare outputs from program *before* and *after* code changes to enable comparison and contrast.
- Differences are quickly highlighted

Coverage Test

- Used to monitor the lines of code actually executed when test code is run.
- This generates a percentage of coverage – useful for finding untested logic pathways

Performing a Unit Test

testprimes.py (our unit test for is_prime_number)

```python
import unittest

#unit test should be testing one function
from primenumber import is_prime_number

class PrimeUnitTest(unittest.TestCase):
    """Tests for primes.py module"""

    def test_return_type(self):
        """should return a Boolean"""
        self.assertIsInstance(is_prime_number(10), bool)

    def test_is_eleven_prime(self):
        """is eleven (11) revealed to be a prime?"""
        self.assertTrue(is_prime_number(11))

    def test_special_one_is_prime(self):
        """is special case (1) revealed to be a prime? It should
        not be as it only has one positive divisor"""
        self.assertFalse(is_prime_number(1))

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

primenumber.py (has functions to test)

```python
def is_prime_number(value):
    """Check to see if given value is prime."""
    if value > 1:
        for number in range(2, value):
            result = value % number
            if  result == 0:
                return False
        return True
    return False

def get_next_prime(value):
    """Get next prime available, larger than value."""
    index = value
    while True:
        index += 1
        if is_prime_number(index):
            return index
```

```
test_is_eleven_prime
(__main__.PrimeUnitTest)
is eleven (11) revealed to be a
prime? ... ok
test_return_type
```

Performing a coverage test

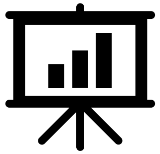
```python
import trace

#create a trace object
my_trace = trace.Trace(trace=0, count=1, timing=True)

# you'll see a lot of screen output with trace enabled.
my_trace.run('import primerun')

#this may take a while, especially if trace output is enabled...
results = my_trace.results()
results.write_results(show_missing=True, summary=True, coverdir=".")
```

primenumber.py (has functions to test)

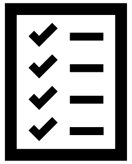
```python
def is_prime_number(value):
    """Check to see if given value is prime."""
    if value > 1:
        for number in range(2, value):
            result = value % number
            if  result == 0:
                return False
        return True
    return False

def get_next_prime(value):
    """Get next prime available, larger than value."""
    index = value
    while True:
        index += 1
        if is_prime_number(index):
            return index
```

primerun.py (our short test program)

```python
#test code for use by trace module
from primenumber import is_prime_number

for counter in range(1, 20):
    if is_prime_number(counter):
        print(f"{counter} is prime")
    else:
        print(f"{counter} is not prime")
```



Summary statistics



.cover files

Performing a coverage test

.cover files

```
>>>>>> def is_prime_number(value):
           """Check to see if given value is prime."""
   19:     if value > 1:
   81:         for number in range(2, value):
   73:             result = value % number
   73:             if  result == 0:
   10:                 return False
    8:         return True
           else:
    1:         return False

>>>>>> def get_next_prime(value):
           """Get next prime available, larger than value."""
>>>>>>     index = value
>>>>>>     while True:
>>>>>>         index += 1
>>>>>>         if is_prime_number(index):
>>>>>>             return index
```

Summary statistics

```
lines   cov%   module    (path)
   14    50%   primenumber   (C:/Users/Me/AppData/Local/Programs/Python/Python37-32\primenumber.py)
    5   100%   primerun   (C:/Users/Me/AppData/Local/Programs/Python/Python37-32\primerun.py)
  439    14%   rpc   (C:\Users\Me\AppData\Local\Programs\Python\Python37-32\lib\idlelib\rpc.py)
  364     0%   run   (C:\Users\Me\AppData\Local\Programs\Python\Python37-32\lib\idlelib\run.py)
  597     2%   threading   (C:\Users\Me\AppData\Local\Programs\Python\Python37-32\lib\threading.py)
  450     0%   trace   (C:\Users\Me\AppData\Local\Programs\Python\Python37-32\lib\trace.py)
```

UNIT TESTS

EXERCISE 7

Structured text file formats

There are three types of text file data formats you are likely to encounter:

- Comma separated values
- Oldest and most portable data format
- Often the CSV format is not properly standardised, e.g. could be *tab separated values* but *still use* .csv extension!

.csv

- eXensible Mark up Language
- Will be familiar to anyone who has used HTML
- Popular format, especially for SOAP-based web APIs
- Can be vulnerable to exploits so use with caution!

.xml

- JavaScript Object Notation
- Considered lightweight and compact
- Commonly used in RESTful APIs
- Used in many NoSQL databases, e.g. MongoDB

.json

CSV – reading from

Opening a CSV file for reading in Python is remarkably simple:

```
"author","title","published"
"Kelly, L","Java",2006
"Windmill, D","Networking",2002
```

- Header (fieldname) row is optional
- Each row is a separate record, values are separated by commas (usually!) and non-numeric data is usually double-quoted
- DictReader essentially creates a dictionary entry of values for each row, using the header row labels as each key.

```
import csv

with open('books.csv','r') as csv_file:
    csv_reader = csv.DictReader(csv_file)
    for row in csv_reader:
        for field in row:
            print(field, ":", row[field])
```

```
author : Kelly, L
title : Java
published : 2006
author : Windmill, D
title : Networking
published : 2002
```

CSV – writing to

Creating a CSV file in Python is also straightforward:

```
import csv

#open the file for writing
with open('mynewfile.csv', 'w', newline='') as csvfile:
    #set up the writer and the csv settings
    mywriter = csv.writer(csvfile, delimiter=',',
                quotechar='"', quoting=csv.QUOTE_NONNUMERIC)

    #write the csv headings
    mywriter.writerow(['ip_address','firstname','lastname', 'qty'])

    #this could be in a loop of course, for more than 1 row
    mywriter.writerow(['192.168.1.30', 'Phil', 'Cat', 2])
```

```
"ip_address","firstname","lastname","qty"
"192.168.1.30","Phil","Cat",2
```

- A new file is created for writing – potentially overwriting an existing one.
- CSV writer method can have delimiter, quote character and quoting strategies set.
- Header rows should ideally be written (this aids future processing)
- Data rows can be written from any source, in this case a simple list.

XML – reading from

Reading XML files in Python can be tricky; the ElementTree module is the easiest option:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<books>
    <book>
        <author>Kelly, L</author>
        <title>Java</title>
        <published>2006</published>
    </book>
    <book>
        <author>Windmill, D</author>
        <title>Networking</title>
        <published>2002</published>
    </book>
</books>
```

```python
import xml.etree.ElementTree as ET

tree = ET.parse('books.xml')
books = tree.getroot()
for book in books:
    for value in book:
        print(value.tag, ':', value.text)
```

```
author : Kelly, L
title : Java
published : 2006
author : Windmill, D
title : Networking
published : 2002
```

JSON – reading from

Reading JSON files in Python is incredibly straightforward:


```
{
  "books": {
    "book": [
      {
        "author": "Kelly, L",
        "title": "Java",
        "published": "2006"
      },
      {
        "author": "Windmill, D",
        "title": "Networking",
        "published": "2002"
      }
    ]
  }
}
```


```python
import json

with open('books.json') as books_file:
    books = json.load(books_file)
    for book in books['books']['book']:
        print('author:', book['author'])
        print('title:', book['title'])
        print('published:', book['published'])
```

```
author : Kelly, L
title : Java
published : 2006
author : Windmill, D
title : Networking
published : 2002
```

Creating Binary Files

In Python, **bytes** is an immutable type of object which can stores a sequence of values from 0 to 255 (an 8-bit value). These byte objects can be sliced (using an index) but *cannot* be modified.

A binary file is used to write these bytes from RAM to a more permanent backing storage media.

Python strings (str) are Unicode but can be encoded as bytes. Bytes may be decoded to Unicode strings

```
#write bytes to a file

filename = "data.bin"
with open(filename, "wb") as bin_file:
    #bytes to the file
    bin_file.write("Hello, world!\n".encode('utf8'))
    #we can also get the bytes written...
    num_bytes = bin_file.write(b'100 2.45 Hello')
    print(f"Wrote {num_bytes} bytes to {filename}.")
```

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	48	65	6C	6C	6F	2C	20	77	6F	72	6C	64	21	0A	31	30	Hello, world!.10
00000010	30	20	32	2E	34	35	20	48	65	6C	6C	6F					0 2.45 Hello

Reading binary files

Example of binary files include executables, images, sound files, video etc.

If their underlying file structure is known, it is possible to write a Python routine to examine certain bytes to identify their ‘signature’ and therefore their file type, i.e. "is it a jpg image?"

For a jpg file, the **first 4**

```
from os import listdir
from os.path import isfile, join
import binascii
import pprint

#specify path to check
check_path = 'C:/Users/Me/Downloads/'

#get list of files
list_files = [join(check_path, f) for f in listdir(check_path)
if isfile(join(check_path, f))]

#identify the hexadecimal signatures of first 4 bytes
jpg_sig = (binascii.unhexlify(b'FFD8FFD8'), binascii.unhexlify(b'FFD8FFE0'),
           binascii.unhexlify(b'FFD8FFEE'), binascii.unhexlify(b'FFD8FFE1'))

#scan the files!
print(f"Scanning {len(list_files)} files...")
jpg_files = []
for filename in list_files:
    print(f"Checking file {filename}...")
    with open(filename, 'rb') as file_to_check:
        signature = file_to_check.read(4)
        if signature in jpg_sig:
            jpg_files.append(filename)

pp = pprint.PrettyPrinter(indent=4)
pp.pprint(jpg_files)
```

Pickling – serialising a data structure

The pickle module serialises ("pickles") and de-serialises a Python object hierarchy into a stream of bytes. The process is sometimes called "flattening" or "marshalling".

Things to remember:

- Encapsulates the data and the object structure.
- Not encrypted!
- Not secure against maliciously constructed

```python
import pickle

#create a simple dictionary
cats_dict = {"Jess": 7, "Frank": 12, "Ron": 8, "Phil": 4}

#create the pickle!
pickle_filename = "cats"
pickle_file = open(pickle_filename, "wb")
pickle.dump(cats_dict, pickle_file)
pickle_file.close()
```

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	80	03	7D	71	00	28	58	04	00	00	00	4A	65	73	73	71	€.}q.(X....Jessq
00000010	01	4B	07	58	05	00	00	00	46	72	61	6E	6B	71	02	4B	.K.X....Frankq.K
00000020	0C	58	03	00	00	00	52	6F	6E	71	03	4B	08	58	04	00	.X....Ronq.K.X..
00000030	00	00	50	68	69	6C	71	04	4B	04	75	2E					..Philq.K.u.

The resulting binary Pickle file contains data and the original Python structure (a dictionary).

Pickling – serialising a data structure

Unpickling a binary Pickle file is remarkedly simple.

All that is required is for you to specify the filename and use the "load" function.

In this example we'll print the imported object's data and its object type.

```
import pickle

#loading the pickle!
pickle_filename = "cats"
pickle_file = open(pickle_filename, "rb")
cats_dict = pickle.load(pickle_file)
pickle_file.close()

#checking the Pickle and its contents...
print(cats_dict)
print(type(cats_dict))
```

```
{'Jess': 7, 'Frank': 12, 'Ron': 8, 'Phil': 4}
<class 'dict'>
```

The previous pickled dictionary is reconstructed perfectly.

Note:

Compression – creating a new file

Python supports a number of different compression and archiving formats including:

- **bz2** (bzip2)
- **gzip** (GNU xzip files)
- **lzma** (alogritm)
- **tarfile** – Tape archives
- **zipfile** – ZIP archives
- **zlib** – GNU zlib compression

In addition the **shutil** module provides a number

```
import gzip
import os

#set the compressed archive filename
arc_filename = 'mytext.txt.gz'
arc_file = gzip.open(arc_filename, 'wb')
try:
    #zip this content...
    arc_file.write(b'Hello, world!')
finally:
    arc_file.close()
    print(f"{arc_filename} contains ", end="")
    print(f"{os.stat(arc_filename).st_size} compressed bytes")
```

Example showing gzip module being used to create a GNU zip file from a sequence of bytes

Compression – zipping an existing CSV

Python's **shutil** module provides a number of high-level operations that include compression.

This example demonstrates the gzip and shutil modules being used to compress an existing CSV file.

```
#one way of zipping your csv...
import gzip
import shutil

#tested in Microsoft Windows
with open('mynewfile.csv', 'rb') as infile:
    with gzip.open('mynewfile.gz', 'wb') as outfile:
        shutil.copyfileobj(infile, outfile)
```

Note that all files must be read and written as Binary files – we are processing these byte-by-byte.

Working with TAR files

- This example demonstrates the creation of a new tar file from an existing list of files.
- In addition, techniques are shown for listing the tar file contents (using two different techniques) and extracting the files in the tar to

```
import tarfile
import os

#folders used in the tar process (create it and copy sample files into the first one)
source_dir = "./data"
destination_dir = "./copy"

#tar filename
newarchive = "newarchive.tar"

#opens for gzip compressed writing; will auto close
with tarfile.open(newarchive, "w:gz") as tar:
    #add all files for this folder
    tar.add(source_dir, arcname=os.path.basename(source_dir))

#open existing tar file for reading; will auto close
with tarfile.open(newarchive, "r") as tar:
    #iterate through the filenames (including directories)
    for filename in tar.getnames():
        print(filename)

    #Linux "ls"-style listing
    tar.list()

    #extract everything to a new folder (optional param - files to extract)
    tar.extractall(destination_dir)
```

EXTENSION ACTIVITIES

EXERCISE 8

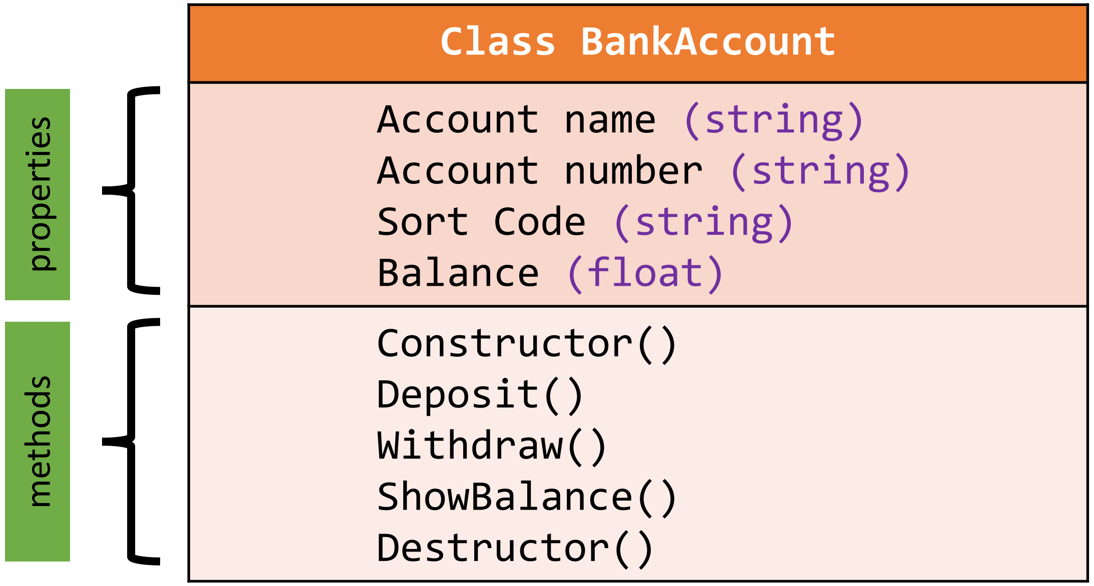
Classes

Object Oriented Programming (**OOP**) is a programming **paradigm** which approaches program design based on how different entities interact, e.g. a customer and their bank account.

A class is an **encapsulation** of an entity's:

- **properties** (its data)
- **methods** (its functions)

It forms a **template** of what the entity should **look like** and how it should **behave**.



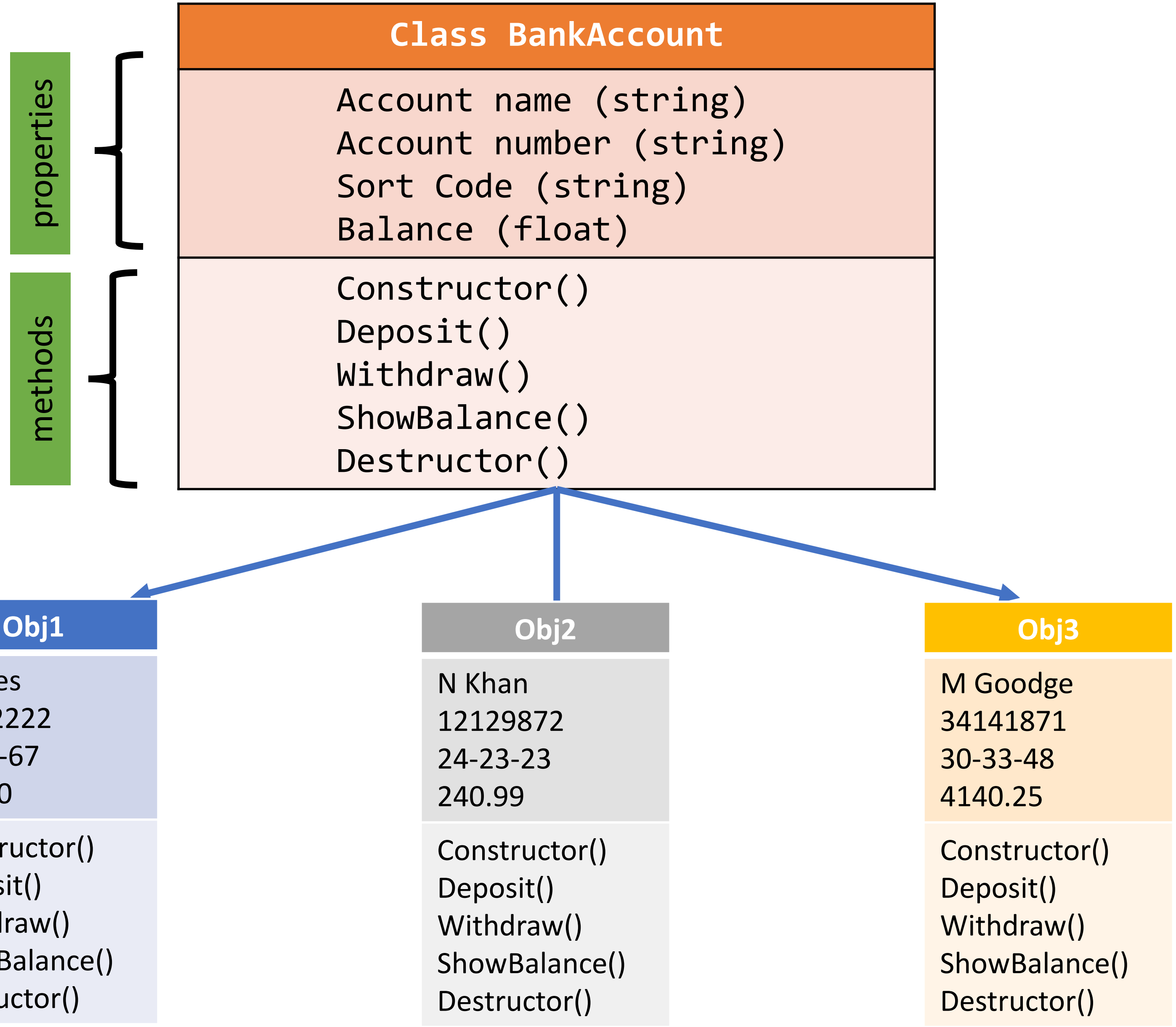
A simple **class schema** is often used to visualise a class, its properties and methods

Object

An **object** is created when a class is **instantiated**.

In other words, an object is a **concrete instance of a class**.

Although an object shares the **same methods** as other objects created from the **same class**, its **property**



Implementing a class

- A Python implementation of our basic BankAccount class.
- The __**init**__ "magic" method is our constructor, running automatically when an object is created.
- "@" (pie) syntax is used to create **decorators** which create properties we can use to set and get our "name mangled"

```
class BankAccount(object):

    num_accounts = 0
    def __init__(self, newbal):
        self.__balance = newbal
        BankAccount.num_accounts += 1

    def deposit(self, amount):
        self.__balance += amount

    def withdraw(self, amount):
        self.__balance -= amount

    def show_balance(self):
        print(f"Balance is {self.__balance:<8.2f}")

    @property
    def balance(self):
        return self.__balance

    @balance.setter
    def balance(self, newbal):
        self.__balance = newbal

    def __str__(self):
        return f"{self.__balance:<8.2f}:{BankAccount.num_accounts:<3}"
```

Instantiating and using an object

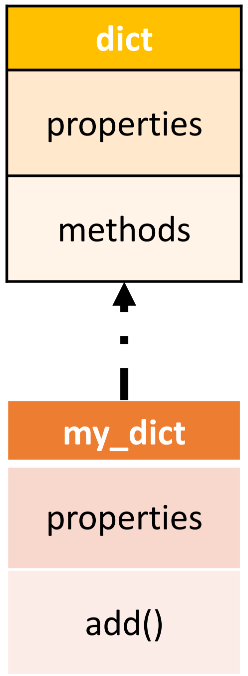
```
#create first object
phil = BankAccount(100)
phil.show_balance()

phil.deposit(50)
phil.show_balance()

#print(phil.__balance)
print(hasattr(phil, '__balance'))

#create another object
jess = BankAccount(200)
print(BankAccount.num_accounts)

#setting the balance via the decorator
phil.balance = 700
print("Phil's new balance is", phil.balance)

#test the __str__ method
print(phil)
```

```
Balance is 100.00
Balance is 150.00
False
2
Phil's new balance is 700
700.00  :2
```

Inheritance

- This example demonstrates the creation of a new class which **extends** the **dict** (dictionary) class to implement a custom "add" method.
- In very simple terms, our new "add" method

```
# create a bespoke dictionary class
class my_dict(dict):

    # new method to add a new key:value
    def add(self, key, value):
        self[key] = value


dict_obj = my_dict()

dict_obj.add(1, 'Phil')
dict_obj.add(2, 'Ron')

print(dict_obj)
```




```
{1: 'Phil', 2: 'Ron'}
```

Static Methods

- A **static method** is part of a class definition but is **bound to the class not** the object.
- A **function** in the context of a **class**.
- As there is no object, there is no object data to access

The preferred Python implementation of a **static method** uses the "@" (pie) decorator annotation:

```python
import re

class Postcode(object):
    """Validates and processes UK postcodes"""
    regex = r'^([a-z]{1,2})([0-9]{1,2}) ?([0-9])([a-z]{2})$'

    @staticmethod
    def is_valid(postcode: str)->bool:
        _matches = re.match(Postcode.regex, postcode, re.IGNORECASE)
        if _matches:
            return True
        return False
```

This static method can be called **without** reference to an object:

```python
#ad hoc test using no object
user_postcode = input("Enter postcode: ")
if Postcode.is_valid(user_postcode):
    print(f"Postcode '{user_postcode}' is valid.")
```

Class methods

- A **class method** is part of a class definition but is **bound to the class not** the object.
- As there is no object, there is no object data to access.
- This example demonstrates a practical, popular use which is to

The preferred Python implementation of a **class method** uses the "@" (pie) decorator annotation:

```
class Postcode(object):
    """Validates and processes UK postcodes"""
    regex = r'^([a-z]{1,2})([0-9]{1,2}) ?([0-9])([a-z]{2})$'

    @classmethod
    def from_facets(cls, *facets):
        _temp_postcode = ''.join(facets)
        return Postcode(_temp_postcode)
```

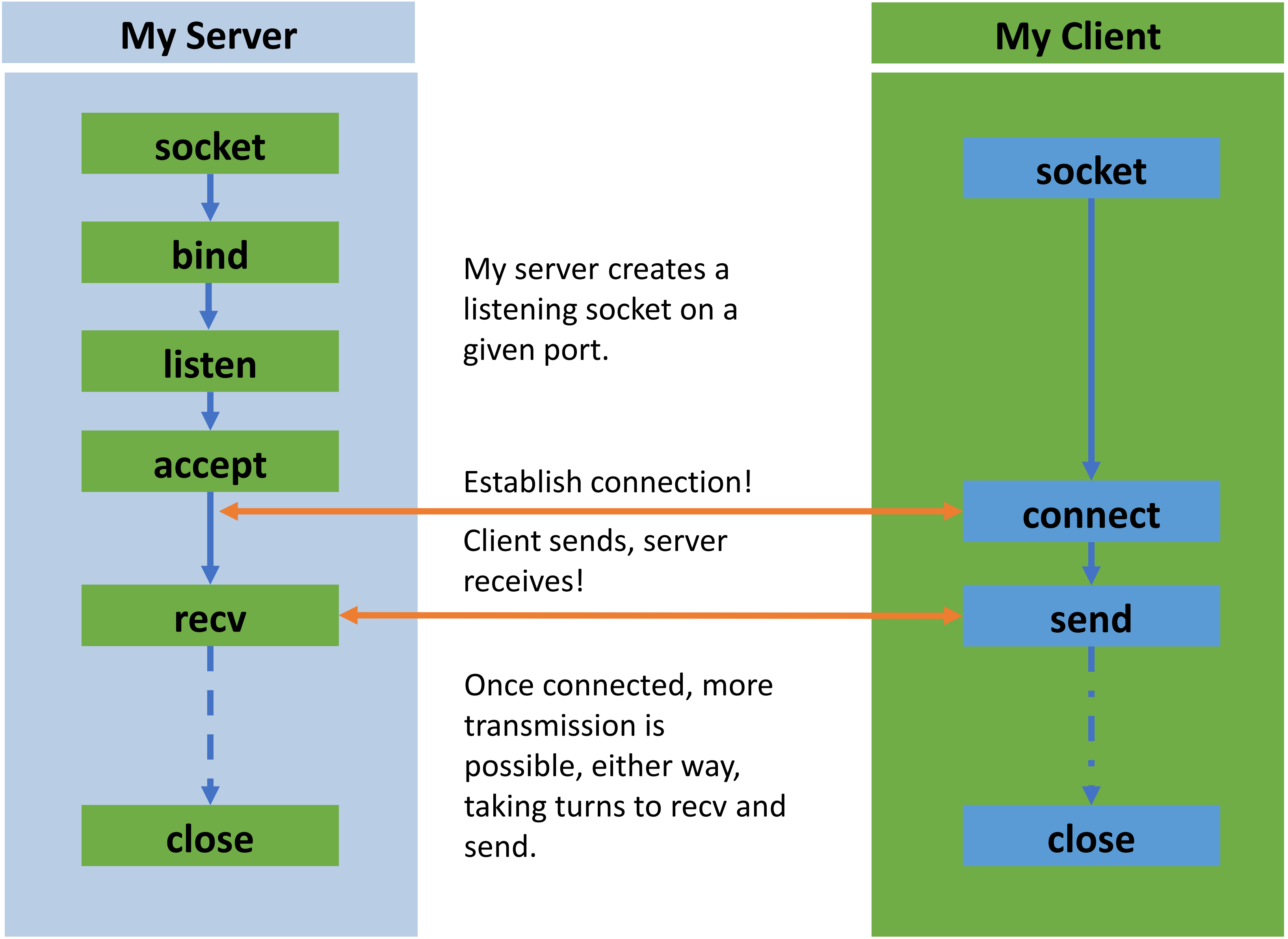
Compare and contrast the two different instantiations for a Postcode object:

```
#create an object using a string (constructor)
P1 = Postcode("gl13qn")

#create an object using facets (alternative "constructor")
P2 = Postcode.from_facets('gl', '1', '3', 'qn')
```

Connection-based communication

- This diagram shows the sequence of socket-based API calls and data flows for the TCP-based connection.

- Connection is made using a handshake that ensures both client and server



Simple socket-based connection

my_server.py

```
#server code
from socket import *
from collections import namedtuple

ipv4 = namedtuple("ipv4", ["ip_addr", "port"])

BACKLOG = 1
BUFFER_SIZE = 4096
FLAGS = 0

my_socket = socket(AF_INET, SOCK_STREAM)
ipv4_pair = ipv4('127.0.0.1', 1066)
my_socket.bind(ipv4_pair)
my_socket.listen(BACKLOG)

print("Waiting for connection on port", ipv4_pair.port)
new_socket_object, r_address = my_socket.accept()

print("Connection made from:", r_address)
received_data = new_socket_object.recv(BUFFER_SIZE, FLAGS)

print("Data received:")
decoded_received_data = received_data.decode("utf-8")
print(decoded_received_data)

print("Closing connection")
my_socket.close()
```

my_client.py

```
#client code
from socket import *
from collections import namedtuple

FLAGS = 0

ipv4 = namedtuple("ipv4", ["ip_addr", "port"])

my_socket = socket(AF_INET, SOCK_STREAM)
ipv4_pair = ipv4('127.0.0.1', 1066)

my_socket.connect(ipv4_pair)

print("Sending message")
msg = b"Mr. Watson--come here--I want to see you."

my_socket.send(msg, FLAGS)

print("Closing connection")
my_socket.close()
```

Parsing command line arguments

- This example demonstrates the use of **argparse** – a module which makes writing user-friendly command-line interfaces **very simple!**
- This interface accepts username and password arguments at the command line

```
from collections import ChainMap
import os, argparse

#create our defaults
defaults = {'user':'guest', 'email':'mark.fishpool@qa.com'}

#parse command line arguments
parser = argparse.ArgumentParser()
parser.add_argument('-u', '--user')
parser.add_argument('-e', '--email')
namespace = parser.parse_args()

#create a dictionary of valid command line arguments
command_line_args = {k:v for k, v in vars(namespace).items() if v}

#create a chainmap of these dictionaries (in preferential order)
combined = ChainMap(command_line_args, os.environ, defaults)

#find the *first* matching instances (in preferred order)
print(combined['user'])
print(combined['email'])
```

```
python chainmapex.py -u frank
frank
mark.fishpool@qa.com
```

Executing commands via the OS shell

- It is possible to run Operating System internal and external commands using the subprocess module, although other techniques can be used.
- In this example, the Windows version of the code opens a shell and pipes the stdout response back into

```python
import subprocess

#Linux; long listing format
cmd_result = subprocess.run(['ls', '-l'], stdout=subprocess.PIPE)

#Windows; with generated short name (8.3) equivalents
cmd_result = subprocess.run(['dir', '/X'], shell=True, stdout=subprocess.PIPE)

print(cmd_result.stdout.decode("utf-8"))
```

Extract of STDOUT with no decode... (bytes)

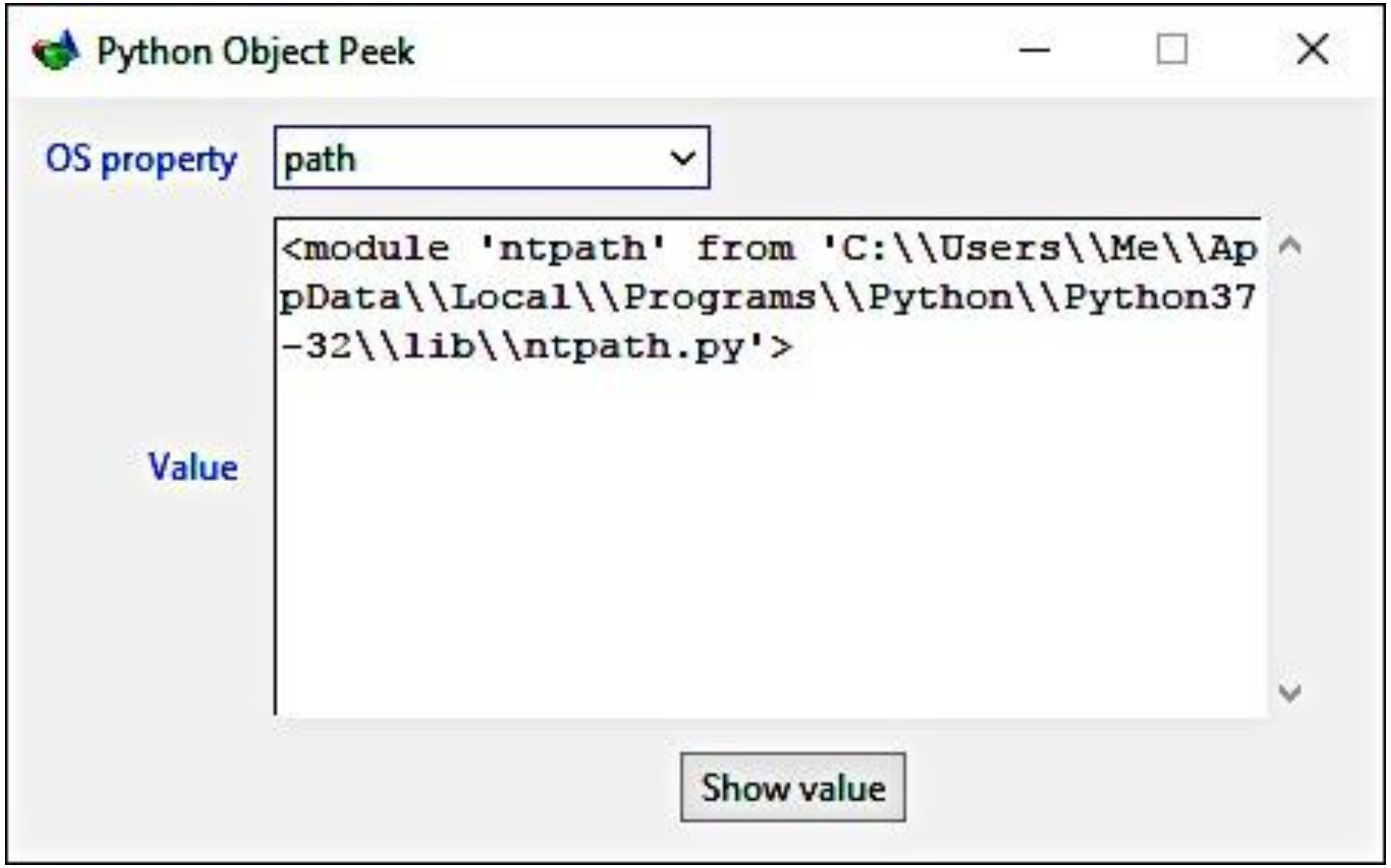
```
21/05/2019  19:52    <DIR>                       .\r\n21/05/2019  19:52
<DIR>                          ..\r\n19/12/2018  22:57               849
ABCEXA~1.PY  abcexample.py\r\n19/12/2018  20:37               714 ANNOTA~1.PY
annotations.py\r\n23/04/2019  22:45               613
arpcheck.py\r\n16/03/2019  14:22               288 AVERAG~1.PY
averagesales.py\r\n
```

Extract of STDOUT With decode... (string)

```
21/05/2019  19:52    <DIR>                       .
21/05/2019  19:52    <DIR>                       ..
19/12/2018  22:57               849 ABCEXA~1.PY  abcexample.py
19/12/2018  20:37               714 ANNOTA~1.PY  annotations.py
23/04/2019  22:45               613              arpcheck.py
16/03/2019  14:22               288 AVERAG~1.PY  averagesales.py
```

Building a GUI-based Python script

- Building a GUI application in Python typically involves the use of the Tkinter module and it's TTK sub-module to provide access to a number of different GUI widgets.
- You are going to build an application that provides the user with a dynamic list of OS module properties which can be

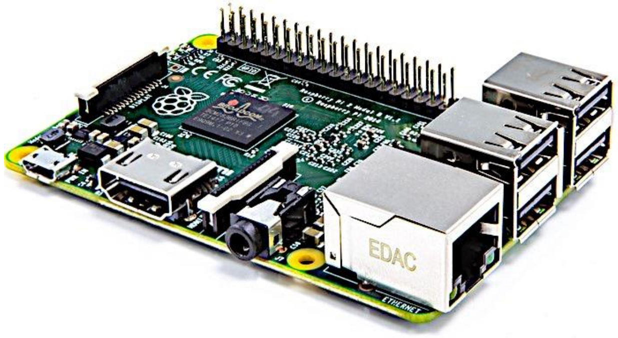


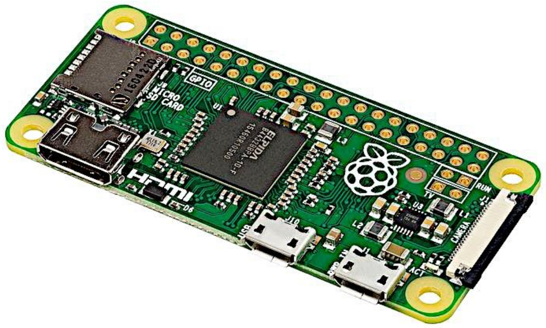
CREATING A GUI

EXERCISE 9 AND 10

Different Raspberry Pi Models

- Original model launched in 2012; as of 2018, over 18 million sold
- Known as a "single board" computer (SBC)
- Many different Pi Models are available, with some models now technically discontinued
- Features and capabilities vary from model to model, e.g. CPU speed, available connectivity etc
- Each model can be further modified, e.g. overclocking the CPU (making them run faster

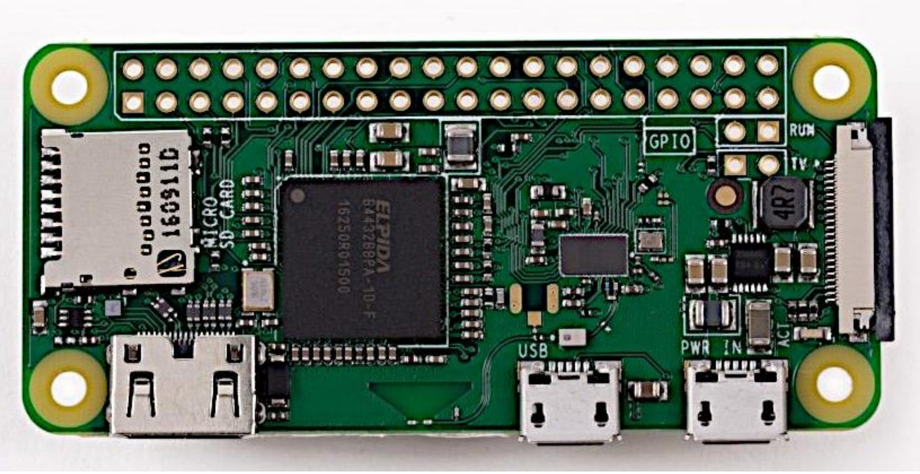




Contrasting Raspberry Pi 2B and Raspberry Pi Zero form factors

Technical Specification, e.g. Raspberry Pi Zero

- 1GHz, 32-bit single-core ARM **C**entral **P**rocessing **U**nit (**CPU**)
- 512MB **R**andom **A**ccess **M**emory (RAM)
- Mini **H**igh-**D**efinition **M**edia **I**nterface (**HDMI**) and **U**niversal **S**erial **B**us (**USB**) **O**n-**T**he-**G**o (**OTG**) ports
- Micro USB power
- **H**ardware **A**ttached on **T**op (**HAT**)-compatible 40-pin **G**eneral-**p**urpose **I**nput/**O**utput header



Headerless
Raspberry Pi Zero W

Interfaces

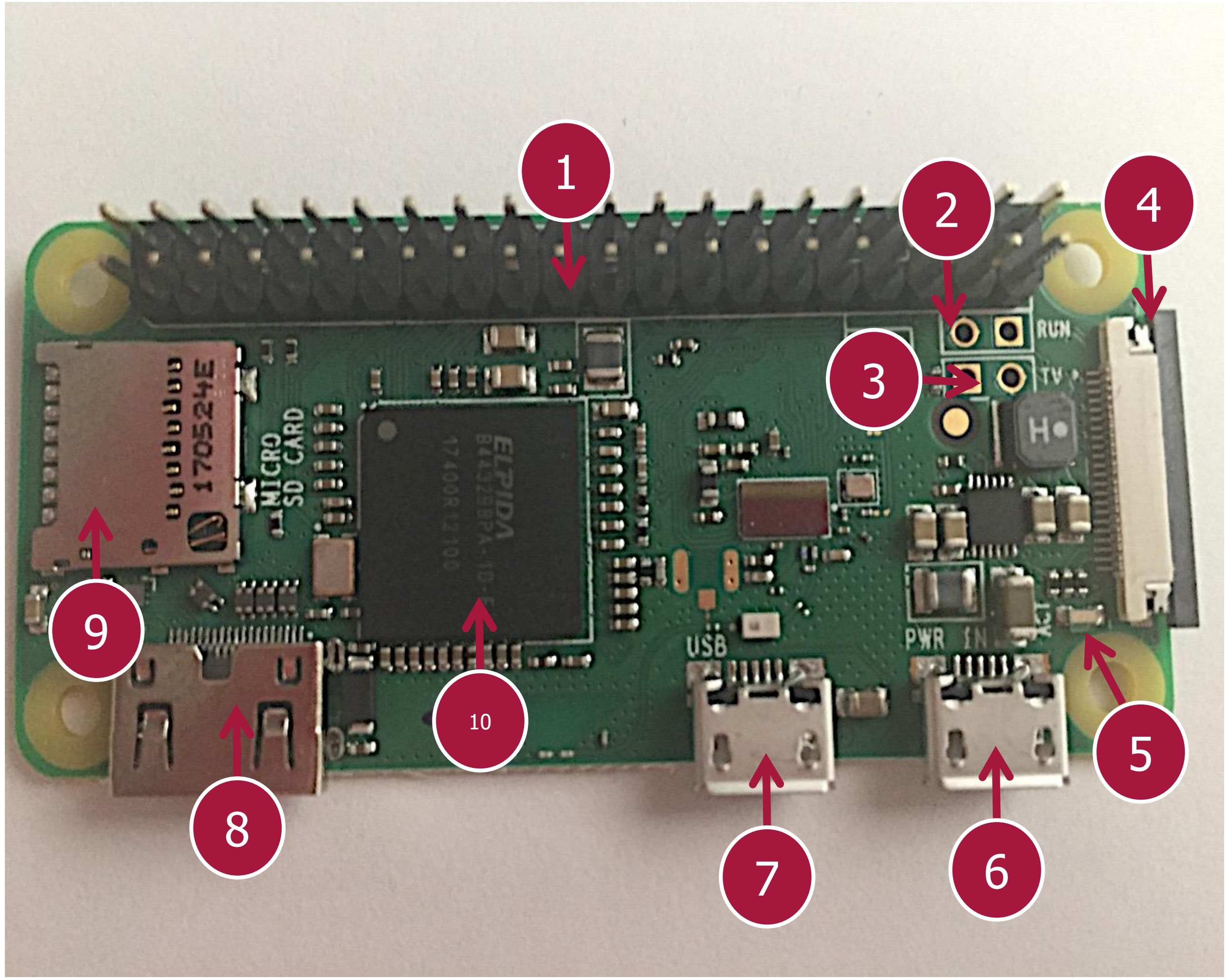
Standard connections include:

- Mini **HDMI** for display screen output; this needs an adaptor
- **USB** “on-the-go” Micro-B for keyboard, mice and USB hubs; this typically needs an adaptor
- **CSI** for connecting digital cameras
- **GPIO header** is used to connect and control external devices; the PI comes in headerless form as standard
- A Pi Zero W**H** has pre-soldered

USB OTG adaptor cable

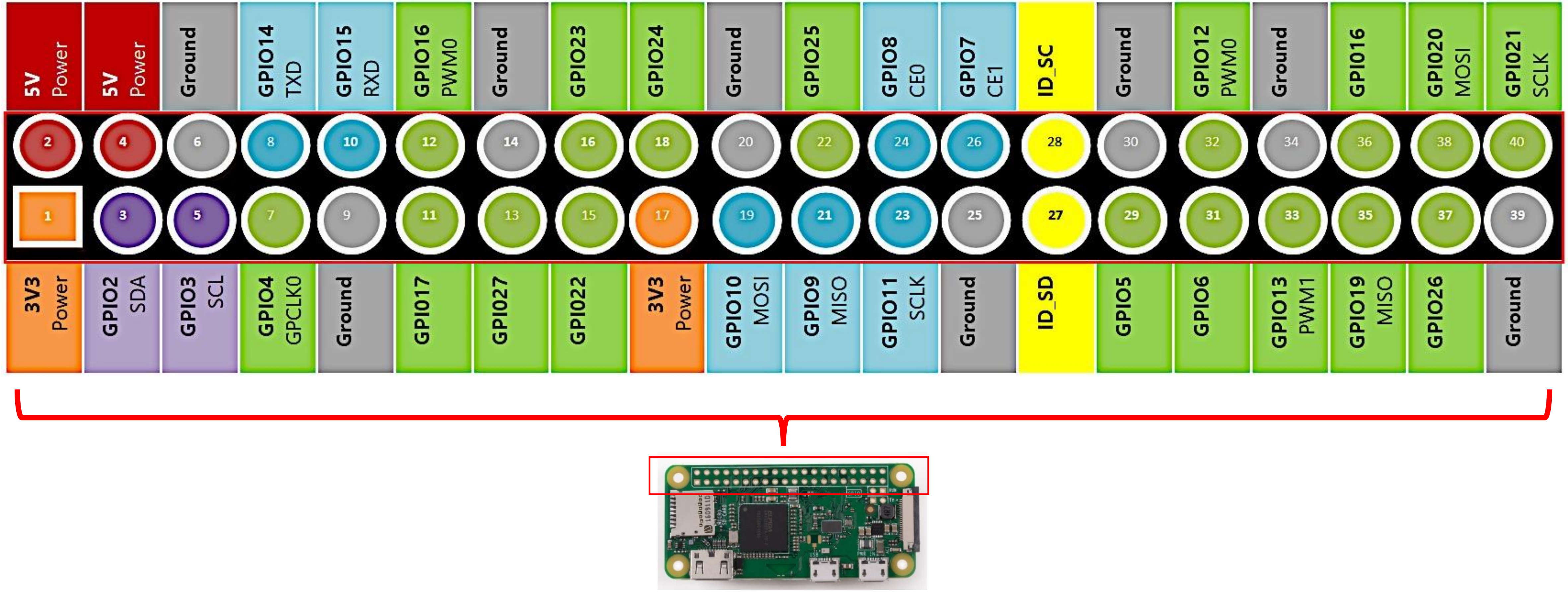
Standard and mini HDMI connectors

Raspberry Pi: a typical form factor



1	40 pin (2 x 20) **General-purpose input/output** (**GPIO**) header
2	**Run** Can be used to perform a hard reset or restart the Pi Zero W after shutdown.
3	**TV** Support for NTSC/PAL composite TV output via soldered connector and RCA plug.
4	**Camera Serial Interface** (**CSI**) connector
5	**ACT (LED also known as led0)** Typically shows "disk" (e.g. MicroSD card read/write) activity but it is programmable.
6	**Micro USB (PWR IN)** Note. Pi Zero W draws its power through this USB connection.
7	**Micro USB or USB "on-the-go" (OTG)** for keyboard, mouse, USB hub etc.
8	**Mini HDMI out** Video output but includes HDMI audio
9	**MicroSD Card slot** A bootable Operating System such as Raspbian is inserted here.
10	**Broadcom BCM2835** Single-core 1Ghz ARM CPU (with integrated GPU) 512MB SDRAM

Raspberry Pi Zero W GPIO pinout



Pi specific commands

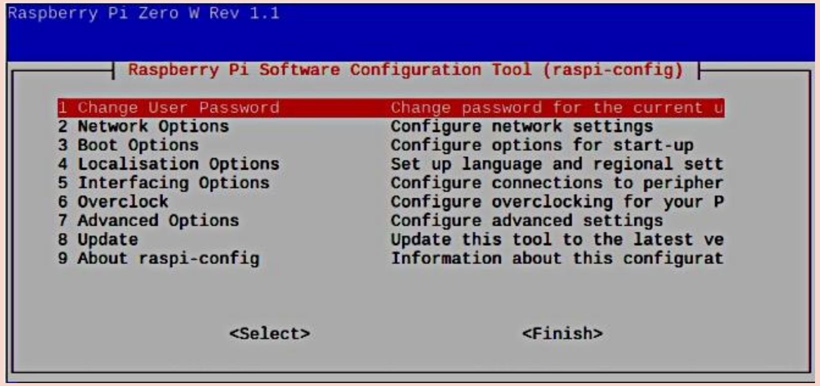
Command	Purpose	Example
pinout	A very useful utility for querying the Raspberry Pi's GPIO pin-out information. Running pinout will result in a graphical representation of the single board computer and its GPIO pin-out. Each pin is identified by its BCOM designation and pin number.	

Pi specific commands

<table>
<tr><th>Command</th><th>Purpose</th><th>Example</th></tr>
<tr>
<td>sudo raspi-config</td>
<td>A useful utility for configuring the default options of a Raspberry Pi.

Sudo is required because its is owned by the superuser (Uid 0; root)

Options allow the user to modify:

• Password
• Network options
• Boot options
• Localisation, e.g. Time zone, keyboard layout etc
• Interface options, e.g. Enabling Secure Shell (SSH) and Virtual Networking Computing (VNC) services for remote connection
• Advanced options, e.g. Video and audio
• Overclocking
• Updating the tool itself</td>
<td>
</td>
</tr>
</table>

Editing

In Raspbian, the editing of common **A**merican **S**tandard **C**ode [for] **I**nformation **I**nterchange (**ASCII**) text files, e.g. configuration files and scripts, can be performed using **Vi IM**proved (**vim**) and **N**ano's **A**

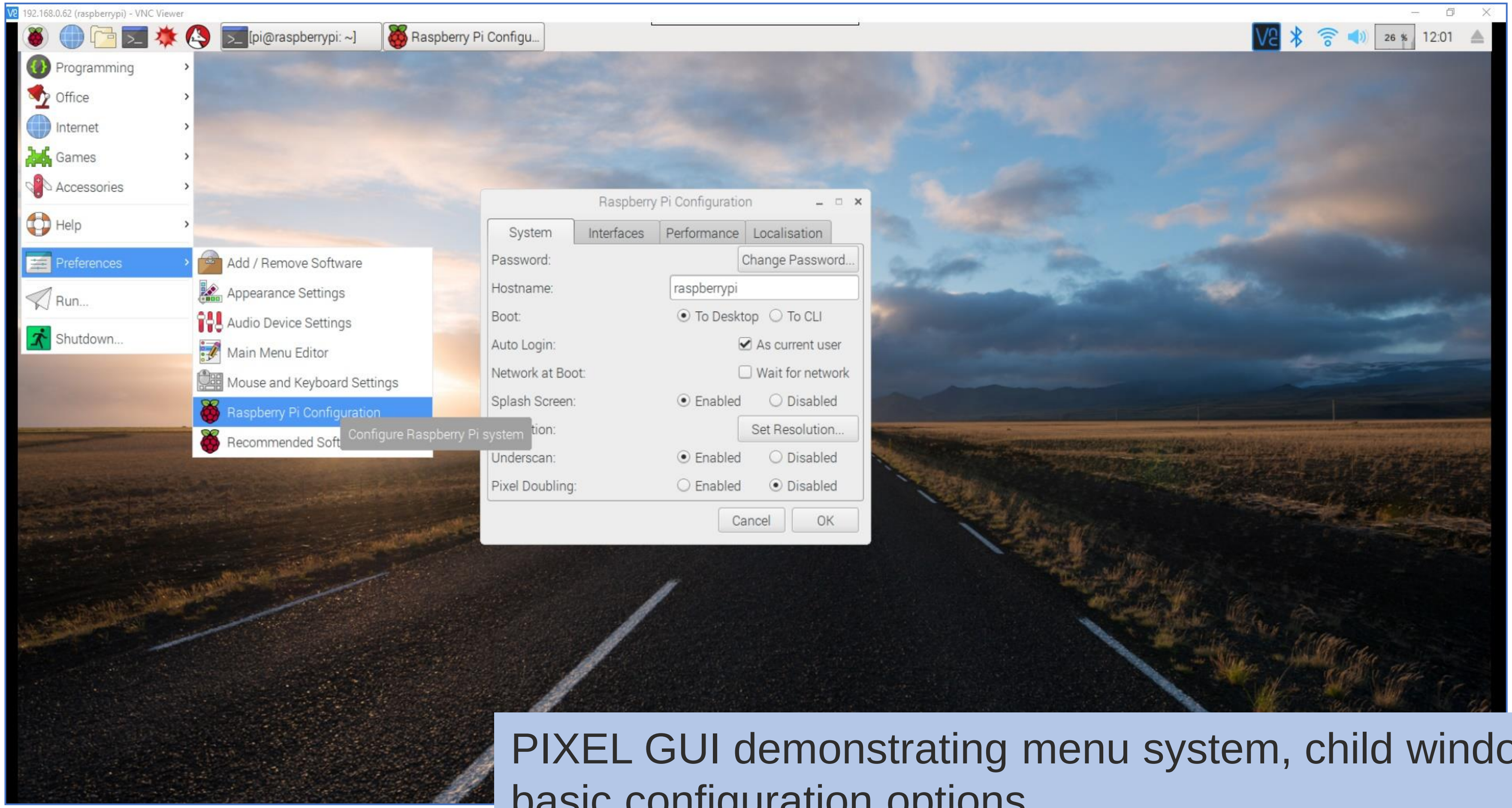
Vim	Nano

Overview of PIXEL

- Modified version of Lightweight X11 Desktop Environment (LXDE)
- Primarily written in C
- Deliberately lean memory usage compared to other Window managers such as Gnome or KDE
- Installed with programming tools (Python, Scratch, Mathematica), an office productivity suite (LibreOffice),internet applications (Chromium), games (Minecraft Pi) and system accessories.
- RealVNC's VNC server has been ported to the Pi and integrated into PIXEL, permitting easy remote client connections from other OS

- https://www.raspberrypi.org/blog/introducing-pixel/

Overview of PIXEL

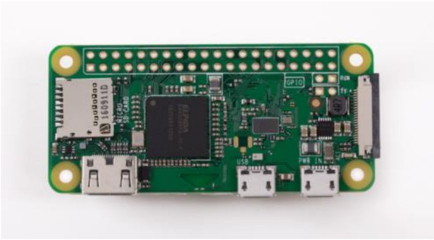


PIXEL GUI demonstrating menu system, child windows and basic configuration options

About VNC

- VNC is a graphical desktop sharing system using a protocol called **R**emote **F**rame **B**uffer (**RFB**)
- The target computer to control must have a **VNC server** ("VNC Connect") installed and be running in "service" mode
- Controlling computer must have a **VNC client** ("VNC Viewer") running
- The target machine must be accessible via a network connection, e.g. **L**ocal **A**rea

1) Rasberry Pi Zero running VNC Connect under Raspbian



2) Wireless Access Point (WAP)

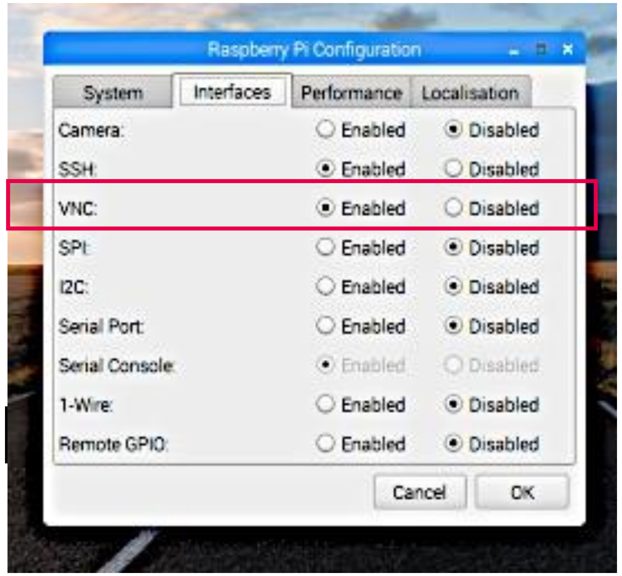
3) PC (or Mac) running VNC Viewer

Raspberry Pi and VNC

- Both VNC Connect and VNC Viewer **are** included with Raspbian **by default.**
- I

```
sudo apt-get update
sudo apt-get install realvnc-vnc-server realvnc-vnc-viewer
```

lly:



- For security, VNC Server is typically **not** enabl
- Enabling via Shell

  Open a Shell via Terminal

  Type:

  ```
  sudo raspbi-config
  ```

  Navigate to Interfacing Options

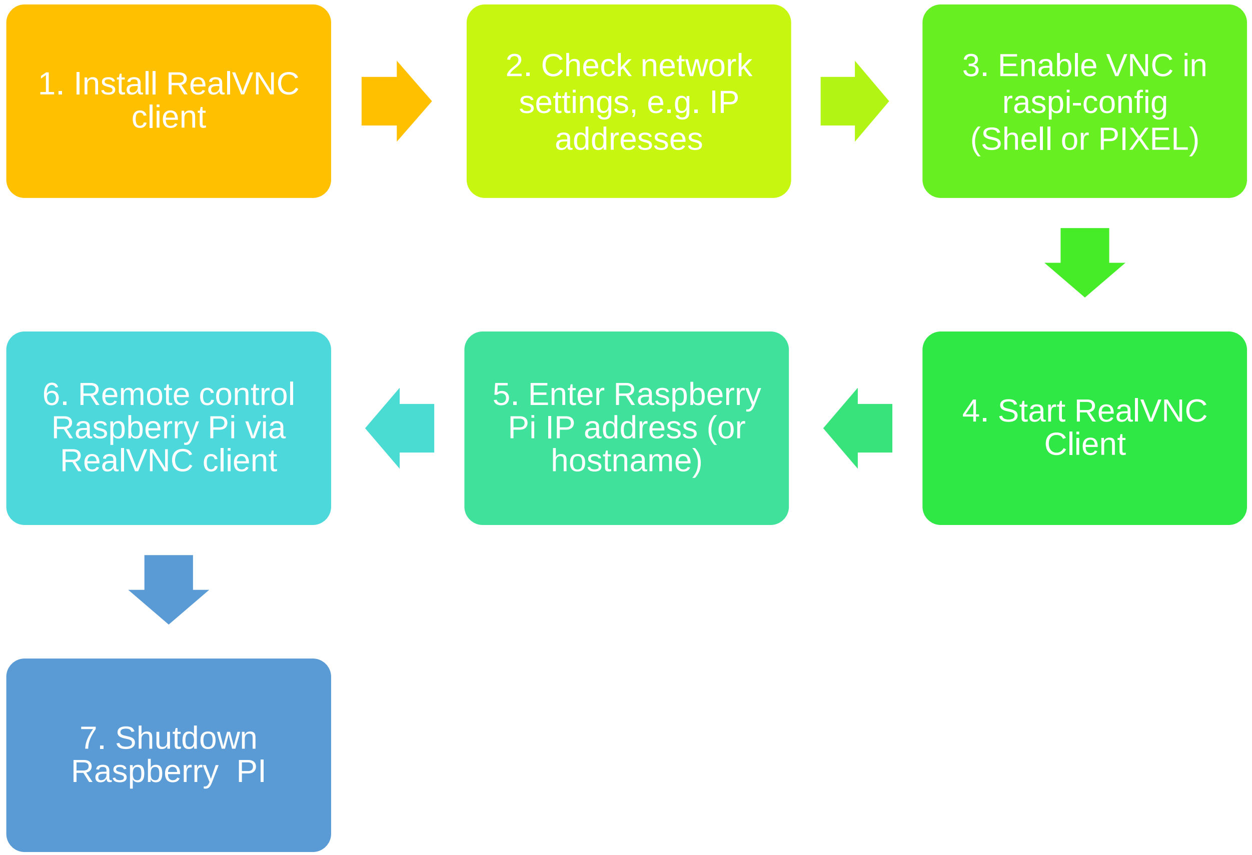
  Select VNC->**Yes**

Enabling via PIXEL desktop:

Menu->Preferences->Raspberry Pi Configuration -> Interfaces

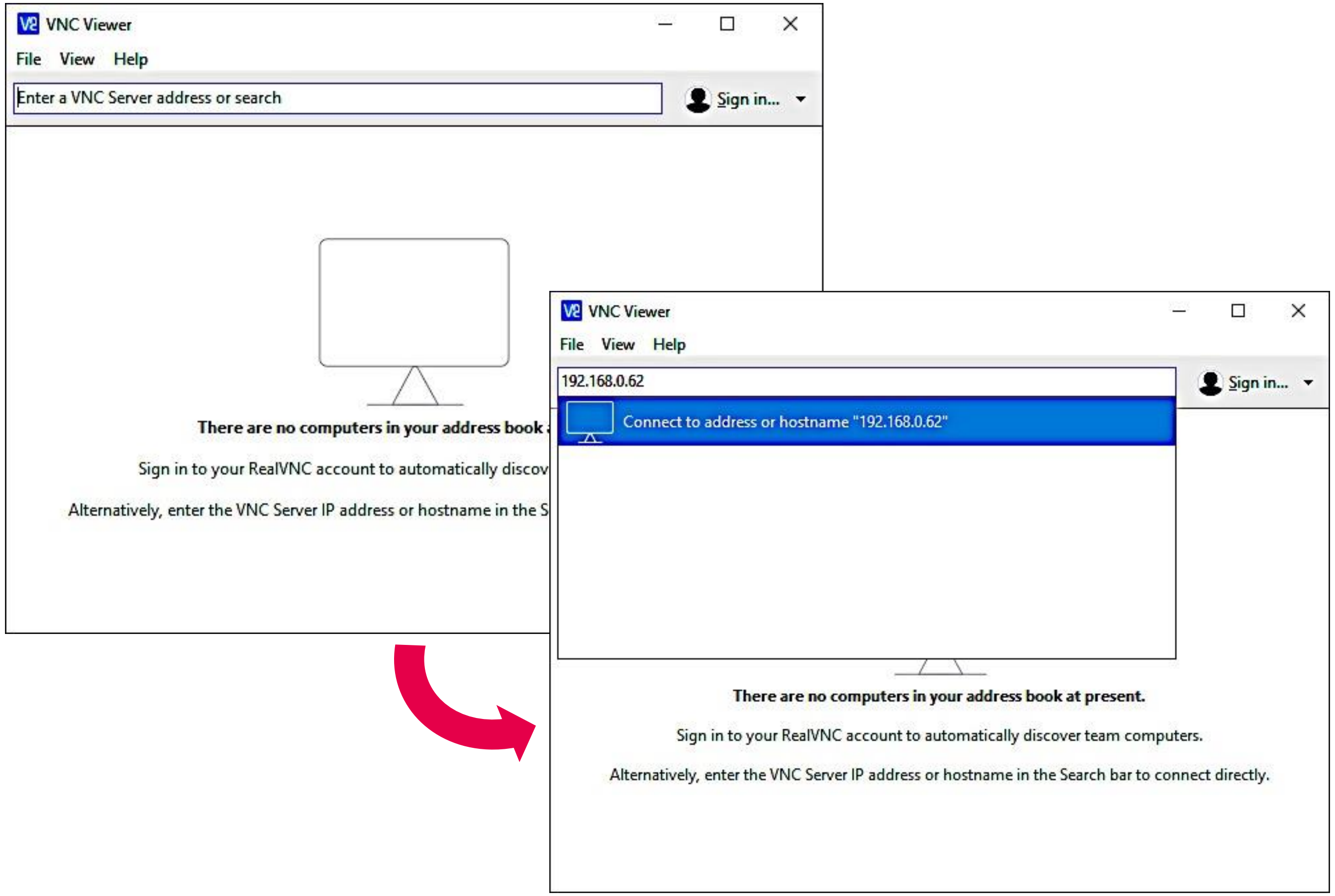
Connecting to Raspberry Pi via VNC - overview

- A Raspberry Pi which is running without screen, keyboard or mouse is said to be "**headless**".
- Control of a headless Raspberry Pi can be performed from a PC, Mac or mobile device.
- Shutting down the Raspberry Pi will **automatically** disconnect the VNC session.



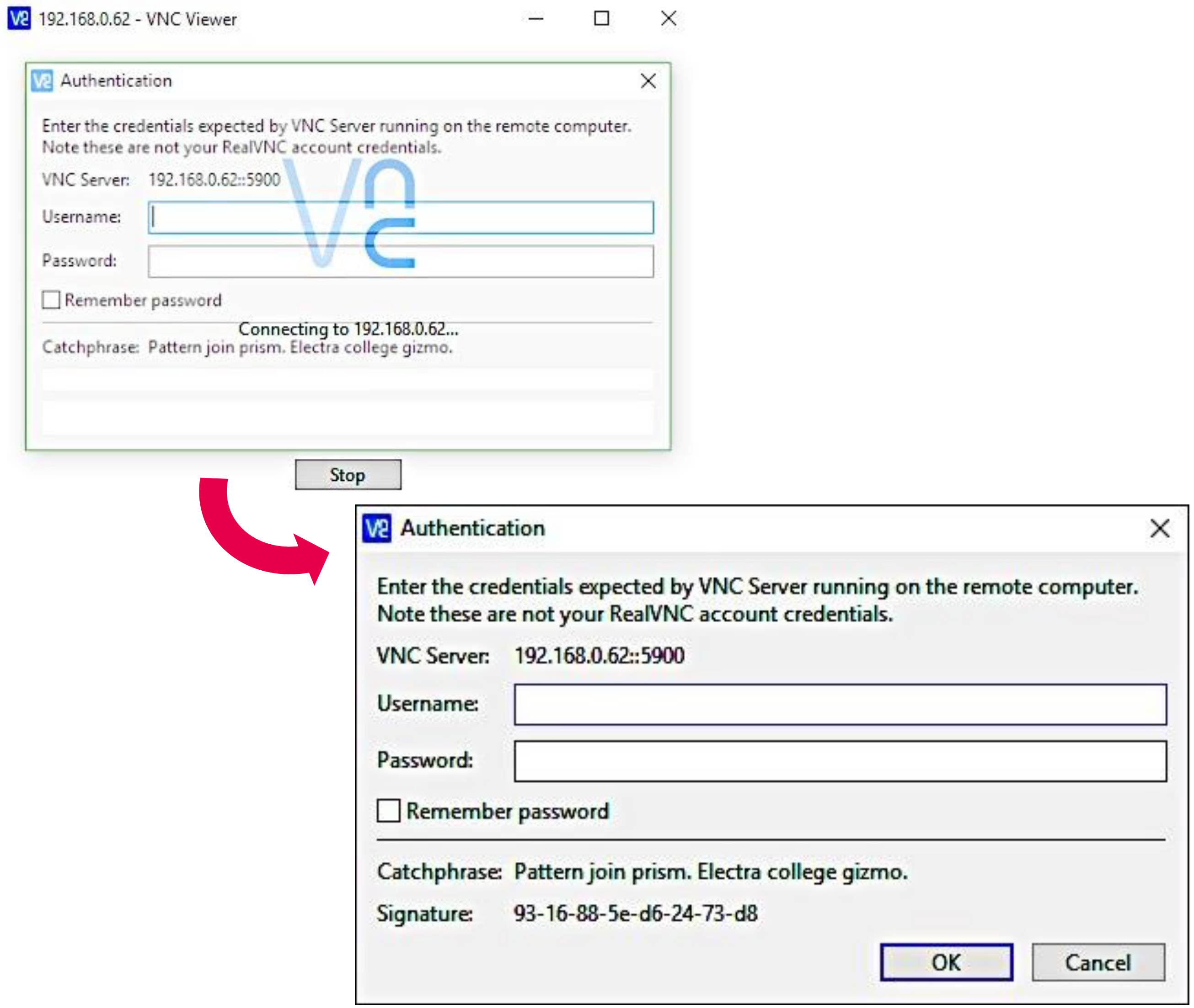
Connecting to Raspberry Pi via VNC

- The VNC Viewer client will store a target machine's IP address (or hostname) in its **address book** if it has been previously used.
- Either **select** the available IP address from the address book **or enter** the **IP address** (or **hostname**) of the target machine



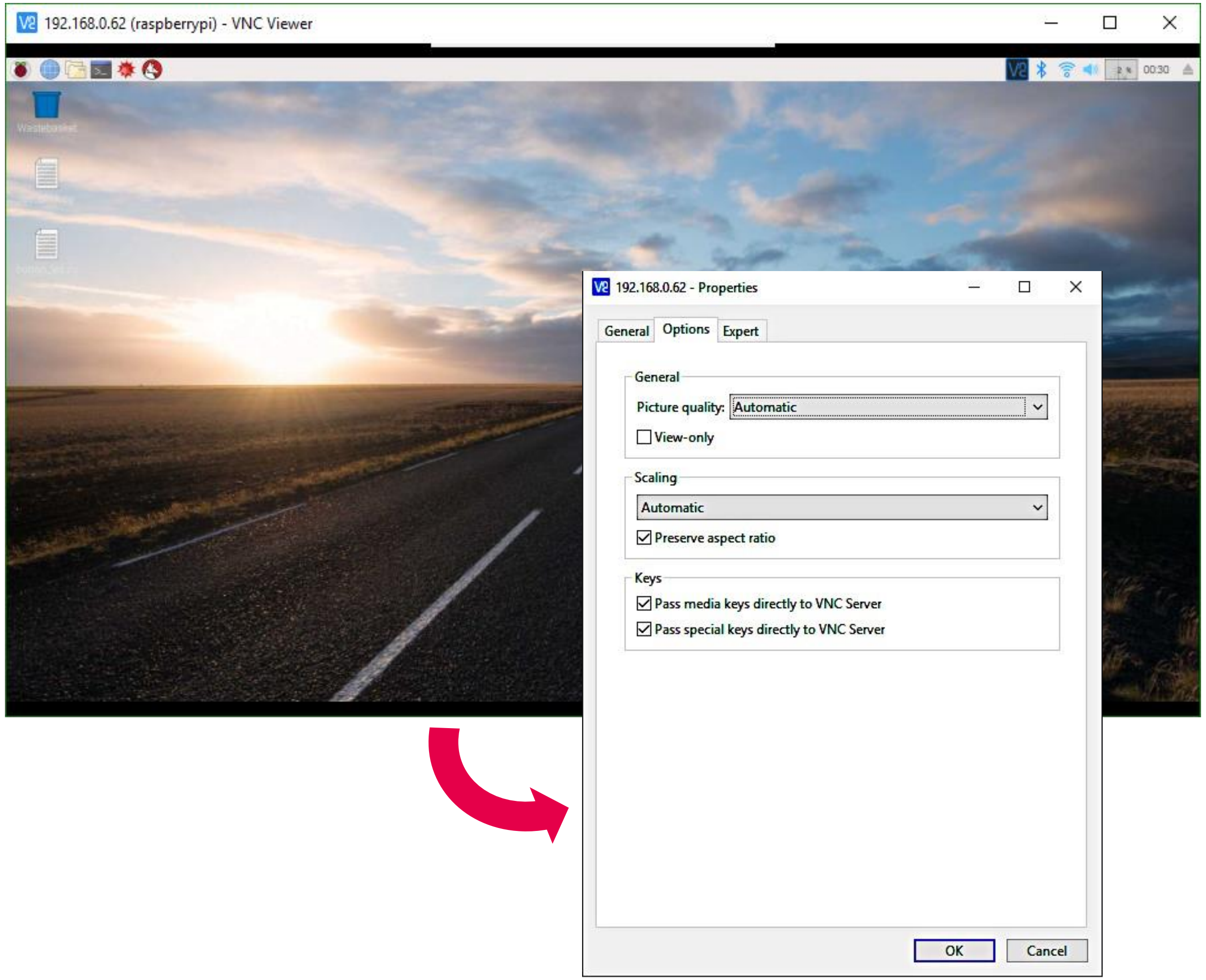
Connecting to Raspberry Pi via VNC

- The VNC Viewer client will start to connect to the remote server
- The default connection port is **5900**
- After a brief delay you should be asked to confirm your credentials on the target computer.
- Network bandwidth will affect image quality and responsiveness
- Unsuccessful attempts will result in an error



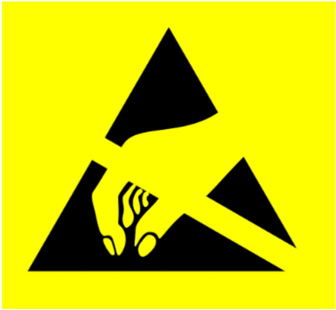
Connecting to Raspberry Pi via VNC

- The VNC server sends small rectangles of its screen framebuffer to the client
- This is updated when rectangle content changes, e.g. Moving a cursor, moving or opening a new window etc
- VNC options can be accessed which alter its server and client behaviour, including

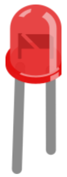
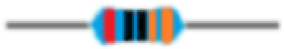
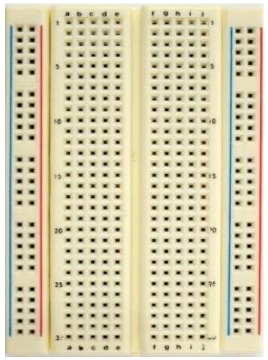


Safety Tips: Things to Remember!

- Circuits should **not** be connected to the Pi **while** it is powered on; connect the circuit to the Pi GPIO while it is safely **shutdown**
- Components can be **easily damaged**, sometimes **irrevocably** by incorrect wiring
- Although most electronic circuits are **relatively robust** in terms of **e**lectro**s**tatic **d**ischarge (**ESD**), **sensible precautions** while working on the Raspberry Pi should be taken
- Some components can **melt** or **explode** even at relatively **small**



Electronic components – visual overview

Light Emitting Diode (LED)	Resistor	Male-to-female jumpers	Male-to-male jumpers	Tilt sensor	Disc thermistor
					
Breadboard	Push switch	Light Dependent Resistor (LDR)	Electrolytic Capacitor	Micro servo	Potentiometer (variable resistor)
					

Electronic component - descriptions

Component	Purpose
Light Emitting Diode (LED)	A type of diode that emits light when activated, e.g. red LED
Resistor	Implements electrical resistance in a circuit, reducing current flow
Male-to-female jumper	Used to connect Raspberry Pi Zero GPIO pins to the breadboard
Male-to-male jumper	Used for patching between different breadboard locations
Tilt sensor	Produces an electrical signal that varies with angular movement
Disc thermistor	A resistor whose properties are reduced by heating
Breadboard	A solder-less device used to quickly assemble and test circuits before finalizing a design
Push switch	A push-to-make switch which allows electricity to flow when pressed
Light Dependent Resistor (LDR)	A resistor whose properties changes with the light intensity falling on it; also know as a photo resistor or photo-conductive cell
Micro servo	A device which can rotate approximately 180 degrees
Potentiometer (variable resistor)	A variable resistor whose properties changes based on a sliding or rotating contact

TIMED LED BLINKING

EXERCISE 11 AND 12

EXTENSION ACTIVITIES

EXERCISE 13