Python basics for revision

Math Operators

From **highest** to **lowest** precedence:

Operators	Operation	Example
**	Exponent	`2 ** 3 = 8`
%	Modulus/Remainder	`22 % 8 = 6`
//	Integer division	`22 // 8 = 2`
/	Division	`22 / 8 = 2.75`
*	Multiplication	`3 * 3 = 9`
-	Subtraction	`5 - 2 = 3`
+	Addition	`2 + 2 = 4`

Examples of expressions:

```
>>> 2 + 3 * 6
# 20

>>> (2 + 3) * 6
# 30

>>> 2 ** 8
# 256

>>> 23 // 7
# 3

>>> 23 % 7
# 2
```

```
>>> (5 - 1) * ((7 + 1) / (3 - 1))
# 16.0
```

Augmented Assignment Operators

Operator	Equivalent
`var += 1`	`var = var + 1`
`var -= 1`	`var = var - 1`
`var *= 1`	`var = var * 1`
`var /= 1`	`var = var / 1`
`var //= 1`	`var = var // 1`
`var %= 1`	`var = var % 1`
`var **= 1`	`var = var ** 1`

Examples:

```
>>> greeting = 'Hello'
>>> greeting += ' world!'
>>> greeting
# 'Hello world!'

>>> number = 1
>>> number += 1
>>> number
# 2

>>> my_list = ['item']
>>> my_list *= 3
>>> my_list
# ['item', 'item', 'item']
```

Walrus Operator

The Walrus Operator allows assignment of variables within an expression while returning the value of the variable

Example:

```
>>> print(my_var:="Hello World!")
# 'Hello world!'

>>> my_var="Yes"
>>> print(my_var)
# 'Yes'

>>> print(my_var:="Hello")
# 'Hello'

# Without using the walrus operator
numbers = [5, 8, 2, 10, 3]
n = len(numbers)
if n > 0:
    print(f'The list has {n} elements.')

# With the walrus operator
numbers = [5, 8, 2, 10, 3]
if (n := len(numbers)) > 0:
    print(f'The list has {n} elements.')
```

The *Walrus Operator*, or **Assignment Expression Operator** was firstly introduced in 2018 via PEP 572, and then officially released with **Python 3.8** in October 2019.

Data Types

Data Type	Examples
Integers	`-2, -1, 0, 1, 2, 3, 4, 5`
Floating-point numbers	`-1.25, -1.0, --0.5, 0.0, 0.5, 1.0, 1.25`
Strings	`'a', 'aa', 'aaa', 'Hello!', '11 cats'`

Concatenation and Replication

String concatenation:

```
>>> 'Alice' 'Bob'
# 'AliceBob'
```

String replication:

```
>>> 'Alice' * 5
# 'AliceAliceAliceAliceAlice'
```

Variables

You can name a variable anything as long as it obeys the following rules:

1. It can be only one word.

```
>>> # bad
>>> my variable = 'Hello'

>>> # good
>>> var = 'Hello'
```

2. It can use only letters, numbers, and the underscore (_) character.

```
>>> # bad
>>> %$@variable = 'Hello'

>>> # good
>>> my_var = 'Hello'

>>> # good
>>> my_var_2 = 'Hello'
```

3. It can't begin with a number.

```
>>> # this wont work
>>> 23_var = 'hello'
```

4. Variable name starting with an underscore (_) are considered as "unuseful".

```
>>> # _spam should not be used again in the code
>>> _spam = 'Hello'
```

Comments

Inline comment:

```
# This is a comment
```

Multiline comment:

```
# This is a
# multiline comment
```

Code with a comment:

```
a = 1  # initialization
```

Please note the two spaces in front of the comment.

Function docstring:

```
def foo():
    """
    This is a function docstring
    You can also use:
    ''' Function Docstring '''
    """
```

The print() Function

The `print()` function writes the value of the argument(s) it is given. [...] it handles multiple arguments, floating point-quantities, and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely:

```
>>> print('Hello world!')
# Hello world!

>>> a = 1
>>> print('Hello world!', a)
# Hello world! 1
```

Use a backslash () to continue a statement to the next line

```
>>> total=1+2+3+4+5+6+7+\
4+5+6

>>> print(total)
# 43
```

Multiple Statements on a single line

```
x=5;y=10;z=x+y
print(z)
# 15
```

The end keyword

The keyword argument **end** can be used to avoid the newline after the output, or end the output with a different string:

```
phrase = ['printed', 'with', 'a', 'dash', 'in', 'between']
>>> for word in phrase:
...     print(word, end='-')
...
# printed-with-a-dash-in-between-
```

The sep keyword

The keyword **sep** specify how to separate the objects, if there is more than one:

```
print('cats', 'dogs', 'mice', sep=',')
# cats,dogs,mice
```

The input() Function

This function takes the input from the user and converts it into a string:

```
>>> print('What is your name?')   # ask for their name
>>> my_name = input()
>>> print('Hi, {}'.format(my_name))
# What is your name?
# Martha
# Hi, Martha
```

`input()` can also set a default message without using `print()`:

```
>>> my_name = input('What is your name? ')  # default message
>>> print('Hi, {}'.format(my_name))
# What is your name? Martha
# Hi, Martha
```

It is also possible to use formatted strings to avoid using .format:

```
>>> my_name = input('What is your name? ')  # default message
>>> print(f'Hi, {my_name}')
# What is your name? Martha
# Hi, Martha
```

The len() Function

Evaluates to the integer value of the number of characters in a string, list, dictionary, etc.:

```
>>> len('hello')
# 5

>>> len(['cat', 3, 'dog'])
# 3
```

Test of emptiness example:

```
>>> a = [1, 2, 3]

# bad
>>> if len(a) > 0:  # evaluates to True
...     print("the list is not empty!")
...
# the list is not empty!
```

```
# good
>>> if a: # evaluates to True
...     print("the list is not empty!")
...
# the list is not empty!
```

The str(), int(), and float() Functions

These functions allow you to change the type of variable. For example, you can transform from an `integer` or `float` to a `string`:

```
>>> str(29)
# '29'

>>> str(-3.14)
# '-3.14'
```

Or from a `string` to an `integer` or `float`:

```
>>> int('11')
# 11

>>> float('3.14')
# 3.14
```

Python control flow

Control flow is the order in which individual statements, instructions, or function calls are executed or evaluated. The control flow of a Python program is regulated by conditional statements, loops, and function calls.

Comparison Operators

Operator	Meaning
==	Equal to
!=	Not equal to
<	Less than
>	Greater Than

Operator	Meaning
<=	Less than or Equal to
>=	Greater than or Equal to

These operators evaluate to True or False depending on the values you give them.

Examples:

```
>>> 42 == 42
True

>>> 40 == 42
False

>>> 'hello' == 'hello'
True

>>> 'hello' == 'Hello'
False

>>> 'dog' != 'cat'
True

>>> 42 == 42.0
True

>>> 42 == '42'
False
```

Boolean Operators

There are three Boolean operators: `and`, `or`, and `not`. In the order of precedence, highest to lowest they are `not`, `and` and `or`.

The `and` Operator's *Truth* Table:

Expression	Evaluates to
`True and True`	`True`
`True and False`	`False`
`False and True`	`False`
`False and False`	`False`

The `or` Operator's *Truth* Table:

Expression	Evaluates to
`True or True`	`True`
`True or False`	`True`
`False or True`	`True`
`False or False`	`False`

The `not` Operator's *Truth* Table:

Expression	Evaluates to
`not True`	`False`
`not False`	`True`

Mixing Operators

You can mix boolean and comparison operators:

```
>>> (4 < 5) and (5 < 6)
True

>>> (4 < 5) and (9 < 6)
False

>>> (1 == 2) or (2 == 2)
True
```

Also, you can mix use multiple Boolean operators in an expression, along with the comparison operators:

```
>>> 2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2
True
>>> 5 > 4 or 3 < 4 and 5 > 5
True
>>> (5 > 4 or 3 < 4) and 5 > 5
False
```

Explanation: - The first expression checks multiple conditions: - `2 + 2 == 4` evaluates to `True`. - `not 2 + 2 == 5` evaluates to `True` since `2 + 2 == 5` is `False`. - `2 * 2 == 2 + 2` evaluates to `True`. - Combining all with `and` results in `True`.

- In the second expression:
  - The sub-expression `3 < 4 and 5 > 5` evaluates to `False` (`True and False`).
  - `5 > 4` evaluates to `True`.
  - With `True or False`, the overall result is `True`.
- For the third expression:
  - The parenthetical expression `(5 > 4 or 3 < 4)` evaluates to `True` (`True or False`).
  - `5 > 5` evaluates to `False`.
  - Combining with `and`, `True and False` results in `False`.

if Statements

The `if` statement evaluates an expression, and if that expression is `True`, it then executes the following indented code:

```
>>> name = 'Debora'

>>> if name == 'Debora':
...     print('Hi, Debora')
...
# Hi, Debora

>>> if name != 'George':
...     print('You are not George')
...
# You are not George
```

The `else` statement executes only if the evaluation of the `if` and all the `elif` expressions are `False`:

```
>>> name = 'Debora'

>>> if name == 'George':
...     print('Hi, George.')
... else:
...     print('You are not George')
...
# You are not George
```

Only after the `if` statement expression is `False`, the `elif` statement is evaluated and executed:

```
>>> name = 'George'

>>> if name == 'Debora':
...     print('Hi Debora!')
... elif name == 'George':
...     print('Hi George!')
...
# Hi George!
```

the `elif` and `else` parts are optional.

```
>>> name = 'Antony'

>>> if name == 'Debora':
...     print('Hi Debora!')
... elif name == 'George':
...     print('Hi George!')
... else:
...     print('Who are you?')
...
# Who are you?
```

Ternary Conditional Operator

Many programming languages have a ternary operator, which define a conditional expression. The most common usage is to make a terse, simple conditional assignment statement. In other words, it offers one-line code to evaluate the first expression if the condition is true, and otherwise it evaluates the second expression.

```
<expression1> if <condition> else <expression2>
```

Example:

```
>>> age = 15

>>> # this if statement:
>>> if age < 18:
```

```
...     print('kid')
... else:
...     print('adult')
...
# output: kid

>>> # is equivalent to this ternary operator:
>>> print('kid' if age < 18 else 'adult')
# output: kid
```

Ternary operators can be chained:

```
>>> age = 15

>>> # this ternary operator:
>>> print('kid' if age < 13 else 'teen' if age < 18 else 'adult')

>>> # is equivalent to this if statement:
>>> if age < 18:
...     if age < 13:
...         print('kid')
...     else:
...         print('teen')
... else:
...     print('adult')
...
# output: teen
```

Switch-Case Statement

In computer programming languages, a switch statement is a type of selection control mechanism used to allow the value of a variable or expression to change the control flow of program execution via search and map.

The *Switch-Case statements*, or **Structural Pattern Matching**, was firstly introduced in 2020 via PEP 622, and then officially released with **Python 3.10** in September 2022. The PEP 636 provides an official tutorial for the Python Pattern matching or Switch-Case statements.

Matching single values

```
>>> response_code = 201
>>> match response_code:
...     case 200:
...         print("OK")
...     case 201:
...         print("Created")
...     case 300:
...         print("Multiple Choices")
...     case 307:
...         print("Temporary Redirect")
...     case 404:
...         print("404 Not Found")
...     case 500:
...         print("Internal Server Error")
...     case 502:
...         print("502 Bad Gateway")
...
# Created
```

Matching with the or Pattern

In this example, the pipe character (`|` or `or`) allows python to return the same response for two or more cases.

```
>>> response_code = 502
>>> match response_code:
...     case 200 | 201:
...         print("OK")
...     case 300 | 307:
...         print("Redirect")
...     case 400 | 401:
...         print("Bad Request")
...     case 500 | 502:
...         print("Internal Server Error")
...
# Internal Server Error
```

Matching by the length of an Iterable

```
>>> today_responses = [200, 300, 404, 500]
>>> match today_responses:
...     case [a]:
...             print(f"One response today: {a}")
...     case [a, b]:
...             print(f"Two responses today: {a} and {b}")
...     case [a, b, *rest]:
...             print(f"All responses: {a}, {b}, {rest}")
...
# All responses: 200, 300, [404, 500]
```

Default value

The underscore symbol (_) is used to define a default case:

```
>>> response_code = 800
>>> match response_code:
...     case 200 | 201:
...         print("OK")
...     case 300 | 307:
...         print("Redirect")
...     case 400 | 401:
...         print("Bad Request")
...     case 500 | 502:
...         print("Internal Server Error")
...     case _:
...         print("Invalid Code")
...
# Invalid Code
```

Matching Builtin Classes

```
>>> response_code = "300"
>>> match response_code:
...     case int():
...             print('Code is a number')
...     case str():
```

```
...             print('Code is a string')
...     case _:
...             print('Code is neither a string nor a number')
...
# Code is a string
```

Guarding Match-Case Statements

```
>>> response_code = 300
>>> match response_code:
...     case int():
...             if response_code > 99 and response_code < 500:
...                 print('Code is a valid number')
...     case _:
...             print('Code is an invalid number')
...
# Code is a valid number
```

while Loop Statements

The while statement is used for repeated execution as long as an expression is `True`:

```
>>> spam = 0
>>> while spam < 5:
...     print('Hello, world.')
...     spam = spam + 1
...
# Hello, world.
# Hello, world.
# Hello, world.
# Hello, world.
# Hello, world.
```

break Statements

If the execution reaches a `break` statement, it immediately exits the `while` loop's clause:

```
>>> while True:
...     name = input('Please type your name: ')
...     if name == 'your name':
...         break
...
>>> print('Thank you!')
# Please type your name: your name
# Thank you!
```

continue Statements

When the program execution reaches a `continue` statement, the program execution immediately jumps back to the start of the loop.

```
>>> while True:
...     name = input('Who are you? ')
...     if name != 'Joe':
...         continue
...     password = input('Password? (It is a fish.): ')
...     if password == 'swordfish':
...         break
...
>>> print('Access granted.')
# Who are you? Charles
# Who are you? Debora
# Who are you? Joe
# Password? (It is a fish.): swordfish
# Access granted.
```

For loop

The `for` loop iterates over a `list`, `tuple`, `dictionary`, `set` or `string`:

```
>>> pets = ['Bella', 'Milo', 'Loki']
>>> for pet in pets:
...     print(pet)
...
# Bella
# Milo
# Loki
```

The range() function

The `range()` function returns a sequence of numbers. It starts from 0, increments by 1, and stops before a specified number:

```
>>> for i in range(5):
...     print(f'Will stop at 5! or 4? ({i})')
...
# Will stop at 5! or 4? (0)
# Will stop at 5! or 4? (1)
# Will stop at 5! or 4? (2)
# Will stop at 5! or 4? (3)
# Will stop at 5! or 4? (4)
```

The `range()` function can also modify its 3 defaults arguments. The first two will be the `start` and `stop` values, and the third will be the `step` argument. The step is the amount that the variable is increased by after each iteration.

```
# range(start, stop, step)
>>> for i in range(0, 10, 2):
...    print(i)
...
# 0
# 2
# 4
# 6
# 8
```

You can even use a negative number for the step argument to make the for loop count down instead of up.

```
>>> for i in range(5, -1, -1):
...     print(i)
...
# 5
# 4
# 3
# 2
# 1
# 0
```

For else statement

This allows to specify a statement to execute in case of the full loop has been executed. Only useful when a `break` condition can occur in the loop:

```
>>> for i in [1, 2, 3, 4, 5]:
...     if i == 3:
...         break
... else:
...     print("only executed when no item is equal to 3")
```

Ending a Program with sys.exit()

`exit()` function allows exiting Python.

```
>>> import sys

>>> while True:
...     feedback = input('Type exit to exit: ')
...     if feedback == 'exit':
...         print(f'You typed {feedback}.')
...         sys.exit()
...
# Type exit to exit: open
# Type exit to exit: close
# Type exit to exit: exit
# You typed exit
```

```
## number even ,odd, negative

# num=int(input("Enter the number"))
num = 3

if num>0:
    print("The number is positive")
    if num%2==0:
        print("The number is even")
    else:
        print("The number is odd")

else:
    print("The number is zero or negative")
```

```
The number is positive
The number is odd
```

```python
# Determine if a year is a leap year
# year = int(input("Enter the year: "))
year = 2024

# Determine if the year is a leap year
if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
    print(year, "is a leap year")
else:
    print(year, "is not a leap year")
```

```
2024 is a leap year
```