# #

text

Examples:

```
# This is a comment
```

Hash symbol: A comment.

Comments are ignored by the computer and discarded when a program is *translated* into *machine code*. They are used by programmers to explain the purpose of sections of code. This is helpful when you return to a program after a period of time, or when you work in teams.

Comments are typically used:

- At the beginning of a program to explain its purpose.
- Before each subprogram: def
- Before each selection statement: if, else, match, case.
- Before each iteration: for, while.
- To explain difficult to comprehend code.
- To remind the author why unusual approaches have been taken.

Associated keywords: `def, if, else, match, case, while`

ABS

variable = abs(*parameter*)

Examples:

```
x = abs(-5)
```

```
x = abs(y)
```

Function: Returns an absolute value of the parameter.

The absolute value is a positive value. For example, the absolute value of -6 is 6. This function is useful for turning a negative number into a positive number.

It can also be useful to swap a number from positive to negative or negative to positive. This can be achieved with `x = -x`.

APPEND

list.append(*parameter*)

Examples:

```
mylist.append("Dave")
```

Method: Adds the parameter to a list.

The example above adds the string "Dave" to the list called mylist. The parameter can be a value, a variable or a list. Append means to add to the end, so the new item is always added to the end of the list and becomes the last item.

You can overwrite existing elements of a list by referring to their index. E.g. `list[2] = "Dave"` will replace what is currently stored in index 2.

Associated keywords: `insert, pop, remove`

CHR

variable = chr(*integer*)

Examples:

```
character = chr(65)
```

Function: Returns the character from a denary ASCII code.

All characters, including letters, numbers and symbols stored in strings are actually stored in binary by the computer. The American Standard Code for Information Interchange (ASCII) is one standard for encoding characters in binary. To make it easier for humans these binary codes can also be output in denary.

For example:

65 = "A"
66 = "B"
67 = "C" etc.

Note that uppercase and lowercase letters have different codes.

Associated keywords: `ord`

CLOSE

pipe.close()

Examples:

```
my_file.close()
```

Method: Closes the file pointed to by the pipe.

It is considered good practice to close a file as soon as possible before further processing takes place. Not only does this release the file for other code that may require the file, but it can in rare cases prevent the file becoming corrupted.

Associated keywords: `open, read, readline, write`

COPY

newlist = *list*.copy()

Examples:

```
mylist = oldlist.copy()
```

Method: Copies all the items in one list into another.

Note you cannot use `mylist = oldlist` with lists as you would with variables because `mylist` will become a *pointer* to `oldlist` in memory. This means you will still be updating the same list if you don't use the `copy` method.

DEF

def *identifier*(*parameters*):

Examples:

```
def square(x):
      x = x * x
      return x
```

Command: Defines a new subprogram. Subprograms are also called *subroutines*.

Code must be indented inside the subprogram. You cannot use spaces in the identifier name of the subprogram. It is common to use underscores to separate words in the name of the subprogram instead.

Don't forget the colon at the end of this command.

Subprograms can be *procedures* that do not return a value or *functions* that do return a value. Procedures are used to structure a program into smaller more manageable parts. This is known as *problem decomposition*. Functions are used to create reusable program components. Subprograms avoid unnecessary code duplication and make the code easier to read which also makes finding errors in code, called *debugging* easier.

Associated keywords: `return`

FIND

variable = *string*.find(*string*)

Examples:

```
i = "Hello World".find("W")
```

Method: Returns the index of where a string can be found inside another string.

In the example above, i is assigned to be 6, the index of the character "W" in the string, "Hello World". Like items in a list, each character of a string has an index. Strings are zero-indexed which means the first character is stored at index 0. Remember that a space is also a character.

The method only returns the first occurrence of the search string. If the string cannot be found -1 is returned.

If you want to find more than one occurrence in the string, you should consider using a search algorithm instead. For example, a linear search can use a for loop to iterate over all the characters in a string.

Associated keywords: `replace`

FLOAT

variable = float(*parameter*)

Examples:

```
x = float("5")
```

```
x = float(6)
```

```
x = float(y)
```

Function: Casts a parameter to a floating point number.

Different types of data are stored in different ways by the computer. Inputs taken from the keyboard are always sequences of alphanumeric characters called *strings*. These sequences of characters are stored as numbers by the computer using a standard such as *ASCII* or *Unicode*. The input string "5.6" (strings are *qualified* by quotes) is stored as the numbers 53, 46, 54. To do calculations on the input, it first needs to be converted from a string to a number, or from 53, 46, 54 to 5.6. This is known as *casting*.

Whole numbers with no fractional part are known as *integers*. Numbers with a fractional part (decimal places) are known as *floating points*, *floats*, *real numbers* or *reals*.

A *run-time error* will occur if the parameter cannot be cast to a float.

Associated keywords: `int, str`

FOR

for *variable* in *sequence*:

indented code

Examples:

```
for counter in range(5):
      print(counter)
for counter in range(5, 0, -1):
      print(counter)
for item in mylist:
      print(item)
```

Command: Repeats the indented section of code a given number of times.

The sequence can be a set of numbers defined by the range command or the elements of a list.

Code to be executed must be indented. This is often the source of many logic errors, so check your code is indented correctly.

Repeated sections of code are known as *iterations* or *loops*. Use a for command when you want to repeat a known number of times, for example to iterate over all the items in a list.

It is good practice to comment before this command to explain the purpose of the iteration.

It is possible to include another for command within an indented section. This is known as *nesting*.

Although the functionality of a for command can be replicated with a while command it is usually considered good practice to use a for loop for finite iterations.

As a for command will always iterate a finite number of times, to maximise the efficiency of an algorithm you should consider if a while loop or an alternative command could be used instead to terminate the iteration early. For example, if you need to find or consider all the items in a list then a for command is ideal. If you only need to find the first occurrence consider using the index method instead.

It is possible to terminate a for loop before it is complete with a break command, but this is not considered good practice because it creates more than one exit point for the command which increases the complexity of testing. The use of break should usually be avoided.

`Associated keywords: #, range, in, while`

FORMAT

"string {*parameter index*: *format*}".format(*parameters*)

Examples:

```
name = "Dave"

age = 18

print("Hello {0}. You are {1} years
old.".format(name, age))


value = 4.56754

print("{0:.2f}".format(3.56756))
```

Method: Formats a string for output.

As an alternative to concatenation, string formatting provides more options to manipulate variables used in the output.

The parameters are the values to be used, separated with a comma. Each parameter is referred to by an index within the string. E.g.

```
"{0} £{1}".format(a, b)
```

{0} is the first parameter a, {1} is the second parameter, b. The indexes are replaced with the parameters in the string output.

```
label = "Price"

price = 2.99

print("{0} £{1}".format(label, price))
```

Would result in the output:

```
Price £2.99
```

String formatting is often used to output a number to a number of decimal places. For example. 2.99183 to two decimal places is 2.99.

This is achieved with the format `.2f`

E.g.

```
price = 2.99183
print("Price: £{0:.2f}".format(price))
```

Note that this does not perform any rounding, it truncates the output to two decimal places.

Associated keywords: `print`

IF

If *condition*:

indented code

elif *condition*:

indented code

else:

indented code

Examples:

```
If x == y:
      print("The value of x is the same as y")
elif x < y:
      print("The value of x is less than y")
else:
      print("The value of x is greater than y")
```

Command: Selects which code branch to execute next depending on the outcome of a condition.

The condition requires two variables or constants to be compared with mathematical or logic operators (see appendix 1). More than one condition can be combined with logic operators and brackets can be used to group conditions.

E.g.

```
if ((x > y) and (x > 6)):
```

Note that you should not use `if x > y > 6` as it will not work as you expect. Instead, be explicit about which **two** items of data are being compared in **each** condition.

The result of an `if` command is always either True or False.

That means you can also use a shorthand for Boolean conditions. E.g.

`if valid:` is the same as `if valid == True:`

`if not valid:` is the same as `if valid == False:`

The `elif` section is an optional part of the command to include alternative conditions. You can include as many additional elif sections as you need but consider using the command `match` instead if you require more than one elif section.

The `else` section is an optional part to execute if none of the conditions are met, including those in elif sections.

Code to be executed for each section must be indented. This is often the source of many logic errors, so check your code is indented correctly.

It is good practice to comment each section of this command to explain the purpose of each condition.

It is possible to include another if command within an indented section. This is known as *nesting*.

Associated keywords: `#, match, in`

IMPORT

import *library*

Examples:

```
import random
```

```
import math, os, time, turtle
```

Command: Includes additional commands in your program.

Python includes a basic set of commands, but these can be extended by including additional functions provided in *libraries*.

For example, functions associated with random number generation are included in a library called `random`. Functions associated with more advanced mathematics are included in the `math` library. File and folder manipulation in the `os` library and turtle graphics in the `turtle` library. These are just a few of the common libraries, but there are many more.

Libraries speed up programming by using code that has been created by other people with expertise and should already be error free.

Any additional commands that your program requires will need to be *translated* into *machine code* by the computer when your program runs, so to reduce memory requirements and redundant code programmers only import the libraries that their program needs.

It is possible to create your own library of functions.

IN

variable in *list*

Examples:

```
if "Dave" in ["Craig", "Dave"]
```

```
if x in y
```

```
while x in y
```

Command: Returns whether a constant or a variable is contained within a list.

The `in` command is very useful for checking if an item exists within a list and negates the need for a searching algorithm. It can be used as a condition for selection and iteration commands.

Returns True if the variable is in the list or False if not.

Associated keywords: `if, while`

INDEX

variable = list.index(*parameter*)

Examples:

```
i = mylist.index("C")
```

Method: Returns the index of where the parameter can be found in a list.

For example, if a list contained: `["A", "B", "C", "D"]` then `mylist.index("C")` would return 2 because the item "C" is stored at index 2. Remember that lists are zero-indexed which means the first item is stored at index 0. The method only returns the first occurrence of an item in a list.

If the item is not in the list a run-time error will occur so you should only use this method after the condition: `if parameter in mylist` is True.

If you want to find more than one occurrence in the list, you should consider using a search algorithm instead. For example, a linear search can use a for loop to iterate over all the items in a list.

Associated keywords: `find`

INPUT

variable = input(*parameter*)

Examples:

```
surname = input("Enter your surname: ")
```

```
prompt = "Enter your forename: "
forename = input(prompt)
```

Function: returns an input from the keyboard.

Inputs from the keyboard allow user interaction with your program. Inputs are always sequences of alphanumeric characters called *strings* terminated when the user presses the enter key.

The parameter prompt to the user is optional but informs the user what data they are expected to enter.

It is common to add an extra space at the end of the prompt to separate the prompt and the input.

Remember to do calculations on the input, the input data will need to be *cast* to an *integer* or *float*.

INSERT

list.insert(*index*, *parameter*)

Examples:

```
mylist.insert(2, "Dave")
```

Method: Inserts the parameter into a list.

The example above inserts the string "Dave" into the list called mylist at index 2. The indexes of the existing items at index 2 and above are *incremented*. The parameter can be a value, a variable or a list.

You can overwrite existing elements of a list by referring to their index. E.g. `list[2] = "Dave"` will replace what is currently stored in index 2.

Associated keywords: `append, pop, remove`

INT

variable = int(*parameter*)

Examples:

```
x = int("5")
```

```
x = int(6.5)
```

```
x = int(y)
```

Function: Casts a parameter to an integer.

Different types of data are stored in different ways by the computer. Inputs taken from the keyboard are always sequences of alphanumeric characters called *strings*. These sequences of characters are stored as numbers by the computer using a standard such as *ASCII* or *Unicode*. The input string "5" (strings are *qualified* by quotes) is stored as the number 53. To do calculations on the input, it first needs to be converted from a string to a number, or from 53 to 5. This is known as *casting*.

Whole numbers with no fractional part are known as *integers*. Numbers with a fractional part (decimal places) are known as *floating points*, *floats*, *real numbers* or *reals*. Using int to cast a float to an integer also removes the fractional component.

A *run-time error* will occur if the parameter cannot be cast to an integer.

Associated keywords: `float, str`

ISALPHA

string.isalpha()

Examples:

```
if surname.isalpha():
      print("The surname is valid")
else:
      print("The surname is invalid")
```

Method: returns True if the string only contains the letters a to z or A to Z

Can be useful for *validation* routines.

Associated keywords: `isdigit, isalnum`

ISALNUM

string.isalnum()

Examples:

```
if password.isalnumn():
      print("Password is not strong enough")
```

Method: returns True if the string only contains the letters a to z or A to Z or the digits 0 to 9.

Can be useful for *validation* routines.

Associated keywords: `isdigit, isalpha`

ISDIGIT

string.isdigit()

Examples:

```
if not month.isdigit():
      print("The month is invalid")
```

Method: returns True if all the characters in the string are the digits 0 to 9. If this is the case the string can be cast to an integer or float.

Can be useful for *validation* routines.

Associated keywords: `isalpha, isalnum, int, float`

ISLOWER

string.islower()

Examples:

```
if surname.islower():
```

Method: returns True if all the characters in the string are in lowercase (small letters) or False if not.

Can be useful for *validation* routines.

Associated keywords: `isupper, lower, upper`

ISUPPER

string.isupper()

Examples:

```
if surname.isupper():
```

Method: returns True if all the characters in the string are in uppercase (capital letters) or False if not.

Can be useful for *validation* routines.

Associated keywords: `islower, lower, upper`

JOIN

variable = *string*.join(*list*)

Examples:

```
mylist = ["Hello", "World"]
```

```
mystring = "-".join(mylist)
```

Method: Concatenates all items in a list into a single string variable separated by a chosen string.

In the example above, mystring would be assigned, "Hello-World". Each element of mylist has been joined together separated by a hyphen.

Be careful when using escape characters as concatenators. For example, `mystring = "\".join(mylist)` will generate a syntax error.

Join can be useful for creating the output for a single record from multiple fields for a CSV file, where each item is separated by a comma.

Associated keywords: `split`

LEN

variable = len(*parameter*)

Examples:

```
x = len("A", "B", "C")
```

```
x = len(mylist)
```

Function: Returns the number of indexes in the parameter.

Remember the number of indexes includes the first index which is index 0.

The parameter can be a string, in which case the number of characters is returned, or a list in which case the number of elements is returned.

LOWER

Variable = string.lower()

Examples:

```
email = myaddress.lower()
```

Method: returns the string in lowercase (small letters).

Associated keywords: `islower, isupper, upper`

MATCH

match *variable*:

case *value*:

indented code

case _:

indented code

Examples:

```
match day_name:
      case "Thu" | "Fri" | "Sat":
            print("Open 10-5pm")
      case "Sun":
            print("Open 11-3pm")
      case _:
            print("Closed.")
```

Command: An alternative to the if/elif/else structure that is used when there are more than two outcomes. Match is considered a better command to use because it is more readable for multiple values. Don't forget the colon after match and each case.

Possible values can be separated with pipe characters: | (this is the equivalent to the logical OR operator). You can have as many case statements as you need, and each one should include a comment.

`case _:` is the equivalent to `else` and captures any value that was not matched.

Note the match command is only supported in Python 3.10+

Associated keywords: #, `if`

MATH.CEIL

variable = math.ceil(*parameter*)

Examples:

```
x = 65.23
x = math.ceil(x)
```

Function: Rounds a number up to the nearest integer.

In the example above x would be assigned 66 as 0.5 rounding is ignored.

Associated keywords: `import, math.floor, round`

MATH.FLOOR

variable = math.floor(*parameter*)

Examples:

```
x = 65.83
```

```
x = math.floor(x)
```

Function: Rounds a number down to the nearest integer.

In the example above x would be assigned 65 as 0.5 rounding is ignored.

Associated keywords: `import, math.ceil, round`

MATH.PI

variable = math.pi

Examples:

```
x = math.pi
```

Function: Returns the constant pi as 3.141592653589793.

Associated keywords: `import`

MATH.SQRT

variable = math.sqrt(*parameter*)

Examples:

```
x = math.sqrt(25)
```

```
x = math.sqrt(y)
```

Function: Returns the square root of a number.

Associated keywords: `import`

OPEN

variable = open(*filename*, *operation*)

Examples:

```
my_file = open("mydata.txt", "r")
```

Command: Opens a serial text file for reading or writing data.

The first parameter is a filename that data is to be read from or written to. The filename can be a variable and also include a file path. The default path is the same folder that the .py program is stored in.

The second parameter is the operation to perform on the file:

"r" Read data from the file.

"w" Overwrite and existing data in the file.

"a" Append data to the end of the file.

Overwrite and append operations will create a file if one does not already exist. A run-time error will occur if reading is attempted, and the file does not exist.

The variable becomes what is known as a pipe or pointer to the file. Files must always be opened before data can be read from or written to the pipe.

Note a file can only be open for either reading or writing at any one time. You cannot read and write to the same file at the same time.

Associated keywords: `close, read, readline, write`

ORD

variable = ord(*character*)

Examples:

```
ascii_code = ord("A")
```

Function: Returns the ASCII code of a character.

All characters, including letters, numbers and symbols stored in strings are actually stored in binary by the computer. The American Standard Code for Information Interchange (ASCII) is one standard for encoding characters in binary. To make it easier for humans these binary codes can also be output in denary.

For example:

"A" = 65
"B" = 66
"C" = 67 etc.

Note that uppercase and lowercase letters have different codes.

Associated keywords: `chr`

POP

list.pop(*index*)

Examples:

```
mylist.remove(2)
```

Method: Removes an index from a list.

The example above removes the item stored at index 2 from the list called mylist. Remember that this will reduce the index of all other elements stored after the index by -1.

Associated keywords: `append, insert, remove`

PRINT

print(*parameters*)

Examples:

```
print("Hello World")
```

```
print(x)
```

Command: Outputs a value to the screen. The value can be any data type: Boolean, integer, list, float, string.

Don't forget that what you want to output must be enclosed in brackets. Strings will need to be qualified by quotes.

Each print statement puts the output on a new line. You can prevent a new line by concatenating end = "" to the output. E.g.

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

```
print("World")
```

A single blank line, or the end of a line can be output with:

```
print()
```

Multiple parameters can be output on one line. Each parameter is separated with a comma. The output will include an automatic space between each parameter:

```
print("You are", age, "years old")
```

Associated keywords: `format`

RANDOM.CHOICE

random.choice(list)

Examples:

```
letter = random.choice(["A", "B", "C"])
```

Function: Returns a random element from a list. This command must be imported from the `random` library before use.

To generate random elements and not a deterministic sequence, this command also requires `random.seed()` to be used once in the program before `random.choice`.

Associated keywords: `import, random.seed, random.shuffle`

RANDOM.RANDINT

random.randint(*low_value*, *high_value*)

Examples:

```
dice = random.randint(1, 6)
```

Function: Returns a random number between the low and high value parameters inclusive. This command must be imported from the `random` library before use.

To generate random numbers and not a deterministic sequence, this command also requires `random.seed()` to be used once in the program before `random.randint`.

Associated keywords: `import, random.choice, random.seed, random.shuffle`

RANDOM.SEED

random.seed(*value*)

Examples:

```
random.seed()
```

```
random.seed(1)
```

Command: Sets the seed to be used by the random number function. This command must be imported from the `random` library before use.

Computers cannot generate random numbers because they can only perform calculations. Instead, they use a calculation on a number known as a *seed* to generate what looks like a random number to the user. For example, the fractional part of the square root of 55 is 4161984871. If you didn't know the algorithm and the seed value of 55, these digits would appear to be random. Python uses an algorithm known as *Mersenne Twister* to generate random numbers.

By not specifying a value for the seed, the random number function will use the time of day as the seed instead. Specifying a value will ensure the random number function always generates the same deterministic sequence of numbers.

It is important to use `random.seed()` once at the beginning of your program to ensure you get random numbers. You should not need to use this command more than once in your program.

Associated keywords: `import, random.choice, random.randint, random.shuffle`

RANDOM.SHUFFLE

random.shuffle(*list*)

Examples:

```
random.shuffle(mylist)
```

Method: Reorders the items in a list into a random order.

This command must be imported from the `random` library before use.

Associated keywords: `import, random.choice, reverse, sort`

RANGE

range(parameter, parameter, parameter)

Examples:

```
for counter in range(5):
```

```
for counter in range(5, 0, -1)
```

Command: Enumerates a set of numbers.

Used with the for command, range creates a set of numbers for the iteration. For example, `range(5)` will produce the numbers 0, 1, 2, 3, 4. That's five numbers starting at zero.

The set of numbers can be defined with additional parameters. For example, `range(5, 0, -1)` will create a set of five numbers starting at five in increments of -1. The set is 5, 4, 3, 2, 1.

The start, end and increment can be any values. An invalid set of parameters that could never be completed, e.g. (5, 0, 1) will return an empty set.

Associated keywords: `for`

READ

variable = pipe.read()

Examples:

```
my_file = open("data.txt", "r")
```

```
data = my_file.read()
```

Method: Reads all the data from an open file.

The variable becomes all the data in the file. To read just one line at a time, use `readline` instead.

The data will include a hidden end of line escape character. You should always remove this with the `strip` command once the data has been read.

Associated keywords: `close, open, readline, split, strip, write`

READLINE

variable = pipe.readline()

Examples:

```
my_file = open("data.txt", "r")
data = my_file.readline()
```

Method: Reads one line of data from an open file.

The variable becomes all the data in the file until the end of line escape code is reached. To read the whole file at once, use `read` instead.

The data will include a hidden end of line escape character. You should always remove this with the `strip` command once the data has been read.

This method is usually used with a while loop if you want to stop reading once an item has been found, or with a for loop to read all the data from a file.

For example:

```
my_file = open("data.txt", "r")
for line in my_file:
      print(line.strip())
```

Associated keywords: `close, open, read, split, strip, write`

REMOVE

list.remove(*parameter*)

Examples:

```
mylist.remove("Dave")
```

Method: Removes the first occurrence of the parameter from a list.

The example above removes the string "Dave" in the list called mylist. Remember that this will reduce the index of all other elements stored after this index by -1.

An error will occur if the parameter is not in the list, so this should be checked with the commands `if` and `in` before using this command.

To remove all instances of the parameter without an error you would need to use a while loop. E.g.

```
while "Dave" in mylist:
      mylist.remove("Dave")
```

Associated keywords: `append, insert, pop`

REPLACE

***variable* = *string*.replace(*string*, *string*)**

Examples:

```
sentence = "The quick brown fox"

new_sentence = sentence.replace("fox", "dog")
```

Method: Replaces all occurrences of one string with another.

In the example above, new_sentence would be assigned, "The quick brown dog".

Associated keywords: `find`

RETURN

return *expression*

Examples:

```
def square(x):
      return x * x


y = square(5)
```

Command: Returns a value from a subprogram.

Subprograms that return values are called *functions*.

In the example above, the number 5 is passed into the function called square and assigned to the parameter x. The variable is then multiplied by itself and returned as the output from the function square into the variable y which assigns it the value 25.

The return expression can be a Boolean, e.g. return True, a variable, e.g. return total, a list or the result of a calculation.

It is possible to return more than one value in Python, each separated with a comma, but many languages do not support this so it is generally avoided in favour of returning a list.

Subprograms that do not return a value are called *procedures*.

Associated keywords: `def`

REVERSE

list.reverse()

Examples:

```
mylist.reverse()
```

Method: Reverses the items in a list.

This is useful if you want the items in a list in descending order. Use `.sort()` to initially sort the items and then reverse the order with `.reverse()`.

Associated keywords: `sort, random.shuffle`

ROUND

variable = round(*parameter, parameter*)

Examples:

```
x = round(6.532234, 2)
```

```
x = round(y, 3)
```

Function: Rounds a number to a given number of decimal places using the 0.5 rule. E.g. 6.2 would not be rounded up, whereas 6.6 would be.

Associated keywords: `math.ceil, math.floor`

SETUP

Variable.setup(parameter, parameter)

Examples:

```
window.setup(800, 600)
```

Method: Sets the size of the turtle window variable in pixels. The first parameter is the width, the second parameter is the height.

Requires use of `turtle.Screen()` to initialise the window variable first.

Associated keywords: `turtle.Screen, turtle.Screen, turtle.screensize`

SORT

list.sort()

Examples:

```
mylist.sort()
```

Method: Sorts the items in a list into ascending order.

Python uses a *Tim Sort* to order the items in a list.

Associated keywords: `reverse, random.shuffle`

SPLIT

list = *string*.split(*string*)

Examples:

```
mystring = "Hello,World"
```

```
mylist = mystring.split(",")
```

Method: Splits a string into a list.

In the example above, mylist would be assigned, ["Hello", "World"]

A new element is created each time the separator characters are found in the string. In the example above, a comma is used to identify where items should be split.

Split can be useful for inputting a single record from a CSV file.

Associated keywords: `join`

STR

variable = str(*parameter*)

Examples:

```
x = str(6.5)
```

```
x = str(price)
```

Function: Casts a parameter to a string (sequence of alphanumeric characters).

Different types of data are stored in different ways by the computer. Although calculations must be performed on integers and real numbers, inputs and outputs are always strings.

Don't confuse the integer 6 with the string "6". They have different binary codes inside the computer even though they look the same to the user.

It can be necessary to convert or cast a number into a string before it can be concatenated, used with string manipulation or format commands.

Associated keywords: `int, float`

STRIP

variable = string.strip(parameter)

Examples:

```
mystring = "Hello World  "
mystring = mystring.strip()
```

Method: Removes whitespace and hidden escape characters from the beginning or end of a string.

In the example above, the spaces would be removed from the end of the string, but not between the words inside the string.

An optional string parameter can be specified if there are particular characters to remove. For example, if the parameter was "Z" then any leading and training "Z" characters would be removed.

This command is essential to use after reading data from a file to ensure the hidden end of line/record characters are removed before further processing. It can also be helpful as an initial step for pre-processing inputs before validation.

Associated keywords: `read`

TIME.SLEEP

time.sleep(value)

Examples:

```
time.sleep(5)
```

Method: Delays the next line of code executing for a given number of seconds determined by the value.

This command must be imported from the `time` library before use.

Associated keywords: `import`

TRY

try:

indented code

except:

indented code

else:

indented code

finally:

indented code

Examples:

```
try:
      my_file = open("data.txt", "r")
except FileNotFoundError:
      my_file = open("data.txt", "w")
      my_file.write(boilerplate)
      file.close()
      my_file = open("data.txt", "r")
      print("New file created. ")
else:
      print("File opened successfully.")
```

```
finally:
        print("Ready to read data from the file.")
```

Command: Handles run-time exception errors to prevent a program from crashing unexpectedly.

There are some commands that can fail when a program is running due to exceptional circumstances. For example, if a program reads a data file but the user has deleted it or when attempting to write to a file the secondary storage is full. In these cases, the program will crash.

It is considered good practice to trap potential run-time errors with what is known as exception handling techniques.

In the example above a file called data.txt is opened for reading, but if the file does not exist a new file is created, and boilerplate data written to it before it is opened. This ensures the program always has data to read from the file.

The try command is used to denote the start of an exception handling event.

If an exception occurs, the indented code in the except section will be executed. There is no need to specify the type of error that can occur, but it is considered good practice to do so. For example, except FileNotFoundError will execute the indented section if the file cannot be found. More than one except section can be used for multiple events.

The else section executes if no error occurred.

The finally section executes regardless of whether an error occurred or not.

The try command should also be used for possible situations where a division by zero could occur as this will also crash the program.

Associated commands: `open, read, readline, write`

TURTLE.BACK

turtle.back(value)

Examples:

```
turtle.back(100)
```

Method: Moves the turtle backwards a given number of pixels.

Associated keywords: `turtle.forward`

TURTLE.BEGIN_FILL

turtle.begin_fill()

Examples:

```
turtle.begin_fill()
```

Method: Initiates the start of a shape fill.

Associated keywords: `turtle.end_fill, turtle.fillcolor`

TURTLE.CIRCLE

turtle.circle(value, value)

Examples:

```
turtle.circle(100)
```

```
turtle.circle(100, 180)
```

Method: Draws a circle of a radius of the first value to the extent of the second value.

For example, `turtle.circle(100)` will draw a full circle with a radius of 100 pixels. `turtle.circle(50, 180)` will draw a semicircle with a radius of 50 pixels.

Associated keywords: `turtle.begin_fill, turtle.endfill, turtle.fillcolor, turtle.pencolor, turtle.pensize`

TURTLE.DONE

turtle.done()

Examples:

```
turtle.done()
```

Method: Use as the last line of the program to keep the turtle window open until it is closed by the user.

Associated keywords: All turtle library commands.

TURTLE.END_FILL

turtle.end_fill()

Examples:

```
turtle.end_fill()
```

Method: Completes the shape filling process. Once the lines create an enclosed shape it will be filled when this command is executed. The shape fill must be initiated before the first line of the shape is drawn with `turtle.begin_fill()`.

Associated keywords: `turtle.begin_fill, turtle.fillcolor`

TURTLE.FILLCOLOR

turtle.fillcolor(string)

Examples:

```
turtle.fillcolor("blue")
```

Method: Sets the colour to be used to fill a shape.

See appendix for colours.

Associated keywords: `turtle.pencolor`

TURTLE.FORWARD

turtle.forward(value)

Examples:

```
turtle.forward(100)
```

Method: Moves the turtle forwards a given number of pixels.

Associated keywords: `turtle.back`.

TURTLE.HIDETURTLE

turtle.hideturtle()

Examples:

```
turtle.hideturtle()
```

Method: Makes the turtle invisible although it will still draw if the pen is down.

Associated keywords: `turtle.showturtle`

TURTLE.HOME

turtle.home()

Examples:

```
turtle.home()
```

Method: Moves the turtle to the position 0, 0.

Associated keywords: `turtle.reset, turtle.setposition`

TURTLE.LEFT

turtle.left(value)

Examples:

```
turtle.left(90)
```

Method: Turns the turtle anticlockwise a given number of degrees.

Associated keywords: `turtle.right`, `turtle.setheading`

TURTLE.MODE

turtle.mode(string)

Examples:

```
turtle.mode("standard")
```

```
turtle.mode("logo")
```

A turtle in standard mode, initially points to the right (east) and angles are counterclockwise.

A turtle in logo mode, initially points up (north) and angles are clockwise.

Note this method is not supported in all turtle library implementations and will cause an error if not.

Associated keywords: `turtle.setheading`

TURTLE.PENCOLOR

turtle.fillcolor(string)

Examples:

```
turtle.pencolor("blue")
```

Method: Sets the colour to be used for drawing lines.

See appendix for colours.

Associated keywords: `turtle.fillcolor`

TURTLE.PENDOWN

turtle.pendown()

Examples:

```
turtle.pendown()
```

Method: Puts the pen on the drawing canvas so as the turtle moves lines will be drawn (default).

Associated keywords: `turtle.pensize, turtle.penup`

TURTLE.PENSIZE

turtle.pensize(value)

Examples:

```
turtle.pensize(5)
```

Method: Sets the width of the line that is drawn in pixels. Value must be positive. The default value is 1.

Associated keywords: `turtle.pendown, turtle.penup`

TURTLE.PENUP

turtle.penup()

Examples:

```
turtle.penup()
```

Method: Lifts the pen off the drawing canvas so as the turtle moves lines will not be drawn.

Associated keywords: `turtle.penup`

TURTLE.RESET

turtle.reset()

Examples:

```
turtle.reset()
```

Method: Clears the drawing canvas, sets the position of the turtle to 0, 0 and resets the turtle properties to their default values.

Associated keywords: `turtle.home, turtle.setposition`

TURTLE.RIGHT

turtle.right(value)

Examples:

```
turtle.right(90)
```

Method: Turns the turtle clockwise a given number of degrees.

Associated keywords: `turtle.left, turtle.setheading`

TURTLE.SCREEN

Variable = turtle.screen()

Examples:

```
window = turtle.screen()
```

Method: Returns a variable to address the turtle window.

Associated keywords: `setup, turtle.screensize`

TURTLE.SCREENSIZE

turtle.screensize(parameter, parameter)

Examples:

```
turtle.screensize(800, 600)
```

Method: Sets the size of the scrollable drawing canvas in pixels. Scrollbars are only active if turtle drawing terminates with the method `turtle.done()`.

Associated keywords: `setup, turtle.screen`

TURTLE.SETHEADING

turtle.setheading(value)

Examples:

```
turtle.setheading(90)
```

Method: Sets the turtle orientation in degrees regardless of the direction the turtle is currently facing.

If the turtle mode is standard, a heading of zero faces right and positive values are anticlockwise.

If the turtle mode is logo, a heading of zero faces up and positive values are clockwise.

Associated keywords: `turtle.left, turtle.mode, turtle.right`.

TURTLE.SETPOSITION

turtle.setposition(value, value)

Examples:

```
turtle.setposition(100, 100)
```

```
turtle.setposition(-50, -50)
```

Method: Sets the position of the turtle on the drawing canvas in pixels. The first value is the horizontal position, the second value is the vertical position.

Associated keywords: `turtle.home`

TURTLE.SHOWTURTLE

turtle.showturtle()

Examples:

```
turtle.showturtle()
```

Method: Makes the turtle visible.

Associated keywords: `turtle.hideturtle`.

TURTLE.SPEED

turtle.speed(parameter)

Examples:

```
turtle.speed(5)
```

```
turtle.speed("fast")
```

Method: Sets the speed of the turtle when moving.

Parameters can be:

- Values 0-10. Zero is the fastest, ten is the slowest.
- "fastest"
- "fast"
- "normal"
- "slow"
- "slowest"

Associated keywords: All turtle library commands.

TURTLE.TURTLE

Variable = turtle.Turtle()

Examples:

```
x = turtle.Turtle()
```

Method: Creates a new turtle with the variable name x.

Associated keywords: All turtle library commands.

UPPER

Variable = string.upper()

Examples:

```
surname = sinput.upper()
```

Method: returns the string in uppercase (capital letters).

Associated keywords: `islower, isupper, lower`

WHILE

while *condition*:

indented code

Examples:

```
valid_input = False
while not valid_input:
      print("Enter your choice: ")


while choice < 0 or choice > 3:
      print("Enter your choice: ")
```

Command: Repeats the indented section of code until the condition is not met.

Code to be executed must be indented. This is often the source of many logic errors, so check your code is indented correctly.

Repeated sections of code are known as *iterations* or *loops*. Use a while command when it is not known in advance how many iterations will be required.

It is common to ensure the condition cannot be met before the first iteration to ensure the indented code executes at least once.

It is good practice to comment before this command to explain the purpose of the iteration or condition.

More than one condition can be combined with logic operators and brackets can be used to group conditions.

It is possible to include another if commands within an indented section. This is known as *nesting*.

While loops are often used with indented input commands for *validation*, ensuring that the user has entered a valid input before continuing the program.

A special value that uses its presence as a condition to terminate a loop is called a *sentinel value*.

Infinite loops can be created with `while True:` since true will always be true.

Associated keywords: `#, for, in`

WRITE

pipe.write(*variable*)

Examples:

```
my_file = open("data.txt", "w")
```

```
my_file.write("Hello World")
```

Method: Writes data to an open file.

If the file was opened with the "w" operation any existing data in the file will be overwritten. If the file was opened with the "a" operation, data will be appended to the end of the file without overwriting existing data.

Note if you want the next item of data written to the file to be on a new line, you must include the end of line escape code "\n".

It is considered good practice to prepare the data to be written in a single variable and then write that data in one command.

For example:

```
data = item1 + "," + item2 + "\n"
```

```
my_file.write(data)
```

This will write variables item1 and item2 separated by a comma to the file.

Associated keywords: `close, open, read, readline, strip`

Appendix 1

Concatenation

```
x = "Hello" + " " + "World"
```

To concatenate means to join together. A comma can be used to concatenate strings inside a print statement. A plus symbol needs to be used outside of a print statement.

Numbers should be cast to strings before they are concatenated.

Comparison operators

==	`if x == y`	Is x the same as y? (equal)
!=	`if x != y`	Are x and y different? (not equal)
<	`if x < y`	Is x less than y?
<=	`if x <= y`	Is x less than or equal to y?
>	`if x > y`	Is x greater than y?
>=	`if x >= y`	Is x greater than or equal to y?

Note that a double equal is asking a question, a single equal assigns a variable. E.g.

`x == 6` means is x equal to 6?

`x = 6` means x becomes the number 6.

Logical operators

and	`if x > y and x > 6:`	Both conditions must be true for the result to be True.
or	`if x > y or x > 6:`	One of the conditions must be true for the result to be True.
not	`if not x:`	The condition must not be met for the result to be True.

Mathematical operators

+	`x = 6 + 5`	Addition
-	`x = 6 - 5`	Subtraction
*****	`x = 6 * 5`	Multiplication
/	`x = 6 / 5`	Division
//	`x = 6 // 5`	Integer (floor) division
******	`x = 6 ** 5`	Exponentiation
%	`x = 5 % 5`	Modulus

String manipulation

Substrings

Many languages include commands left, mid and right to extract characters from the left, middle or end of a string.

Extracting from the start of a string:

```
variable = "Hello"[0:2]
```

The variable would be assigned "He".

Extracting from the middle of a string:

```
variable = "Hello"[2:5]
```

The variable would be assigned "llo".

Extracting from the end of a string:

```
variable = "Hello"[-2:]
```

The variable would be assigned "lo".

Creating strings of characters

It is possible to use mathematical operators to create strings of characters. For example:

```
variable = "@" * 5
```

The variable would be assigned "@@@@@".

2D arrays and lists

Creating a data structure

An array is a data structure that does not change its size when the program is running and contains only one data type. E.g. a table of integers.

A list is a data structure that dynamically changes its size when the program is running and can contain any data type.

It is common practice to declare and assign 2D arrays and lists in a single line of code using two for loops:

```
array = [["-" for columns in range(4)] for rows in range(3)]
```

Note the command is written on one line.

This will create a 2D array or list that can be visualised as a table:

array			
Index	0	1	2
0	-	-	-
1	-	-	-
2	-	-	-
3	-	-	-

Note that arrays and lists are zero indexed.

2D arrays and lists are an abstraction of memory. The table is just one way of visualising the data structure. Whether you choose to consider this a 3x4 or 4x3 table is up to you. The variables columns and rows are used so that the code is easier to understand.

In the example above the elements (cells) were all assigned to be a hyphen as specified in the command. This can be any data. An empty string: "" or a number.

Reading CSV files

Consider this typical CSV file:

```
Trial A, 77, 85, 88
Trial B, 74, 92, 100
Trial C, 64, 78, 91
```

This can be stored in a 2D list:

data				
Index	0	1	2	3
0	Trial A	77	85	88
1	Trial B	74	92	100
2	Trial C	64	78	91

Reading data from the CSV file into a list can be achieved with this code:

```
def read_file(filename):
    database = []
    file = open(filename, "r")
    # Read each line of data until end of file
    for record in file:
        record = record.strip()
        fields = record.split(",")
        database.append(fields)
    file.close()
    return database
```

Note this example does not include any exception handling for clarity.

```
data = read_file(filename)
print(data[2][3])
```

Would result in the number 91 being output because the data structure has been created assuming data[row][column].

Appendix 2

Turtle window

The turtle window is one size, and the turtle drawing canvas (inside the window) can be a different size. To make the turtle window bigger, a screen needs to be created and set up. Here is an example:

```
width = 800
height = 400
screen = turtle.Screen()
screen.setup(width, height)
```

To make the drawing canvas bigger use `turtle.screensize()`.

In some development environments, the turtle window will close as soon as the program completes. Add `turtle.done()` as the last line in the code file to keep the window open.

Turtle colours

blue	black	green	yellow
orange	red	pink	purple
indigo	olive	lime	navy
orchid	salmon	peru	sienna
white	cyan	silver	gold

Command Index