

Augmented assignment/ Assignment operators

<code>a = b</code>	Standard variable assignment
<code>a += b</code>	Equivalent to <code>a = a + b</code>
<code>a -= b</code>	Equivalent to <code>a = a - b</code>
<code>a *= b</code>	Equivalent to <code>a = a * b</code>
<code>a /= b</code>	Equivalent to <code>a = a / b</code>
<code>a %= b</code>	Equivalent to <code>a = a % b</code>
<code>a **= b</code>	Equivalent to <code>a = a ** b</code>
<code>a //= b</code>	Equivalent to <code>a = a // b</code>

Augmented assignment: Python supports augmented assignment for common arithmetic and logical operators.

Note: This is not an exhaustive list.

Input and output

<code>input()</code>	The input function prints text and expects a value from the user (string typed by user). Type functions (e.g. <code>int()</code>) can be used around it to get only certain types of values.
<code>print()</code>	The print function can print any number of expressions (separated by commas). Successive print statements will display on separate lines. A bare print will print a blank line.

General variable declaration and assignment

Declaration and initial assignment:

```
var_name = new_value
var_name = 3 ** new_value
var_name = other_var
```

General variable declaration and assignment (cont)

etc.

Assignment statements involving initial variable value:

```
var_name = 2 * var_name
var_name = var_name ** 2 + 5
```

Simultaneous assignment:

```
var_1, var_2... = value1, value2...
```

Variables: The basic mechanism by which data is organised and stored (long-term, short-term, and communication etc.). Variables must be declared before referred to in other statements.

Note: Variables can be reassigned as many times as needed.

General for loops

```
for variable in sequence_name:
    code body

for variable in [var1, var2, var3...]:
    code body

for variable in [const1, const2...]:
    code body

for variable in range(...):
    code body
```

For loops: A type of definite iterations. Also reference to as control structures.

Loop index: The variable after the for is called the loop index. It takes on each successive value in sequence.

Dealing with Strings

"String" [<i>index</i>]	String indexing, accesses an individual character found at the specified index in the string.
"String"- [<i>index1</i> : <i>index2</i>]	String slicing, returning a substring of the original string between the specified indexes
"Stringa" + "String-b"	String concatenation, achieved with the + operator and puts the multiple strings together
"String" * <i>int</i>	Repetition, returns the same string repeated a specified number of times in the same new string
<code>len("String")</code>	Finds the character length of a string
for <i>var</i> in "String"	Iterates through all the characters in a string
"str- ing".upper() OR "str- ing".lower()	The upper() function changes all characters to uppercase/lowercase



By **solisoleille** (soleille01)
cheatography.com/soleille01/

Not published yet.
Last updated 3rd August, 2022.
Page 1 of 5.

Sponsored by **Readable.com**
Measure your website readability!
<https://readable.com>

Dealing with Strings (cont)

`ord("c")` The `ord()` function returns the numeric (ordinal) code of a single character, the `chr()` function converts a numeric code to the corresponding character

`\n` Prints to a new line in a string

`"""` Multiline strings, using three double quotes on each side of the text

Strings: Strings are used to represent a sequence of characters, such as: names, addresses, general text etc. They are written in double quotes.

Slicing: The the substring starts at index1 but the last character is at index2-1. The indexes given must both be ints.

Slicing: If either start or end expression is missing, then the start or the end of the string is used.

Other: Not an exhausted list of functions, other useful ones include `strip()`, `count()`, `find()` and `split()` etc.

Searching

Simple searching:

```
wanted_value in list_name - tests
for list membership
list_name.index(wanted_value) - to find the position
```

Searching (cont)

Linear search:

```
for i in range(len(list_name)):
    if list_name[i] == wanted_value:
        return i
return None
```

Binary search:

```
low = 0
high = len(list_name) - 1
while low <= high:
    mid = (low + high) // 2
    item = list_name[mid]
    if wanted_value == item:
        return mid
    if wanted_value < item:
        high = mid - 1
    if wanted_value > item:
        low = mid + 1
return None
```

Simple searching: The problem with this is that the index method raises an exception if the sought item is not present.

Linear search: As soon as a value is encountered that is greater than the target value, the linear search can be stopped without looking at the rest of the data.

Binary search: If the data is already sorted, at each step divide the remaining group of numbers into two parts and ignore the irrelevant one

Dealing with tuples

Ex. `(value1, value2, value3, etc.)`

Sorting by element:

```
list_name.sort(key=lambda x: x[element_index])
```

Tuple: A sequence which looks like a list but uses `()` rather than `[]`. They are immutable, so are used to represent sequences that are not supposed to change.

Lambda function: A small anonymous function which can take any number of arguments, but can only have one expression.

Arithmetic operators

- +** Addition - adds together two values.
- Subtraction - subtracts one value from another.
- *** Multiplication - multiplies two values together.
- /** Floating point division - divides one value by another. The return value is exact (for floating point)
- //** Integer/Floor division - divides one value by another. The remainder is truncated.
- **** Exponentiation - raises a number to the power of another number.
- %** Modulus - returns the remainder of dividing a number with another number.

Arithmetic operators: Used to perform common mathematical operations.

Precedence: Precedence and associativity are as normal as in maths.

Comments

Single line comments:

```
# Comment1
```

```
code # Comment2
```

Multi-line comments:

```
" " "
```

```
Comment
```

```
Continuing comment
```

```
" " "
```

Comments: Comments are ignored by the computer, they exist simply to make the code easier for people to understand.

Dealing with lists

Creating a list (ex.):

```
list_name = [1, "Spam", 3.142, True]
```

```
months = ["Jan", "Feb", "Mar", "Apr",  
..etc.]
```

Indexing/slicing lists:

```
list_name[index] OR
```

```
list_name[index1:index2]
```

Some methods:

```
list_name.append(new_item)
```

```
list.index(object) (returns index of first occurrence)
```

```
list_name.min() and list_name.max()
```

```
list_name.reverse()
```

```
len(list_name)
```

```
list_1 + list_2
```

```
var in list_name: etc.
```

```
list_name.sort()
```

```
list_name.remove(object_to_remove) but the test for this
```

```
list_name.pop(index)
```

Lists: Lists are sequences of arbitrary values enclosed in square brackets. They can hold any datatype.

Mutable: Lists are mutable, meaning they can be changed. Strings can not be changed.

Note: Not an exhaustive list

TBC check

Numeric data types

int Represents whole numbers/integers. Can be positive or negative

float Represents numbers that can have fractional parts - floating point values. (Even if the fractional part is 0)

Note: The float type stores only an approximation to the real number being represented.

Note: Operations on ints produce ints (excluding /), operations on floats produce floats.

Type conversion: Combining an int with a float in an expression will return a float. And we can use the int and round functions to explicitly convert between different types. Converting a float to an int will truncate.

Type: We can use the type function to find the data type.

General if -elif-else-statements

```
if boolean_condition:
```

```
statements to execute if  
condition is True
```

```
elif boolean_n_c and itino
```

```
do these statements if the  
if-statement and elif
```

```
-statements above
```

```
returned False,
```

```
but the test for this  
statement returned True.
```

```
else:
```

```
do these statements if  
none of the above tests returned  
True.
```

if statements: The condition statement is evaluated and if it evaluates to True, the indented statements in the body are executed; otherwise, execution proceeds to next statement.

Note: Don't forget the colon!

General while loops

```
while boolean_condition:  
code body
```

While loop: A form of indefinite/conditional iteration loop. It keeps iterating until the boolean condition is no longer true.

Break statement

```
loop decl:  
code body etc.  
if boolean_n_c and -  
ition:  
break  
code body etc.
```

Break statement: Executing break cases Python to immediately exit the enclosing loop.

Note: It is sometimes used to exit what looks like an infinite loop.

Loop and a half: The loop exit is in the middle of the loop body. It is an elegant way to avoid the priming read in a sentinel loop.

Note: Avoid using break often within loops, because the logic of a loop is hard to follow when there are multiple exits.

Continue statement

```
loop decl:  
code body etc.  
if boolean_n_c and -  
ition:  
continue  
code body etc.
```

Continue statement: Returns the control to the beginning of the loop escaping the rest of the code body.

Recursion

```
def rec_func(n):
    base_case_condition:
        return value
    else:
        return
        computation * rec_func(n - 1)
    return rec_func(n - 1)
```

Recursion: A description of something that refers to itself is called a recursive definition.

Base case: Recursion is not circular because we eventually get to the base case that has a closed expression that can be directly computed.

Dealing with dictionaries (cont)

`dict_name.items()` to return tuples of all the key-value pairs
`dict_name.values()` to return all the values only
`dict_name.get(key, default)` if dictionary has the key return its value, otherwise returns default
`dict_name.setdefault(key, value)` if dictionary has the key do nothing, otherwise set it to value

Dictionary: Widely used collection/compound data type. Allows us to look up information associated with arbitrary keys (mapping)

Note: The order of the keys won't matter.

Logical/Boolean operators (cont)

or Returns True if at least one input is True

Precedence: The interpretation of the expressions relies on the precedence rules for the operators.

Range function

`range(stop)` (starts from 0 and goes up 1 until (stop - 1))
`range(start, stop)` starts from start and goes up 1 until (stop - 1)
`range(start, stop, step)` starts from start and goes by step (positive or negative) until (stop - 1)
`list(range(...))` makes a list

Dealing with dictionaries

Creating dictionaries:

```
dict_name = {}
dict_name = {key1: value1, key2: value2, key3: value3, ...}
```

Adding/initialising/changing key-value pairs:

```
dict_name[key] = new_value
```

Getting objects from keys:

```
dict_name[key] (if the dictionary does not have the key an exception is raised)
```

Some functions/operations:

`key in dict_name` to check if the key exists

`del dict_name[key]` to delete the entry corresponding to the key

`dict_name.pop(key)` to delete the entry and return the value

`dict_name.clear()` to delete all entries in the dictionary

`dict_name.keys()` to return all the key values

String formatting

```
"index :
width.precision,type".format(text)
Example:
"Count {0:0.2 0f} ".format(3.14)
-> 'Count 3.1400 000 000 000 -
001243'
```

Meanings: index - which parameter to insert into the slot; width tells us how many spaces to use to display the value; 0 means to use as much space as necessary; precision is the number of decimal places

Fixed point numbers: Denoted using f in the example.

Logical/Boolean operators

not Inverse the comparison result

and Returns True only if both inputs are True

Importing modules

Importing:

```
import module_name OR
import module_name as new_name
from module_name import function1
```

Calling functions:

```
module_name.function_name(...)
new_name.function_name(...)
function1(...)
```

Module: A file consisting of Python code which can define functions, classes, variables and may also include runnable code.

Note: When Python imports a module, it executes each line. Modules need to be imported in a session only once.

Library: A library is a module with some useful definitions/functions.

Defining a function

```
def func_name():
    code body

def func_name (parameter1,
parameter2 etc.):
    code body
```

Function: A function is a block of organised, reusable code that is used to perform a single, related action. It is invoked or executed by typing its name.

Parameters: Parameters can be used to customise the output of a function. A function that has parameters requires arguments. If that parameter is not specified an error is returned.

Relational/Comparison operators

==	Equal to
!=	Not equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

Relational operators: Operators used to compare two values - usually numbers, but also sometimes other types.

Precedence: All have lower precedence than all arithmetic operators, and higher than all logical operators.

File processing

Opening files:

```
file_var = open (file_name, mode)
(mode - 'r' (read), 'w' (write), or 'a' (append))
OR
```

File processing (cont)

with open(file_name) as file_var: *processing*
ng (when statements in that block have finished running, file will close automatically)
File methods:
file.read() - returns entire remaining contents as single string
file.readline() - returns next line of file. Also reads to and including next newline character
file.readlines() - returns list of remaining lines in file. Each list item is single line including newline characters
Efficient processing:
for line in infile: *processing*

File: A sequence of data that is stored in secondary memory (disk drive). They can contain any data type, and usually contains more than one line of text.

Note: When you've finished working with a file, it needs to be closed. In some cases, not properly closing a file could result in data loss.

Note: Multiple calls to readline() is inefficient.

Note: May use writelines() for writing sequence(list) of strings.

Exception handling

```
try:
    code body
except ErrorType:
    handler code
```

Try-except: When python encounters a try statement, it attempts to execute the try body. If an exception is raised, the handler is executed. If not, control passes to the statement after.

Note: There can be multiple except blocks. This acts like 'elif'.

Except: A bare except acts like an 'else' and catches any errors without a specific exception type.

Note: Exceptions are intended for exceptional circumstances and should not be used as a substitute for if statements.

PseudoRandom numbers

randint Randomly selects an integer value from a range. range() rules apply.

(start, stop, step)

choice(list) Chooses a random member of a given list.

random() Returns a random number in the range [0...1.0). (0 can be returned but not 1.0))

seed() Assign the random number generator a fixed starting point (to give reproducible behaviour during testing)

Pseudorandom number generator: Starts with a seed value to produce a "random" output. The next time a random number is required, the current value is fed back into the function to produce a new number.

Note: This sequence of numbers appears to be random, but if you start the process over again with the same seed number, you'll get the same sequence of "random" numbers.