# #

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`.

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`

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`

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`

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.

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*.

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`

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`

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.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.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.randint`

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`

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`

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 while command 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: #

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

Command Index