1. PYTHON CALL REFERENCE

Quick reference to some common classes, functions and methods of Python and its popular modules, without examples.

For quick reference, each description only has one sentence, see the documentations provided for more details.

Some sections does not fully explain all the parameters provided.

Written by Hoang Tran Nhat Minh,

Hanoi University of Science and Technology,

Data Science and Artificial Intelligence - K65.

2. Built-in Functions

Doc: https://docs.python.org/3/library/functions.html

2.1. `abs()`

```
abs(x)
```

Return the absolute value of a number.

2.2. `bin()`

```
bin(x)
```

Convert an integer number to a binary string prefixed with "`0b`".

2.3. `complex()`

```
class complex([real[, imag]])
```

Return a complex number with the value `real` + `imag` *1j or convert a string or number to a complex number.

2.4. `dict()`

```
class dict(**kwarg)
```

```
class dict(mapping, **kwarg)
```

```
class dict(iterable, **kwarg)
```

Create a new dictionary.

2.5. `dir()`

```
dir([object])
```

With an argument, attempt to return a list of valid attributes for that `object`.

2.6. `divmod()`

```
divmod(a, b)
```

Take two (non complex) numbers as arguments and return a pair of numbers consisting of their quotient and remainder when using integer division.

2.7. `enumerate()`

```
enumerate(iterable, start=0)
```

Return an enumerate object.

2.8. `eval()`

```
eval(expression[, globals[, locals]])
```

The expression argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the globals and locals dictionaries as global and local namespace.

2.9. `float()`

```
class float([x])
```

Return a floating point number constructed from a number or string `x`.

2.10. `format()`

```
format(value[, format_spec])
```

Convert a value to a “formatted” representation, as controlled by `format_spec`.

2.11. `frozenset()`

```
class frozenset([iterable])
```

Return a new `frozenset` object, optionally with elements taken from `iterable`.

2.12. `help()`

```
help([object])
```

Invoke the built-in help system.

2.13. `hex()`

```
hex(x)
```

Convert an integer number to a lowercase hexadecimal string prefixed with "`0x`".

2.14. `input()`

```
input([prompt])
```

The function reads a line from input, converts it to a string (stripping a trailing newline), and returns that.

2.15. `int()`

```
class int([x])
```

```
class int(x, base=10)
```

Return an integer object constructed from a number or string `x`, or return `0` if no arguments are given.

2.16. `isinstance()`

```
isinstance(object, classinfo)
```

Return `True` if the `object` argument is an instance of the `classinfo` argument, or of a (direct, indirect or virtual) subclass thereof.

2.17. `issubclass()`

```
issubclass(class, classinfo)
```

Return `True` if `class` is a subclass (direct, indirect or virtual) of `classinfo`.

2.18. `len()`

```
len(s)
```

Return the length (the number of items) of an object.

2.19. `list()`

```
class list([iterable])
```

Rather than being a function, `list` is actually a mutable sequence type.

2.20. `map()`

```
map(function, iterable, ...)
```

Return an iterator that applies `function` to every item of `iterable`, yielding the results.

2.21. `max()`

```
max(iterable, *[, key, default])
```

```
max(arg1, arg2, *args[, key])
```

Return the largest item in an iterable or the largest of two or more arguments.

2.22. `min()`

```
min(iterable, *[, key, default])
```

```
min(arg1, arg2, *args[, key])
```

2.23. `oct()`

```
oct(x)
```

Convert an integer number to an octal string prefixed with "`0o`".

2.24. `open()`

```
open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True
```

Open `file` and return a corresponding file object.

2.25. `ord()`

```
ord(c)
```

Given a string representing one Unicode character, return an integer representing the Unicode code point of that character.

2.26. `pow()`

```
pow(base, exp[, mod])
```

Return `base` to the power `exp`; if mod is present, return base to the power exp, modulo `mod` (computed more efficiently than `pow(base, exp) % mod`).

2.27. `print()`

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

Print `objects` to the text stream `file`, separated by `sep` and followed by `end`.

2.28. `range()`

```
class range(stop)
```

```
class range(start, stop[, step])
```

Rather than being a function, `range` is actually an immutable sequence type.

2.29. `reversed()`

```
reversed(seq)
```

Return a reverse iterator.

2.30. `round()`

```
round(number[, ndigits])
```

Return `number` rounded to `ndigits` precision after the decimal point.

2.31. `class()`

```
class set([iterable])
```

Return a new `set` object, optionally with elements taken from `iterable`.

2.32. `slice()`

```
class slice(stop)
```

```
class slice(start, stop[, step])
```

Return a slice object representing the set of indices specified by `range(start, stop, step)`.

2.33. `sorted()`

```
sorted(iterable, *, key=None, reverse=False)
```

Return a new sorted list from the items in `iterable`.

2.34. `str()`

```
class str(object='')
```

```
class str(object=b'', encoding='utf-8', errors='strict')
```

Return a `str` version of `object`.

2.35. `sum()`

```
sum(iterable, /, start=0)
```

Sums `start` and the items of an `iterable` from left to right and returns the total.

2.36. `super()`

```
super([type[, object-or-type]])
```

Return a proxy object that delegates method calls to a parent or sibling class of type, useful for accessing inherited methods that have been overridden in a class.

2.37. `tuple()`

```
class tuple([iterable])
```

Rather than being a function, `tuple` is actually an immutable sequence type.

2.38. `type()`

```
class type(object)
```

```
class type(name, bases, dict, **kwds)
```

With one argument, return the type of an object.

2.39. `zip()`

```
zip(*iterables)
```

Make an iterator that aggregates elements from each of the iterables.

3. Built-in Types

Doc: https://docs.python.org/3/library/stdtypes.html

3.1. Strings

3.1.1. `capitalize()`

```
str.capitalize()
```

Return a copy of the string with its first character capitalized and the rest lowercased.

3.1.2. `center()`

```
str.center(width[, fillchar])
```

Return centered in a string of length `width`.

3.1.3. `count()`

```
str.count(sub[, start[, end]])
```

Return the number of non-overlapping occurrences of substring `sub` in the range [`start`, `end`].

3.1.4. `endswith()`

```
str.endswith(suffix[, start[, end]])
```

Return `True` if the string ends with the specified `suffix`, otherwise return `False`.

3.1.5. `find()`

```
str.find(sub[, start[, end]])
```

Return the lowest index in the string where substring `sub` is found within the slice `s[start:end]`, else return `-1`.

3.1.6. `index()`

```
str.index(sub[, start[, end]])
```

Like `find()`, but raise `ValueError` when the substring is not found.

3.1.7. `join()`

```
str.join(iterable)
```

Return a string which is the concatenation of the strings in `iterable`.

3.1.8. `lower()`

```
str.lower()
```

Return a copy of the string with all the cased characters converted to lowercase.

3.1.9. `lstrip()`

```
str.lstrip([chars])
```

Return a copy of the string with leading characters removed.

3.1.10. `replace()`

```
str.replace(old, new[, count])
```

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

3.1.11. `rfind()`

```
str.rfind(sub[, start[, end]])
```

Return the highest index in the string where substring `sub` is found, such that `sub` is contained within `s[start:end]`.

3.1.12. `rindex()`

```
str.rindex(sub[, start[, end]])
```

Like `rfind()` but raises `ValueError` when the substring `sub` is not found.

3.1.13. `rstrip()`

```
str.rstrip([chars])
```

Return a copy of the string with trailing characters removed.

3.1.14. `split()`

```
str.split(sep=None, maxsplit=-1)
```

Return a list of the words in the string, using `sep` as the delimiter string.

3.1.15. `splitlines()`

```
str.splitlines([keepends])
```

Return a list of the lines in the string, breaking at line boundaries.

3.1.16. `startswith()`

```
str.startswith(prefix[, start[, end]])
```

Return `True` if string starts with the `prefix`, otherwise return `False`.

3.1.17. `strip()`

```
str.strip([chars])
```

Return a copy of the string with the leading and trailing characters removed.

3.1.18. `title()`

```
str.title()
```

Return a titlecased version of the string where words start with an uppercase character and the remaining characters are lowercase.

3.1.19. `upper()`

```
str.upper()
```

Return a copy of the string with all the cased characters converted to uppercase.

3.2. Sets

3.2.1. `issubset()`

```
issubset(other)
```

```
set <= other
```

Test whether every element in the set is in other.

3.2.2. `issuperset()`

```
issuperset(other)
```

```
set >= other
```

Test whether every element in other is in the set.

3.2.3. `union()`

```
union(*others)
```

```
set | other | ...
```

Return a new set with elements from the set and all others.

3.2.4. `intersection()`

```
intersection(*others)
```

```
set & other & ...
```

Return a new set with elements common to the set and all others.

3.2.5. `difference()`

```
difference(*others)
```

```
set - other - ...
```

Return a new set with elements in the set that are not in the others.

3.2.6. `symmetric_difference()`

```
symmetric_difference(other)
```

```
set ^ other
```

Return a new set with elements in either the set or other but not both.

3.2.7. `copy()`

```
copy()
```

Return a shallow copy of the set.

3.2.8. `update()`

```
update(*others)
```

```
set |= other | ...
```

Update the set, adding elements from all others.

3.2.9. `intersection_update()`

```
intersection_update(*others)
```

```
set &= other & ...
```

Update the set, keeping only elements found in it and all others.

3.2.10. `difference_update()`

```
difference_update(*others)
```

```
set -= other | ...
```

Update the set, removing elements found in others.

3.2.11. `symmetric_difference_update()`

```
symmetric_difference_update(other)
```

```
set ^= other
```

Update the set, keeping only elements found in either set, but not in both.

3.2.12. `add()`

```
add(elem)
```

Add element `elem` to the set.

3.2.13. `remove()`

```
remove(elem)
```

Remove element `elem` from the set or raise `KeyError` if `elem` is not contained in the set.

3.2.14. `discard()`

```
discard(elem)
```

Remove element `elem` from the set if it is present.

3.2.15. `pop()`

```
pop()
```

Remove and return an arbitrary element from the set or raises `KeyError` if the set is empty.

3.3. Tuples

3.3.1. `count()`

```
count(x)
```

Total number of occurrences of `x`.

3.3.2. `index()`

```
index(x[, i[, j]])
```

Index of the first occurrence of `x` (at or after index `i` and before index `j`)

4. Data Structures

Doc: https://docs.python.org/3/tutorial/datastructures.html

4.1. Lists

4.1.1. `append()`

```
list.append(x)
```

Add an item to the end of the list.

4.1.2. `extend()`

```
list.extend(iterable)
```

Extend the list by appending all the items from the iterable.

4.1.3. `insert()`

```
list.insert(i, x)
```

Insert an item at a given position.

4.1.4. `remove()`

```
list.remove(x)
```

Remove the first item from the list whose value is equal to `x`.

4.1.5. `pop()`

```
list.pop([i])
```

Remove the item at the given position in the list, and return it; if no index is specified, it will be the last item in the list.

4.1.6. `index()`

```
list.index(x[, start[, end]])
```

Return zero-based index in the list of the first item whose value is equal to `x` or raise a `ValueError` if there is no such item.

4.1.7. `count()`

```
list.count(x)
```

Return the number of times x appears in the list.

4.1.8. `sort()`

```
list.sort(*, key=None, reverse=False)
```

Sort the items of the list in place.

4.1.9. `reverse()`

```
list.reverse()
```

Reverse the elements of the list in place.

4.1.10. `copy()`

```
list.copy()
```

Return a shallow copy of the list.

4.2. Dictionaries

4.2.1. `iter()`

```
iter(d)
```

Return an iterator over the keys of the dictionary, which is a shortcut for `iter(d.keys())`.

4.2.2. `copy()`

```
copy()
```

Return a shallow copy of the dictionary.

4.2.3. `get()`

```
get(key[, default])
```

Return the value for `key` if `key` is in the dictionary, else `default`; if `default` is not given, it defaults to None, so that this method never raises a `KeyError`.

4.2.4. `items()`

```
items()
```

Return a new view of the dictionary's items (`(key, value)` pairs).

4.2.5. `keys()`

```
keys()
```

Return a new view of the dictionary's keys.

4.2.6. `pop()`

```
pop(key[, default])
```

If `key` is in the dictionary, remove it and return its value, else return `default`; if `default` is not given and `key` is not in the dictionary, a `KeyError` is raised.

4.2.7. `popitem()`

```
popitem()
```

Remove and return a `(key, value)` pair from the dictionary, pairs are returned in LIFO (last-in, first-out) order.

> *Changed in version 3.7*: LIFO order is now guaranteed. In prior versions, popitem() would return an arbitrary key/value pair.

4.2.8. `setdefault()`

```
setdefault(key[, default])
```

If `key` is in the dictionary, return its value. If not, insert `key` with a value of `default` and return `default`. `default` defaults to None.

4.2.9. `update()`

```
update([other])
```

Update the dictionary with the key/value pairs from `other`, overwriting existing keys, then return `None`.

4.2.10. `values()`

```
values()
```

Return a new view of the dictionary's values.

5. Modules

5.1. `pickle`

Doc: https://docs.python.org/3/library/pickle.html

```
import pickle
```

5.1.1. `dump()`

```
pickle.dump(obj, file, protocol=None, *, fix_imports=True, buffer_callback=None)
```

Write the pickled representation of the object `obj` to the open file object `file`.

5.1.2. `load()`

```
pickle.load(file, *, fix_imports=True, encoding="ASCII", errors="strict", buffers=None)
```

Read the pickled representation of an object from the open file object `file` and return the reconstituted object hierarchy specified therein.

5.2. `random`

Doc: https://docs.python.org/3/library/random.html

```
import random
```

5.2.1. `randrange()`

```
random.randrange(stop)
```

```
random.randrange(start, stop[, step])
```

Return a randomly selected element from `range(start, stop, step)`, which is equivalent to `choice(range(start, stop, step))`, but doesn't actually build a range object; the positional argument pattern matches that of `range()`.

5.2.2. `choice()`

```
random.choice(seq)
```

Return a random element from the non-empty sequence `seq`; if `seq` is empty, raises `IndexError`.

5.2.3. `choices()`

```
random.choices(population, weights=None, *, cum_weights=None, k=1)
```

Return a `k` sized list of elements chosen from the `population` with replacement; if the `population` is empty, raises IndexError.

5.2.4. `sample()`

```
random.sample(population, k, *, counts=None)
```

Return a `k` length list of unique elements chosen from the `population` sequence or set.

5.2.5. `random()`

```
random.random()
```

Return the next random floating point number in the range [0.0, 1.0).

5.2.6. `uniform()`

```
random.uniform(a, b)
```

Return a random floating point number `N` such that `a <= N <= b` for `a <= b` and `b <= N <= a` for `b < a`; the end-point value `b` may or may not be included.

5.3. `time`

Doc: https://docs.python.org/3/library/time.html

```
import time
```

5.3.1. `sleep()`

```
time.sleep(secs)
```

Suspend execution of the calling thread for the given number of seconds.

5.3.2. `time()`

```
time.time() → float
```

Return the time in seconds since the epoch as a floating point number.

5.4. `math`

Doc: https://docs.python.org/3/library/math.html

```
import math
```

5.4.1. `ceil()`

```
math.ceil(x)
```

Return the ceiling of `x`, the smallest integer greater than or equal to `x`.

5.4.2. `floor()`

```
math.floor(x)
```

Return the floor of `x`, the largest integer less than or equal to `x`.

5.4.3. `gcd()`

```
math.gcd(*integers)
```

Return the greatest common divisor of the specified integer arguments.

5.4.4. `isclose()`

```
math.isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)
```

Return `True` if the values `a` and `b` are close to each other and `False` otherwise.

5.4.5. `trunc()`

```
math.trunc(x)
```

Return the `Real` value `x` truncated to an `Integral` (usually an integer).

5.4.6. `exp()`

```
math.exp(x)
```

Return `e` raised to the power `x`, where `e` = 2.718281… is the base of natural logarithms.

5.4.7. `log()`

```
math.log(x[, base])
```

With one argument, return the natural logarithm of `x` (to base `e`); with two arguments, return the logarithm of `x` to the given `base`.

5.4.8. `pow()`

```
math.pow(x, y)
```

Return `x` raised to the power `y`.

5.4.9. `sqrt()`

```
math.sqrt(x)
```

Return the square root of x.

5.4.10. `sin()`

```
math.sin(x)
```

Return the sine of `x` radians.

5.4.11. `cos()`

```
math.cos(x)
```

Return the cosine of `x` radians.

5.4.12. `tan()`

```
math.tan(x)
```

Return the tangent of `x` radians.

5.4.13. `acos()`

```
math.acos(x)
```

Return the arc cosine of `x`, in radians.

5.4.14. `asin()`

```
math.asin(x)
```

Return the arc sine of `x`, in radians.

5.4.15. `atan()`

```
math.atan(x)
```

Return the arc tangent of x, in radians.

5.4.16. `degrees()`

```
math.degrees(x)
```

Convert angle `x` from radians to degrees.

5.4.17. `radians()`

```
math.radians(x)
```

Convert angle `x` from degrees to radians.

5.5. `numpy`

Doc: https://numpy.org/doc/stable/

I will not copy their documentation here, I will explain things myself instead. In my opinion, the doc of `numpy` is a bit long but it is really clear and specific, and you will have to seek for help on their doc as long as you work on data related fields.

```
import numpy
```

5.5.1. `numpy.array`

```
numpy.array(object, dtype=None, *, copy=True, order='K', subok=False, ndmin=0, like=None)
```

Return a `ndarray` object, a list/tuple of lists/tuples of... is allowed for `object`.

5.5.2. `numpy.arange`

```
numpy.arange([start, ]stop, [step, ]dtype=None, *, like=None)
```

Works like `range`, but floats are allowed.

5.5.3. `numpy.linspace`

```
numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)
```

Array with a specified number of elements, and spaced equally between the specified beginning and end values.

5.5.4. `numpy.eye`

```
numpy.eye(N, M=None, k=0, dtype=<class 'float'>, order='C', *, like=None)
```

Identity-matrix-like `ndarray`.

5.5.5. `numpy.diag`

```
numpy.diag(v, k=0)
```

Diagonal-matrix-like `ndarray`.

5.5.6. `numpy.zeros`

```
numpy.zeros(shape, dtype=float, order='C', *, like=None)
```

Initialize a `ndarray` with full of `0`s.

5.5.7. `numpy.ones`

```
numpy.ones(shape, dtype=None, order='C', *, like=None)
```

Initialize a `ndarray` with full of `1`s.

5.5.8. `numpy.ndarray.shape`

```
ndarray.shape
```

The shape of a `ndarray` a.k.a how many “rows” are there in each dimension, in tuple form.

6. Files

Doc: https://docs.python.org/3/tutorial/inputoutput.html

6.1. `read()`

```
f.read([size])
```

Read a file’s contents, return it as a string (in text mode) or bytes object (in binary mode); when `size` is omitted or negative, the entire contents of the file will be read and returned.

6.2. `write()`

```
f.write(string)
```

Write the contents of `string` to the file, return the number of characters written.

6.3. `tell()`

```
f.tell()
```

Return an integer giving the file object’s current position in the file represented as number of bytes from the beginning of the file when in binary mode and an opaque number when in text mode.

6.4. `seek()`

```
f.seek(offset[, whence])
```

Change the file object's position; the position is computed from adding `offset` to a reference point; the reference point is selected by the `whence` argument; a `whence` value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point; `whence` can be omitted and defaults to 0.

7. Data model (Dunder or magic methods)

Doc: https://docs.python.org/3/reference/datamodel.html

This section is of a table form to avoid confusion and... tiredness.

This dunder method...	Means
`object.__doc__`	docstring
`object.__name__`	name
`object.__lt__`	`<`
`object.__le__`	`<=`
`object.__ne__`	`!=`
`object.__gt__`	`>`
`object.__ge__`	`>=`
`object.__dir__`	`dir()`
`object.__add__(self, other)`	`+`
`object.__sub__(self, other)`	`-`
`object.__mul__(self, other)`	`*`
`object.__truediv__(self, other)`	`/`
`object.__floordiv__(self, other)`	`//`
`object.__mod__(self, other)`	`%`
`object.__divmod__(self, other)`	`divmod()`
`object.__pow__(self, other[, modulo])`	`pow()`, `**`
`object.__and__(self, other)`	`&`
`object.__xor__(self, other)`	`^`
`object.__or__(self, other)`	`|`