Python / Concept

Scripting Language

The interpreter executes source code directly without compiling it into an executable program, resulting in lower efficiency and execution performance.

Cross-Platform

Python is open-source, and the same Python code can run on different platforms as long as a compatible interpreter is available.

Identifiers

Composed of letters, underscores, and numbers, but cannot start with a number.

Python / Core

Constants and Variables

```
# coding=utf-8  # Declare file encoding
```

Numeric types

Example:

```
a = 999             # Decimal
a, b = 100, 200     # Multiple assignment
a = 0b1000          # Binary (8 in decimal)
a = 0o1000          # Octal (512 in decimal)
a = 0x189           # Hexadecimal (393 in decimal)
a = 3 + 4j          # Complex number (real=3, imag=4)
a = 3.0 + 4.0j      # Complex with float parts
```

Usage:

a/2

Return a float result (e.g., 100 / 2 gives 50.0).

a//2

Performs floor division and returns an integer if both operands are integers

Float

Size:

Typically 8 bytes (64 bits)

Range

Approximately: ±1.7 × 10^(-308) to ±1.7 × 10^(308)

(Depends on platform and implementation, usually IEEE 754 double precision)

Example:

```
a=1000.0
```

String

Size:

Depends on content and platform (internally variable-length).

Example:

```
a = "John"              # Double quotes
```

```
a = 'John'                # Single quotes (There is no functional difference between 'John' and "John")
a, b = "John", "Alice"     # Multiple assignment
a = "ab" + "c" * 2         # Result: 'abcc'
a = "abcdef"[2:5]          # Substring: 'cde' (index 5 excluded)
a = b'a\x01c'              # Bytes string: [97, 1, 99]
a=u'sp\xc4m'               # Unicode string: 'spÄm'
a=r'C:\Desktop'            # Raw string to ignore escape sequences, Output: 'C:\\Desktop'
a = f'xx{val}xx'           # Formatted String, Output: 'xx123xx'
```

Usage:

"world" in my_str

Check if "world" is a substring of my_str → returns True or False.

"\n".join(my_list)

Join elements of a list with a newline character.

"a {0} b {1}".format(name ,age)

Positional formatting using indices.

"a {name} b {url}".format(**my_dict)

Format using dictionary keys with keyword arguments.

"a {0[0]} b {0[1]}".format(my_list)

Access list elements by index (the 0 refers to the first argument).

"John %s like %s" %('repstr1', 'replace2')

Old-style formatting using % operator.

"Hey %(name) s, there is a 0x%(errno)x error!" %{ "name": name, "errno": errno }

Dictionary-based old-style formatting.

Boolean

Size:

Internally represented as integers: True = 1, False = 0

Range

Only two possible values: True, False

Example:

```
a=True          # Assign True
a,b=True,False  # Multiple assignment
a=False         # Assign False
```

List

In Python, lists do not automatically expand when you assign to an index that is out of range

Example:

```
a=[]
a,b=[],[]
a=[[1, 3, 4], [2, 3, 5], [1, 2, 3, 5], [2, 5]]
a=[0] * 10   # Creates a list with 10 zeros
```

Usage:

`value = a[3]`

Access list value

`a[3] = 9`

Change list value

`a[-1] = 9`

Change last value

```
a = [1, 2, 2, 3]
a[-1] = 9
print(a)  # Output: [1, 2, 2, 9]
```

`a[999]`

IndexError: list index out of range

`sublist = a[2:5]`

Get elements from index 2 to index 4 (5 is exclusive)

```
a = [1, 2, 2, 3]
sublist = a[2:5]
print(sublist) # Output: [2, 3]

a = [1, 2, 2, 3]
sublist = a[-1:5]
print(sublist) # Output: [3]

a = [1, 2, 2, 3]
sublist = a[-1:2]
print(sublist) # Output: []

a = [1, 2, 2, 3]
sublist = a[3:2]
print(sublist) # Output: []
```

`a.append(10)`

Extend the list, adds 10 at the end of the list

`a.extend([4, 5])`

Extend the list, adds 4 and 5 to the end of the list

`a.insert(1, 10)`

Inserts 10 at index 1, shifting other elements

Time Complexity: O(n), where n is the number of elements in the list.

```
arr = [1, 2, 3]
arr.insert(5, 10)
print(arr)  # Output: [1, 2, 3, 10]

arr = [1, 2, 3]
arr.insert(1, 10)
print(arr)  # Output: [1, 10, 2, 3]
```

`if 3 in arr:`

Check if 3 is in the array

`if 3 in numDict.get(val):`

If numDict.get(val) returns None, then Python will raise a TypeError because None is not iterable, and the in operator cannot be used to check membership in None.

`if val in numDict and 3 in numDict[val]:`

Do something if 3 is in the value of the given key

Dictionary

Example:

```
a={}  #  {} is the syntax for an empty dictionary in Python, not a set
a,b={},{}
a={"name":"lala","age":12 }
a=dict(hours=10)
a=dict()
```

Usage:

```
value = a['name']          # Access dictionary value
a["city"] = "New York"     # Add a new key-value pair
a.get("city", "empty")      # Return a default value ( None if not present)
"city" in numDict           # Check if "city" is a key in numDict.
```

Set

Example:

```
a={'a',  'b',  'c'}
a,b=set(),set()
a=set( 'abc' )
```

Tuple

Example:

```
a=(1,  'spam',  4,  'U')
a,b=tuple(),tuple()
a=tuple('spam')
```

Usage:

```
b, c, d, e = a      # Unpacking tuple
val = a[1:5]        # Slice tuple from index 1 to 4 (exclusive)
val = a[3]          # Access third element
```

Data Type Conversion

int(x)
: Converts x to an integer

long(x)
: Converts x to a long integer (Python 2.x only).

float(x)
: Converts x to a floating-point number.

complex(real [,imag])
: Creates a complex number with real and an optional imag part.

str(x)
: Converts x to a string.

repr(x)
: Returns a string representation of x that could be used as a valid Python expression.

eval("expression")
: Evaluates the Python expression in a string and returns the result.

tuple(s)
: Converts sequence s to a tuple.

list(s)
: Converts sequence s to a list.

chr(68)
: Converts an integer to its corresponding character (e.g., chr(68) returns 'D').

unichr(2)
: Converts an integer to a Unicode character (Python 2.x only).

ord('x')

Converts a character to its corresponding integer value (e.g., ord('x') returns 120).

hex(99)

Converts an integer to its hexadecimal string representation (e.g., hex(99) returns '0x63').

oct(88)

Converts an integer to its octal string representation (e.g., oct(88) returns '0o130').

bytearray(b'\x01\x02\x03')

Returns a new bytearray object.

Type Checking

type(x)

Returns the type of x (does not consider subclass relationships).

// type(a) == str checks if a is exactly of type str.

isinstance(x, y)

Checks if x is an instance of y, considering subclass relationships.

// isinstance(1, int) returns True.

// isinstance(1.0, float) returns True.

// isinstance("xxx", str) returns True.

Operators

Notes:

The ++ increment operator is not valid in Python, unlike in languages like C or Java.

9 // 8

Performs integer division (returns an integer result, i.e., 1).

pass

A placeholder for an empty function or block, effectively a no-op.

for

```
for val in list:
    print(val)
    # Iterates through the list, getting each value.

for ind in range(len(list)):
    # Iterates through indices of a list.

for ind, val in enumerate( sequence ):
    # Iterates through both indices and values of a sequence.

[print(val) for val in list]
    # To loop through a list and perform an action (e.g., print values)
    # You cannot use the assignment = inside a generator expression directly.

[print(val) for val in list if val > 10]
    # To loop through a list and perform an action only on certain it

new_list = [item * 2 for item in list]
    # To loop through a list and modify another list
```

```
(func(val) for val in list)
    # To loop without storing results in memory as a list (generating items lazily)

for key in dict1:
    print(key)
    # Iterates through the dictionary, getting each key

for key in dict1.keys():
    print(key)
    # Iterates through the dictionary, getting each key

for key in dict1.values():
    print(key)
    # Iterates through the dictionary, getting each value

for key, val in dict1.items():
    print(key)
    print(val)
    # Iterating over key-value pairs.

dict2 = {key: val for key, val in dict1.items() if value > 10}
    # Create a new dictionary with values greater than 10
```

while

```
while not (condition):
    print("while")
    # The loop continues until the condition is true.
```

if

```
if val is None:
    print(val)
    # Checks if val is not None using a negated condition.
    # In Python, is checks object identity, not value equality. That means:
    # val is 23 checks if val is the exact same object in memory as the integer 23.
    # val == 23 checks if val has the same value as 23
elif not val:
    print(val+1)
    # Checks if val is falsy (e.g., None, False, empty strings "", 0, empty list [], empty dictionary {}, empty tuple ()).
elif val > 3:
    print(val+2)
else:
    print(val+3)
    # Checks if val is exactly None.
    # Example:
    #     val is not None and val > 2
    #     val is None or val > 2
```

```
    #     numDict.get(val) is not None and left != -1
```

a = a **if** a>b **else** b

```
    # Returns a if a > b; otherwise, returns b.
```

Type Hints (introduced in Python 3.5)

Basic Types

```
x: int = 42
y: float = 3.14
z: str = "hello"
is_active: bool = True
```

Union (Multiple Allowed Types)

```
from typing import Union
num: Union[int, float] = 10  # Can be either int or float
```

In Python 3.10+, you can use | instead:

```
num: int | float = 10
```

Optional (Allows None)

```
from typing import Optional
name: Optional[str] = None  # Equivalent to Union[str, None]
```

In Python 3.10+:

```
name: str | None = None
```

Collection Types

Lists, Tuples, Sets, Dicts

```
from typing import List, Tuple, Set, Dict

numbers: List[int] = [1, 2, 3]
coords: Tuple[float, float] = (10.5, 20.2)
tags: Set[str] = {"python", "typing"}
user: Dict[str, int] = {"age": 25}
```

Python 3.9+ uses built-in generics instead:

```
numbers: list[int] = [1, 2, 3]
user: dict[str, int] = {"age": 25}
```

Callable (Functions)

```
from typing import Callable

def add(x: int, y: int) -> int:
    return x + y

operation: Callable[[int, int], int] = add
```

Any (Disables Type Checking)

```
from typing import Any

data: Any = "string"
data = 42  # Allowed
```

Custom Classes

```
class User:
    def __init__(self, name: str):
        self.name = name

user: User = User("Alice")
```

Type Aliases

```
from typing import Union

Number = Union[int, float]
age: Number = 30
```

Generics (For Reusable Types)

```
from typing import TypeVar, Generic

T = TypeVar("T")

class Box(Generic[T]):
    def __init__(self, content: T):
        self.content = content

int_box = Box   # Box containing an int
```

Self-Referencing Types (Type and Self)

```
from typing import Type, Self

class Animal:
    def create(cls: Type["Animal"]) -> "Animal":
        return cls()

    def copy(self) -> Self:
        return self
```

```
class SegmentTree:
   def __init__(self, left: "SegmentTree" = None):
      self.left = left
```

Literal (Fixed Values)

```
from typing import Literal

status: Literal["success", "failure"] = "success"
```

Python / Features

Function

Usage

```
def func(a, b=9, c=None, *rest):
    # *rest: Collects extra positional arguments into a tuple.
    global globalval
    globalval += 1
        # This modifies the global variable 'globalval'
        # When modifying a global variable inside a function, you must declare it as global.
        # Otherwise, Python treats it as a local variable and raises an error if accessed as global later.
    return a+b

func(1, 2, 3, 4)
```

```
    # Passing arguments, 4 will go into *rest

def func(a, b, **rest):
    # **rest: Collects extra keyword arguments into a dictionary.
  return a+b

func(1, 2, name="craig", age=12 )
    # The keyword arguments are captured in the **rest dictionary
func(a= 1, b= 2, name="craig", age=12 )
```

Class

Definition

```
class Person:
    # Class content

class Son(Person):                  # Inherit from Person
   def __init__(self, name, age):
      Person.__init__(self)         # Call parent class constructor
      self.name = name
      self.age = age

per = Person("aa", 22)              # Create an instance
```

Fields

```
money=99.9;          # static members  (it can be accessed directly by the class name: Person.money)
```

Constructor

```
def __init__(self, name, age):          # Constructor (called when an instance is created)
    self.name = name
    self.age = age
    Person.money += 1
```

Methods

```
def func(self):                         # Instance method
    print(self.name, self.age)
    self.func2()                         # Calling another instance method

@staticmethod
def statfunc( a ):              # Static method, can be called as Person.statfunc() (does not require `self`)
    print(Person.money)

@classmethod                    # Class method (can be called without instantiating the class)
def clsfunc(cls):
    print(cls.money)
    cls().func()                # Calls an instance method
```

Built-in Attributes

Person.__dict__
    Dictionary containing class attributes
Person.__doc__
    Class documentation string
Person.__name__
    Class name
Person.__module__
    Module where the class is defined
    Example: If className is in module mymod, then className.__module__ == "mymod"
Person.__bases__
    Tuple containing all parent classes

Instance Attribute Access

hasattr(per, 'age')
    Returns True if the attribute 'age' exists
getattr(per, 'age')
    Retrieves the value of 'age'
setattr(per, 'age', 8)
    Sets the value of 'age' to 8
delattr(per, 'age')
    Deletes the attribute 'age'

Exception

```
# try block contains code that might throw an exception.
try:
    raise Exception("aaa")    # Raise an exception
# except block catches and handles exceptions of the specified type (Exception in this case).
except Exception:
    print("aaa")
# finally block executes no matter what, even if an exception was raised or not.
finally:
    print("final")
```

Import package

Absolute Import

```
from <module-name> import a, b          # Absolute import
```

Imports a and b from <module-name>.
Python first checks if a and b are variables in the __init__.py file of the package.
Then it checks if <module-name> is a subpackage or module, and raises an ImportError if not found.

Relative Import

```
from ...<module-name> import *          # Relative import
```

The . refers to the current directory, and each additional . refers to the parent directory.
For example, ... moves up two levels in the directory structure.

Python / Build-in Libraries

Global

__name__

It can be used to check if the file is being run directly or imported.

If the file is run directly, __name__ is set to "__main__".

If the file is imported as a module, __name__ will be set to the module's name.

```
if __name__ == "__main__":
    print("This script is being run directly")
```

__file__

The path of the current Python file. It can be an absolute or relative path,depending on how the script is executed.

```
print(__file__)  # Prints the relative or absolute path of the current script
```

input("str")

Prompts the user for input

print("str")

Prints "str"

range(2, 6)

Range from 2 to 5 (6 is not included)

```
for i in range(2, 6):
    print(i)  # Output: 2 3 4 5
```

range(0, 10, 2)

Range from 0 to 10 with a step of 2

```
for i in range(0, 10, 2):
    print(i)
```

sorted(iterable)

Work on any iterable (such as lists, tuples, strings, etc.) and returns a new sorted list, leaving the original iterable unchanged.

Time complexity:

- Best case: O(n), when the list is already sorted.
- Average case: O(nlogn), which is the typical case for most sorting tasks.
- Worst case: O(nlogn), even in the worst scenario where the list is in reverse order.

bisect

bisect.bisect_right(a, x, lo=0, hi=len(a))

Return the index where to insert item x in list a, assuming a is sorted.

Find the first element that satisfies a[i] > x.

If x already appears in the list, i points just beyond the rightmost x already there.

Optional args lo (default 0) and hi (default len(a)) bound the slice of a to be searched.

Time Complexity: O(log n)

The list a is assumed to be sorted, and the methods use binary search to find the appropriate insertion point.

Binary search divides the list in half on each iteration, which leads to a time complexity of O(log n).

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

i = bisect.bisect_right(a, 2)
print(i)  # Output: 3
```

```
i = bisect.bisect_right(a, 2.5)
print(i)  # Output: 3

i = bisect.bisect_right(a, 999)
print(i)  # 4

i = bisect.bisect_right(a, -999)
print(i)  # 0


a = [3, 8]
b = [3, 3, 8, 8]
c = [3, 3, 5, 8, 8]

L, R = bisect_right(a, val1), bisect_left(a, val2)

Case 1 (val2 <= 3):
    [3, 8]
     L
     R
    [3, 3, 8, 8]
     L
     R

Case 2 (val1 >= 3, val2 <=8):
    [3, 8]
        L
        R
    [3, 3, 8, 8]
           L
           R
    [3, 3, 5, 8, 8]
           L  R

Case 3 (val1 >= 3 and val1 < 8, val2 >=8):
    [3, 8]
        L
        R
    [3, 3, 8, 8]
           L
              R
    [3, 3, 5, 8, 8]
           L     R


Case 4 (val1 >=8):
    [3, 8]
        L
        R
    [3, 3, 8, 8]
              L
              R




[0, val1, val2, 4]  (val1 >= 0, val2 < 4)
val1 < val2 < a[L]   (a[L] = 4, a[R] = 4)


[0, val1, 4, val2]  (val1 >= 0, val2 > 4)
0 < val1 < val2 a[r]   (a[l] equals a[r])
```

bisect.bisect_left(a, x, lo=0, hi=len(a))

Return the index where to insert item x in list a, assuming a is sorted.

Find the first element that satisfies a[i] >= x.

So if x already appears in the list, i points just the leftmost x already there.

Time Complexity: O(log n)

The list a is assumed to be sorted, and the methods use binary search to find the appropriate insertion point.

Binary search divides the list in half on each iteration, which leads to a time complexity of O(log n).

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

i = bisect.bisect_left(a, 2)
print(i)  # Output: 1 (first 2)

i = bisect.bisect_left(a, 2.5)
print(i)  # Output: 3 (before 3, after 2)

i = bisect.bisect_left(a, 999)
print(i)  # Output: 4

i = bisect.bisect_left(a, -999)
print(i)  # Output: 0
```

bisect.insort_left(a, x, lo=0, hi=len(a))

Insert item x in list a, and keep it sorted assuming a is sorted.

If x is already present, it is inserted before the leftmost existing element.

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

bisect.insort_left(a, 2)
print(a)  # Output: [1, 2, 2, 2, 3] (inserts before first 2)

bisect.insort_left(a, 2.5)
print(a)  # Output: [1, 2, 2, 2, 2.5, 3] (inserts before 3, after 2s)

bisect.insort_left(a, 999)
print(a)  # Output: [1, 2, 2, 2, 2.5, 3, 999] (inserts at the end)

bisect.insort_left(a, -999)
print(a)  # Output: [-999, 1, 2, 2, 2, 2.5, 3, 999] (inserts at the beginning)
```

collections

Counter(list)

Return a frequency map of the list

```
// [1, 2, 2, 3, 3, 3, 4]  -> {3: 3, 2: 2, 1: 1, 4: 1}
   counter.update([1, 2, 2, 3])  ->  {3: 4, 2: 4, 1: 2, 4: 1}
```

Updates the counter with the elements from another iterable or dictionary.

```
// "hello world" -> {'l': 3, 'o': 2, 'h': 1, 'e': 1, ' ': 1, 'w': 1, 'r': 1, 'd': 1}
counter.values()     # Access frequency values
```

Time complexity: O(n)

list

Python < 3.9

```
from collections import Counter
from typing import List   # For Python < 3.9; otherwise, use list[int]
class Solution:
    def intersect(self, nums1: List[int], nums2: List[int]) -> List[int]:
```

Python 3.9+

```
from collections import Counter
class Solution:
    def intersect(self, nums1: list[int], nums2: list[int]) -> list[int]:
```

list.append(element)

Add an element to the end of the list.

list.insert(index, element)

Insert element x at index i.

list.count(value)

Return the number of times x appears in the list.

list.extend(iterable)

Append elements from another iterable to the end of the list.

list.index(value, start=0, end=len(list))

Return the index of the first occurrence of x.

list.sort()

Sort the list in-place and returns None.

Time complexity:

- Best case: O(n), when the list is already sorted.
- Average case: O(nlogn), which is the typical case for most sorting tasks.
- Worst case: O(nlogn), even in the worst scenario where the list is in reverse order.

Space Complexity: O(n)

This space is used to store temporary arrays during the merging phase.

tuple

Tuples are immutable — their elements cannot be changed after creation.

cmp(tup1, tup2)

Compare two tuples element by element. (Note: Only available in Python 2)

len(tup)

Return the number of elements in the tuple.

max(tup)

Return the maximum value in the tuple.

min(tup)

Return the minimum value in the tuple.

tuple(seq)

Convert a sequence (like a list) into a tuple.

index(val)

Return the index of the first occurrence of val in the tuple.

count(val)

Return the number of times val appears in the tuple.

str

str.replace(old, new, count)

Returns a copy of the string with all occurrences of the substring old replaced by new.

You can optionally specify a maximum number of replacements with the count argument.

str.split(separator=None, maxsplit=-1)

Split a string by a separator; maxsplit limits the number of splits.

str.rstrip([chars])

Remove trailing newline or other specified characters.

str.upper()

Convert all characters to uppercase.

str.lower()

Convert all characters to lowercase.

str.capitalize()

Capitalize the first character; lowercase the rest.

str.title()

Returns a copy of the string with the first letter of each word capitalized and the rest in lowercase. Words are defined by whitespace.

Example:

```
s = "hello world"
new_s = s.title()
print(new_s)  # "Hello World"
```