# Tool box:

**Variables:**

Type:

1. String: “hello world!!”
2. Float: the number with decimal spot
3. Integer: the number without decimal spot

**Mod:**

The Python modulo operator calculates the remainder of dividing two values. This operator is represented by the percentage sign (%). The syntax for the modulo operator is: number1 % number2. The first number is divided by the second then the remainder is returned

## **1 for loop**

Example 1:

For i in range(1,1000):

XXX

Example 2:

For element in list\_1:

XXXX

## **2. while (XXX)**

Example 1:

While(Is\_A\_Girl()==TRUE):

XXXX

## **3. if XXX:**

XXX

Elif XXX:

XXX

Else:

XXX

If “XXX” in “XXXX”: “XXXX” is a string. This if statement means check “XXX” is included in “XXXX”

If “abc” in “abcdefg”: True

If “ab” in “acbdefg”: Failed.

## **list: list is your school bag, you can put everything inside.**

List.append(XXX)

List.extend(XXX) XXX should be iterable

List.pop(index)

List.push(index)

List.count(value)

List.remove(value)

List.reverse()

List.sort()

## **5. a = input(“XXX”)**

Note: the type of a is a string

## **6 print(XXX)**

Print out the value of XXX

## **7.string**

String.capitalize()

String.endswith(suffix) -> bool

String.find(sub) -> return the lowest index in S where substring sub is found

String.isdigit()->bool

String.islower()->bool

String.isspace()->bool

String.join(iterable)

Return a string which is the concatenation of the strings in the iterable. The separator between elements is S

String.lstrip()

Return a copy of the strings S with leading white space removed. If chars is given and not None, remove characters in chars instead

String.replace(old, new)->str

String.split(sep)->list of strings

String.splitlines()->list of strings

String.statswith(XXX)->bool

String.strip([chars])

Return a copy of the string S with leading and trailing whitespace removed if chars is given and not None remove characters in chars instead

## **8.function:**

def XXX(a,b,c):

XXXX

Return XX

**9．Dictionary: is a hashable iterable type.**

1. {key:value}

Example:

a = {“Jack” : 89, “Tim” : 29, “Marry” : 100, “Lisa” : 79, “Gos” : 59“Timmy” : 83, “Julia” : 69}

For loop:

For k, v in a.items():

Print(k)

Print(v)

1. **Break:**

Terminate the loop (while, for)

While(1):

XXXXX

If XX:

break

XXXXX

1. **File:**
2. Open(“xxxx.txt”, “r/w/a+”) as f:

r : read only

W: enable write

A+: append student msg into the file.

1. F.Read() => all data type:string
2. F.readlines => read data line by line and return a list .
3. f.close()
4. With open(“xxx.txt”,”r/w/a+”) as f:
5. **List.sort(key = myfunc)**

Def myfunc(e):

XXXX

E presents each element in the list.

1. Regular expression

.

(Dot.) In the default mode, this matches any character except a newline. If the [DOTALL](https://docs.python.org/3/library/re.html" \l "re.DOTALL" \o "re.DOTALL) flag has been specified, this matches any character including a newline.

^

(Caret.) Matches the start of the string, and in [MULTILINE](https://docs.python.org/3/library/re.html" \l "re.MULTILINE" \o "re.MULTILINE) mode also matches immediately after each newline

$

Matches the end of the string or just before the newline at the end of the string, and in [MULTILINE](https://docs.python.org/3/library/re.html" \l "re.MULTILINE" \o "re.MULTILINE) mode also matches before a newline. foo matches both ‘foo’ and ‘foobar’, while the regular expression foo$ matches only ‘foo’. More interestingly, searching for foo.$ in 'foo1\nfoo2\n' matches ‘foo2’ normally, but ‘foo1’ in [MULTILINE](https://docs.python.org/3/library/re.html" \l "re.MULTILINE" \o "re.MULTILINE) mode; searching for a single $ in 'foo\n' will find two (empty) matches: one just before the newline, and one at the end of the string.

\*

Causes the resulting RE to match 0 or more repetitions of the preceding RE, as many repetitions as are possible. ab\* will match ‘a’, ‘ab’, or ‘a’ followed by any number of ‘b’s.

+

Causes the resulting RE to match 1 or more repetitions of the preceding RE. ab+ will match ‘a’ followed by any non-zero number of ‘b’s; it will not match just ‘a’.

?

Causes the resulting RE to match 0 or 1 repetitions of the preceding RE. ab? will match either ‘a’ or ‘ab’.

\*?, +?, ??

The '\*', '+', and '?' qualifiers are all greedy; they match as much text as possible. Sometimes this behaviour isn’t desired; if the RE <.\*> is matched against '<a> b <c>', it will match the entire string, and not just '<a>'. Adding ? after the qualifier makes it perform the match in non-greedy or minimal fashion; as few characters as possible will be matched. Using the RE <.\*?> will match only '<a>'.

{m}

Specifies that exactly m copies of the previous RE should be matched; fewer matches cause the entire RE not to match. For example, a{6} will match exactly six 'a' characters, but not five.

{m,n}

Causes the resulting RE to match from m to n repetitions of the preceding RE, attempting to match as many repetitions as possible. For example, a{3,5} will match from 3 to 5 'a' characters. Omitting m specifies a lower bound of zero, and omitting n specifies an infinite upper bound. As an example, a{4,}b will match 'aaaab' or a thousand 'a' characters followed by a 'b', but not 'aaab'. The comma may not be omitted or the modifier would be confused with the previously described form.

{m,n}?

Causes the resulting RE to match from m to n repetitions of the preceding RE, attempting to match as few repetitions as possible. This is the non-greedy version of the previous qualifier. For example, on the 6-character string 'aaaaaa', a{3,5} will match 5 'a' characters, while a{3,5}? will only match 3 characters.

\

Either escapes special characters (permitting you to match characters like '\*', '?', and so forth), or signals a special sequence; special sequences are discussed below.

If you’re not using a raw string to express the pattern, remember that Python also uses the backslash as an escape sequence in string literals; if the escape sequence isn’t recognized by Python’s parser, the backslash and subsequent character are included in the resulting string. However, if Python would recognize the resulting sequence, the backslash should be repeated twice. This is complicated and hard to understand, so it’s highly recommended that you use raw strings for all but the simplest expressions.

[]

Used to indicate a set of characters. In a set:

Characters can be listed individually, e.g. [amk] will match 'a', 'm', or 'k'.

Ranges of characters can be indicated by giving two characters and separating them by a '-', for example [a-z] will match any lowercase ASCII letter, [0-5][0-9] will match all the two-digits numbers from 00 to 59, and [0-9A-Fa-f] will match any hexadecimal digit. If - is escaped (e.g. [a\-z]) or if it’s placed as the first or last character (e.g. [-a] or [a-]), it will match a literal '-'.

Special characters lose their special meaning inside sets. For example, [(+\*)] will match any of the literal characters '(', '+', '\*', or ')'.

Character classes such as \w or \S (defined below) are also accepted inside a set, although the characters they match depends on whether [ASCII](https://docs.python.org/3/library/re.html" \l "re.ASCII" \o "re.ASCII) or [LOCALE](https://docs.python.org/3/library/re.html" \l "re.LOCALE" \o "re.LOCALE) mode is in force.

Characters that are not within a range can be matched by complementing the set. If the first character of the set is '^', all the characters that are not in the set will be matched. For example, [^5] will match any character except '5', and [^^] will match any character except '^'. ^ has no special meaning if it’s not the first character in the set.

To match a literal ']' inside a set, precede it with a backslash, or place it at the beginning of the set. For example, both [()[\]{}] and []()[{}] will both match a parenthesis.

Support of nested sets and set operations as in [Unicode Technical Standard #18](https://unicode.org/reports/tr18/) might be added in the future. This would change the syntax, so to facilitate this change a [FutureWarning](https://docs.python.org/3/library/exceptions.html" \l "FutureWarning" \o "FutureWarning) will be raised in ambiguous cases for the time being. That includes sets starting with a literal '[' or containing literal character sequences '--', '&&', '~~', and '||'. To avoid a warning escape them with a backslash.

|

A|B, where A and B can be arbitrary REs, creates a regular expression that will match either A or B. An arbitrary number of REs can be separated by the '|' in this way. This can be used inside groups (see below) as well. As the target string is scanned, REs separated by '|' are tried from left to right. When one pattern completely matches, that branch is accepted. This means that once A matches, B will not be tested further, even if it would produce a longer overall match. In other words, the '|' operator is never greedy. To match a literal '|', use \|, or enclose it inside a character class, as in [|].