

The *Best Python Cheat Sheet

Just what you need

Built-in (1)	Execution (26)	List (9)	Set (10)
Bytes (11)	Flow control (4)	Number (22)	String (17)
Class (12)	Function (11)	Operator (3)	Test (26)
Debug (26)	Generator (17)	Regex (20)	Time (23)
Decorator (14)	Iterator (16)	Resource (26)	Tuple (8)
Dictionary (9)	Keyword (1)	Scope (6)	Types (26)
Exception (24)	Library (26)	Sequence (7)	

Keyword

and	continue	for	match ^①	True
as	def	from	None	try
assert	del	global	nonlocal	type ^①
async	elif	if	not	while
await	else	import	or	with
break	except	in	pass	yield
case ^①	False	is	raise	^①
class	finally	lambda	return	—

^①Soft keywords

Built-in

Built-in functions

abs(number)	Absolute value of number	bytes(...)	New bytes object from byte-integers, string, or bytes
aiter(async_iterable)	Asynchronous iterator for an asynchronous iterable	callable(object)	True if object is callable
all(iterable)	True if all elements of iterable are true (all([]) is True)	chr(i)	One character string for unicode ordinal i (0 <= i <= 0x10ffff)
any(iterable)	True if any element of iterable is true (any([]) is False)	classmethod(func)	Transform function into class method
ascii(object)	Return repr(object) with non-ASCII characters escaped	compile(source, ...)	Compile source into code or AST object
bin(number)	Convert number to binary string	complex(real=0, imag=0)	Complex number with the value real + imag*1j
bool(object)	Boolean value of object, see __bool__	delattr(object, name)	Delete the named attribute, if object allows
breakpoint(*args, **kwds)	Drop into debugger by calling sys.breakpointhook(*args, **kwds)	dict(...)	Create new dictionary
bytearray(...)	New array of bytes from byte-integers, string, bytes, or object with buffer API	dir([object])	List of names in the local scope, or object.__dir__() or attributes
		divmod(x, y)	Return (quotient x//y, remainder x%y)

<code>enumerate(iterable, start=0)</code>	Enumerate object as (n, item) pairs with n initialised to start value	<code>iter(object, ...)</code>	Iterator for object
<code>eval(source, globals=None, locals=None)</code>	Execute Python expression string, or code object from <code>compile()</code>	<code>len(object)</code>	Length of object
<code>exec(source, globals=None, locals=None)</code>	Execute Python statements string, or code object from <code>compile()</code>	<code>list(...)</code>	Create list
<code>filter(func, iterable)</code>	Iterator yielding items where <code>bool(func(item))</code> is True, or <code>bool(item)</code> if <code>func</code> is None	<code>locals()</code>	Dictionary of current local symbol table
<code>float(x=0)</code>	Floating point number from number or string	<code>map(func, *iterables)</code>	Apply function to every item of iterable(s)
<code>format(object, format_spec='')</code>	Formatted representation	<code>max(..., key=func)</code>	Largest item of iterable or arguments, optionally comparing value of <code>func(item)</code>
<code>frozenset(...)</code>	New frozenset object	<code>memoryview(object)</code>	Access internal object data via buffer protocol
<code>getattr(object, name[, default])</code>	Get value of named attribute of object, else default or raise exception	<code>min(..., key=func)</code>	Smallest item of iterable or arguments, optionally comparing value of <code>func(item)</code>
<code>globals()</code>	Dictionary of current module namespace	<code>next(iterator[, default])</code>	Next item from iterator, optionally return default instead of <code>StopIteration</code>
<code>hasattr(object, name)</code>	True if object has named attribute	<code>object()</code>	New featureless object
<code>hash(object)</code>	Hash value of object, see <code>object.__hash__()</code>	<code>oct(number)</code>	Convert number to octal string
<code>help(...)</code>	Built-in help system	<code>open(file, ...)</code>	Create file object from path string/bytes or integer file descriptor
<code>hex(number)</code>	Convert number to lowercase hexadecimal string	<code>ord(chr)</code>	Integer representing Unicode code point of character
<code>id(object)</code>	Return unique integer identifier of object	<code>pow(base, exp, mod=None)</code>	Return <i>base</i> to the power of <i>exp</i>
<code>__import__(name, ...)</code>	Invoked by the import statement	<code>print(*values, sep=',', end='\n', file=sys.stdout, flush=False)</code>	Print object to <code>sys.stdout</code> , or text stream file
<code>input(prompt='')</code>	Read string from <code>sys.stdin</code> , with optional prompt	<code>property(...)</code>	Property decorator
<code>int(...)</code>	Create integer from number or string	<code>range(...)</code>	Generate integer sequence
<code>isinstance(object, cls_or_tuple)</code>	True if object is instance of given class(es)	<code>repr(object)</code>	String representation of object for debugging
<code>issubclass(cls, cls_or_tuple)</code>	True if class is subclass of given class(es)	<code>reversed(sequence)</code>	Reverse iterator

<code>round(number, ndigits=None)</code>	Number rounded to ndigits precision after decimal point	<code>sum(iterable, start=0)</code>	Sums items of iterable, optionally adding start value
<code>set(...)</code>	New set object	<code>super(...)</code>	Proxy object that delegates method calls to parent or sibling
<code>setattr(object, name, value)</code>	Set object attribute value by name	<code>tuple(iterable)</code>	Create a tuple
<code>slice(...)</code>	Slice object representing a set of indices	<code>type(...)</code>	Type of an object, or build new type
<code>sorted(iterable, key=func, reverse=False)</code>	New sorted list from the items in iterable, optionally comparing value of func(item)	<code>vars([object])</code>	Return object.__dict__ or locals() if no argument
<code>staticmethod(func)</code>	Transform function into static method	<code>zip(*iterables, strict=False)</code>	Iterate over multiple iterables in parallel, strict requires equal length
<code>str(...)</code>	String description of object		

Operator

Precedence (high->low)	Description
<code>(...), [...], {...}, {...:...,}</code>	tuple, list, set, dict
<code>s[i] s[i:j] s.attr f(...)</code>	index, slice, attribute, function call
<code>await x</code>	await expression
<code>+x, -x, ~x</code>	unary positive, negative, bitwise NOT
<code>x ** y</code>	power
<code>x * y, x @ y, x / y, x // y, x % y</code>	multiply, maxtrix multiply, divide, floor divide, modulus
<code>x + y, x - y</code>	add, subtract
<code>x << y x >> y</code>	bitwise shift left, right
<code>x & y</code>	bitwise and
<code>x ^ y</code>	bitwise exclusive or
<code>x y</code>	bitwise or
<code>x<y x<=y x>y x>=y x==y x!=y</code> <code>x is y x is not y</code> <code>x in s x not in s</code>	comparison, identity, membership
<code>not x</code>	boolean negation
<code>x and y</code>	boolean and
<code>x or y</code>	boolean or
<code>... if ... else ...</code>	conditional expression
<code>lambda</code>	lambda expression
<code>:=</code>	assignment expression

Assignment	Usually equivalent
<code>a = b</code>	Assign object b to label a
<code>a += b</code>	<code>a = a + b</code>
<code>a -= b</code>	<code>a = a - b</code>
<code>a *= b</code>	<code>a = a * b</code>
<code>a /= b</code>	<code>a = a / b</code> (true division)
<code>a //= b</code>	<code>a = a // b</code> (floor division)
<code>a %= b</code>	<code>a = a % b</code>
<code>a **= b</code>	<code>a = a ** b</code>
<code>a &= b</code>	<code>a = a & b</code>
<code>a = b</code>	<code>a = a b</code>
<code>a ^= b</code>	<code>a = a ^ b</code>
<code>a >>= b</code>	<code>a = a >> b</code>
<code>a <<= b</code>	<code>a = a << b</code>

Assignment expression

Assign and return value using the *walrus operator*.

```
count = 0
while (count := count + 1) < 5:
    print(count)

>>> z = [1, 2, 3, 4, 5]
>>> [x for i in z if (x:=i**2) > 10]
[16, 25]
```

Assignment unpacking

Unpack multiple values to a name using the *splat operator*.

```
head, *body = s # assign first value of s to head, remainder to body
head, *body, tail = s # assign first and last values of s to head and tail,
remainder to body
*body, tail = s # assign last value of s to tail, remainder to body
s = [*iterable[, ...]] # unpack iterable to list
s = (*iterable[, ...]) # unpack iterable to tuple
s = {*iterable[, ...]} # unpack iterable to set
d2 = {**d1[, ...]} # unpack mapping to dict
```

Flow control

```
for item in <iterable>:
    ...
[else:
    ...] # only if loop completes without break

while <condition>:
    ...
[else:
    ...] # only if loop completes without break

break # immediately exit loop
continue # skip to next loop iteration
return[ value] # exit function, return value | None
yield[ value] # exit generator, yield value | None
assert <expr>[, message] # if not <expr> raise AssertionError([message])
```

```

if <condition>:
    ...
[elif <condition>:
    ...]*
[else:
    ...]

<expression1> if <condition> else <expression2>

with <expression>[ as name]: # context manager
    ...

```

Context manager

A *with* statement takes an object with special methods:

- `__enter__()` - locks resources and optionally returns an object
- `__exit__()` - releases resources, handles any exception raised in the block, optionally suppressing it by returning True

```

class AutoClose:
    def __init__(self, filename):
        self.filename = filename
    def __enter__(self):
        self.f = open(self.filename)
        return self.f
    def __exit__(self, exc_type, exception, traceback):
        self.f.close()

```

```

>>> with AutoClose('test.txt') as f:
...     print(f.read())
Hello world!

```

Match

3.10+

```

match <expression>:
    case <pattern>[ if <condition>]: # conditional match, if "guard" clause
        ...
    case <pattern1> | <pattern2>:    # OR pattern
        ...
    case _:                        # default case
        ...

```

Match case pattern

<code>1/'abc'/True/None/math.pi</code>	Value pattern, match literal or dotted name
<code><name></code>	Capture pattern, match any object and bind to name
<code>_</code>	Wildcard pattern, match any object
<code><type>()</code>	Class pattern, match any object of that type
<code><type>(<attr>=<pattern name>, ...)</code>	Class pattern, match object with matching attributes
<code><pattern> <pattern> [...]</code>	Or pattern, match any of the patterns left to right
<code>[<pattern>[, ...[, *args]]</code>	Sequence pattern (list tuple), match any sequence with matching items (but not string or iterator), may be nested
<code>{<value_pattern>: <pattern>[, ...[, **kws]]}</code>	Mapping pattern, match dictionary with matching items, may be nested

<code><pattern> as <name></code>	Bind match to name
<code><builtin>(<name>)</code>	Builtin pattern, shortcut for <code><builtin>()</code> as <code><name></code> (e.g. <code>str</code> , <code>int</code>)

- Class patterns
 - **Do not** create a new instance of the class
 - Accept positional parameters if class defines `__match_args__` special attribute (e.g. `dataclass`)
- Sequence patterns support assignment unpacking
- Names bound in a match statement are visible after the match statement

Scope

Scope levels:

Builtin	Names pre-assigned in <i>builtins</i> module	Generator expression	Names contained within generator expression
Module (global)	Names defined in current module Note: Code in global scope cannot access local variables	Comprehension	Names contained within comprehension
Enclosing (closure)	Names defined in any enclosing functions	Class	Names shared across all instances
Function (local)	Names defined in current function Note: By default, has read-only access to module and enclosing function names By default, assignment creates a new local name <i>global <name></i> grants read/write access to specified module name <i>nonlocal <name></i> grants read/write access to specified name in closest enclosing function defining that name	Instance	Names contained within a specific instance
		Method	Names contained within a specific instance method

- `globals()` - return Dictionary of module scope variables
- `locals()` - return Dictionary of local scope variables

```

>>> global_name = 1
>>> def read_global():
...     print(global_name)
...     local_name = "only available in this function"
>>> read_global()
1
>>> def write_global():
...     global global_name # enable write to global
...     global_name = 2
>>> write_global()
>>> print(global_name)
2
>>> def write_nonlocal():
...     closure_name = 1
...     def nested():
...         nonlocal closure_name # enable write to nonlocal
...         closure_name = 2
...     nested()
...     print(closure_name)
>>> write_nonlocal()
2

```

```

class C:
    class_name = 1 # shared by all instances
    def __init__(self):
        self.instance_name = 2 # only on this instance
    def method(self):
        self.instance_name = 3 # update instance name
        C.class_name = 3 # update class name
        method_name = 1 # set method local name

```

Sequence

Operations on sequence types (Bytes, List, Tuple, String).

<code>x in s</code>	True if any <code>s[i] == x</code>	<code>s.index(x[, start[, stop]])</code>	Smallest <code>i</code> where <code>s[i] == x</code> , start/stop bounds search
<code>x not in s</code>	True if no <code>s[i] == x</code>		
<code>s1 + s2</code>	Concatenate <code>s1</code> and <code>s2</code>	<code>reversed(s)</code>	Iterator on <code>s</code> in reverse order For string use <code>reversed(list(s))</code>
<code>s * n, n * s</code>	Concatenate <code>n</code> copies of <code>s</code>		
<code>s.count(x)</code>	Count of <code>s[i] == x</code>	<code>sorted(s, cmp=func, key=getter, reverse=False)</code>	New sorted list
<code>len(s)</code>	Count of items		
<code>min(s)</code>	Smallest item of <code>s</code>		
<code>max(s)</code>	Largest item of <code>s</code>		

Indexing

Select items from sequence by index or slice.

```

>>> s = [0, 1, 2, 3, 4]
>>> s[0]          # 0-based indexing
0
>>> s[-1]         # negative indexing from end
4
>>> s[slice(2)]    # slice(stop) - index from 0 until stop (exclusive)
[0, 1]
>>> s[slice(1, 5, 3)] # slice(start, stop[, step]) - index from start to stop
(exclusive), with optional step size (+|-)
[1, 4]
>>> s[:2]         # slices are created implicitly when indexing with ':'
[start:stop:step]
[0, 1]
>>> s[3::-1]      # negative step
[3, 2, 1, 0]
>>> s[1:3]
[1, 2]
>>> s[1:5:2]
[1, 3]

```

Comparison

- A sortable class should define `__eq__()`, `__lt__()`, `__gt__()`, `__le__()` and `__ge__()` special methods.
- With `functools @total_ordering` decorator a class need only provide `__eq__()` and one other comparison special method.
- Sequence comparison: values are compared in order until a pair of unequal values is found. The comparison of these two values is then returned. If all values are equal, the shorter sequence is lesser.

```

from functools import total_ordering

@total_ordering
class C:
    def __init__(self, a):
        self.a = a
    def __eq__(self, other):
        if isinstance(other, type(self)):
            return self.a == other.a
        return NotImplemented
    def __lt__(self, other):
        if isinstance(other, type(self)):
            return self.a < other.a
        return NotImplemented

```

Tuple

Immutable hashable sequence.

<code>s = ()</code>	Empty tuple
<code>s = (1, 'a', 3.0)</code> <code>s = 1, 'a', 3.0</code>	Create from items
<code>s = (1,)</code>	Single-item tuple
<code>(1, 2, 3) == (1, 2) + (3,)</code>	Add makes new tuple
<code>(1, 2, 1, 2) == (1, 2) * 2</code>	Multiply makes new tuple

Named tuple

Tuple subclass with named items. Also: `typing.NamedTuple`


```
>>> from collections import namedtuple
>>> Point = namedtuple('Point', ('x', 'y')) # or namedtuple('Point', 'x y')
>>> p = Point(1, y=2)
Point(x=1, y=2)
>>> p[0]
1
>>> p.y
2
```

List

Mutable non-hashable sequence.

<code>s = []</code>	Empty list	<code>s.extend(it)</code>	Add items from iterable to end
<code>s = [1, 'a', 3.0]</code>	Create from items	<code>s[len(s):len(s)] = it</code>	
<code>s = list(range(3))</code>		<code>s.insert(i, x)</code>	Insert item at index i
<code>s[i] = x</code>	Replace item index i with x	<code>s[i:i] = [x]</code>	
<code>s[<slice>] = it</code>	Replace slice with iterable	<code>s.remove(x)</code>	Remove first item where <code>s[i] == x</code>
<code>del s[<slice>]</code>	Remove slice	<code>del s[s.index(x)]</code>	
<code>s[<slice>] = []</code>		<code>y = s.pop([i])</code>	Remove and return last item or indexed item
<code>s.append(x)</code>	Add item to end	<code>s.reverse()</code>	Reverse items in place
<code>s += x</code>		<code>s.sort(cmp=func, key=getter, reverse=False)</code>	Sort items in place, default ascending
<code>s[len(s):len(s)] = [x]</code>			

List comprehension

```
result = [<expression> for item1 in <iterable1>{ if <condition1>}
          {for item2 in <iterable2>{ if <condition2>} ... for itemN in <iterableN>{
if <conditionN>}}]

# is equivalent to:

result = []
for item1 in <iterable1>:
    for item2 in <iterable2>:
        ...
        for itemN in <iterableN>:
            if <condition1> and <condition2> ... and <conditionN>:
                result.append(<expression>)
```

Dictionary

Mutable non-hashable key:value pair mapping.

<code>dict()</code>	Empty dict	<code>dict.fromkeys(keys, value=None)</code>	Create from keys, all set to value
<code>dict(<sequence mapping>)</code>	Create from key:value pairs	<code>d.keys()</code>	Iterable of keys
<code>{'d':4, 'a':2}</code>		<code>d.values()</code>	Iterable of values
<code>dict(**kwds)</code>	Create from keyword arguments	<code>d.items()</code>	Iterable of (key, value) pairs
<code>dict(zip(keys, values))</code>	Create from sequences of keys and values	<code>d.get(key, default=None)</code>	Get value for key, or default

<code>d.setdefault(key, default=None)</code>	Get value for key, add if missing	<code>d.clear()</code>	Remove all items
<code>d.pop(key[, default])</code>	Remove and return value for key, raise <code>KeyError</code> if missing and no default	<code>d.copy()</code>	Shallow copy
<code>d.popitem()</code>	Remove and return (key, value) pair (last-in, first-out)	<code>d1.update(d2)</code> <code>d1 = d2</code> 3.9+	Add/replace key:value pairs from d2 to d1
		<code>d3 = d1 d2</code> 3.9+ <code>d3 = {**d1, **d2}</code>	Merge to new dict, d2 trumps d1

```
# defaultdict(<callable>) sets default value returned by callable()
import collections
collections.defaultdict(lambda: 42) # dict with default value 42
```

Dict comprehension

```
# {k: v for k, v in <iterable>[ if <condition>]}

>>> {x: x**2 for x in (2, 4, 6) if x < 5}
{2: 4, 4: 16}
```

Set

Mutable (*set*) and immutable (*frozenset*) sets.

<code>set()</code> <code>frozenset()</code>	Empty set	<code>s.clear()</code> <i>[mutable]</i>	Remove all elements
<code>{1, 2, 3}</code>	Create from items, note: {} creates empty dict - sad!	<code>s1.intersection(s2[, s3...])</code> <code>s1 & s2</code>	New set of shared elements
<code>set(iterable)</code> <code>{*iterable}</code>	Create from iterable	<code>s1.intersection_update(s2)</code> <i>[mutable]</i>	Update s1 to intersection with s2
<code>len(s)</code>	Cardinality	<code>s1.union(s2[, s3...])</code> <code>s1 s2</code>	New set of all elements
<code>v in s</code> <code>v not in s</code>	Test membership	<code>s1.difference(s2[, s3...])</code> <code>s1 - s2</code>	New set of elements unique to s1
<code>s1.issubset(s2)</code>	True if s1 is subset of s2	<code>s1.difference_update(s2)</code> <i>[mutable]</i>	Remove s1 elements intersecting with s2
<code>s1.issuperset(s2)</code>	True if s1 is superset of s2	<code>s1.symmetric_difference(s2)</code> <code>s1 ^ s2</code>	New set of unshared elements
<code>s.add(v)</code> <i>[mutable]</i>	Add element	<code>s1.symmetric_difference_update(s2)</code> <i>[mutable]</i>	Update s1 to symmetric difference with s2
<code>s.remove(v)</code> <i>[mutable]</i>	Remove element (<code>KeyError</code> if not found)	<code>s.copy()</code>	Shallow copy
<code>s.discard(v)</code> <i>[mutable]</i>	Remove element if present	<code>s.update(it1[, it2...])</code> <i>[mutable]</i>	Add elements from iterables
<code>s.pop()</code> <i>[mutable]</i>	Remove and return arbitrary element (<code>KeyError</code> if empty)		

Set comprehension

```
# {x for x in <iterable>[ if <condition>]}

>>> {x for x in 'abracadabra' if x not in 'abc'}
{'r', 'd'}
```

Bytes

Immutable sequence of bytes. Mutable version is *bytearray*.

<code>b'abc\x42'</code>	Create from ASCII characters and <code>\x00-\xff</code>	<code><bytes> = <bytes> [<slice>]</code>	Return <i>bytes</i> even if only one element
<code>bytes(<ints>)</code>	Create from int sequence	<code>list(<bytes>)</code>	Return ints in range 0 to 255
<code>bytes(<str>, 'utf-8')</code> <code><str>.encode('utf-8')</code>	Create from string	<code><bytes_sep>.join(<byte_objs>)</code>	Join <i>byte_objs</i> sequence with <i>bytes_sep</i> separator
<code><int>.to_bytes(length, order, signed=False)</code>	Create from int (order='big' 'little')	<code>str(<bytes>, 'utf-8')</code> <code><bytes>.decode('utf-8')</code>	Convert bytes to string
<code>bytes.fromhex('<hex>')</code>	Create from hex pairs (can be separated by whitespace)	<code>int.from_bytes(bytes, order, signed=False)</code>	Return int from bytes (order='big' 'little')
<code><int> = <bytes> [<index>]</code>	Return int in range 0 to 255	<code><bytes>.hex(sep=' ', bytes_per_sep=2)</code>	Return hex pairs

```
def read_bytes(filename):
    with open(filename, 'rb') as f:
        return f.read()

def write_bytes(filename, bytes_obj):
    with open(filename, 'wb') as f:
        f.write(bytes_obj)
```

Function

Function definition

```
# var-positional
def f(*args): ...           # f(1, 2)
def f(x, *args): ...        # f(1, 2)
def f(*args, z): ...        # f(1, z=2)

# var-keyword
def f(**kwds): ...          # f(x=1, y=2)
def f(x, **kwds): ...       # f(x=1, y=2) | f(1, y=2)

def f(*args, **kwds): ...   # f(x=1, y=2) | f(1, y=2) | f(1, 2)
def f(x, *args, **kwds): ... # f(x=1, y=2, z=3) | f(1, y=2, z=3) | f(1, 2, z=3) | f(1, 2, 3)
def f(*args, y, **kwds): ... # f(x=1, y=2, z=3) | f(1, y=2, z=3)

# positional-only before /
def f(x, /, y): ...         # f(1, 2) | f(1, y=2)
def f(x, y, /): ...         # f(1, 2)

# keyword-only after *
def f(x, *, y): ...         # f(x=1, y=2) | f(1, y=2)
def f(*, x, y): ...         # f(x=1, y=2)
```

Function call

```
args = (1, 2)           # * expands sequence to positional arguments
kws = {'x': 3, 'y': 4}   # ** expands dictionary to keyword arguments
func(*args, **kws)       # is the same as:
func(1, 2, x=3, y=4)
```

Class

Instantiation

```
class C:
    """Class docstring."""
    def __init__(self, a):
        """Method docstring."""
        self.a = a
    def __repr__(self):
        """Used for repr(c), also for str(c) if __str__ not defined."""
        return f'{self.__class__.__name__}({self.a!r})'
    def __str__(self):
        """Used for str(c), e.g. print(c)"""
        return str(self.a)
    @classmethod
    def get_class_name(cls): # passed class rather than instance
        return cls.__name__
    @staticmethod
    def static(): # passed nothing
        return 1

>>> c = C(2) # instantiate

# under the covers, class instantiation does this:
obj = cls.__new__(cls, *args, **kws)
if isinstance(obj, cls):
    obj.__init__(*args, **kws)
```

Instance property

```
class C:
    @property
    def f(self):
        if not hasattr(self, '_f'):
            return
        return self._f
    @f.setter
    def f(self, value):
        self._f = value
```

Class special methods

Operator	Method
self + other	__add__(self, other)
other + self	__radd__(self, other)
self += other	__iadd__(self, other)
self - other	__sub__(self, other)
other - self	__rsub__(self, other)
self -= other	__isub__(self, other)
self * other	__mul__(self, other)
other * self	__rmul__(self, other)
self *= other	__imul__(self, other)
self @ other	__matmul__(self, other)
other @ self	__rmatmul__(self, other)
self @= other	__imatmul__(self, other)
self / other	__truediv__(self, other)
other / self	__rtruediv__(self, other)
self /= other	__itruediv__(self, other)
self // other	__floordiv__(self, other)
other // self	__rfloordiv__(self, other)
self //= other	__ifloordiv__(self, other)
self % other	__mod__(self, other)
other % self	__rmod__(self, other)
self %= other	__imod__(self, other)
self ** other	__pow__(self, other)
other ** self	__rpow__(self, other)
self **= other	__ipow__(self, other)
self << other	__lshift__(self, other)
other << self	__rlshift__(self, other)
self <<= other	__ilshift__(self, other)
self >> other	__rshift__(self, other)
other >> self	__rrshift__(self, other)
self >>= other	__irshift__(self, other)
self & other	__and__(self, other)
other & self	__rand__(self, other)
self &= other	__iand__(self, other)
self other	__or__(self, other)
other self	__ror__(self, other)
self = other	__ior__(self, other)
self ^ other	__xor__(self, other)
other ^ self	__rxor__(self, other)
self ^= other	__ixor__(self, other)
divmod(self, other)	__divmod__(self, other)
divmod(self, other)	__rdivmod__(self, other)

Operator	Method
-self	<code>__neg__(self)</code>
+self	<code>__pos__(self)</code>
<code>abs(self)</code>	<code>__abs__(self)</code>
~self	<code>__invert__(self)</code> [bitwise]
<code>self == other</code>	<code>__eq__(self)</code> [default 'is', requires <code>__hash__</code>]
<code>self != other</code>	<code>__ne__(self)</code>
<code>self < other</code>	<code>__lt__(self, other)</code>
<code>self <= other</code>	<code>__le__(self, other)</code>
<code>self > other</code>	<code>__gt__(self, other)</code>
<code>self >= other</code>	<code>__ge__(self, other)</code>
<code>item in self</code>	<code>__contains__(self, item)</code>
<code>bool(self)</code>	<code>__bool__(self)</code>
<code>if self:</code>	
<code>if not self:</code>	
<code>bytes(self)</code>	<code>__bytes__(self)</code>
<code>complex(self)</code>	<code>__complex__(self)</code>
<code>float(self)</code>	<code>__float__(self)</code>
<code>int(self)</code>	<code>__int__(self)</code>
<code>round(self)</code>	<code>__round__(self[, ndigits])</code>
<code>math.ceil(self)</code>	<code>__ceil__(self)</code>
<code>math.floor(self)</code>	<code>__floor__(self)</code>
<code>math.trunc(self)</code>	<code>__trunc__(self)</code>
<code>dir(self)</code>	<code>__dir__(self)</code>
<code>format(self)</code>	<code>__format__(self, format_spec)</code>
<code>hash(self)</code>	<code>__hash__(self)</code>
<code>iter(self)</code>	<code>__iter__(self)</code>
<code>len(self)</code>	<code>__len__(self)</code>
<code>repr(self)</code>	<code>__repr__(self)</code>
<code>reversed(self)</code>	<code>__reversed__(self)</code>
<code>str(self)</code>	<code>__str__(self)</code>
<code>self(*args, **kwargs)</code>	<code>__call__(self, *args, **kwargs)</code>
<code>self[...]</code>	<code>__getitem__(self, key)</code>
<code>self[...] = 1</code>	<code>__setitem__(self, key, value)</code>
<code>del self[...]</code>	<code>__delitem__(self, key)</code>
<code>other[self]</code>	<code>__index__(self)</code>
<code>self.name</code>	<code>__getattr__(self, name)</code> <code>__getattr__(self, name)</code> [if <code>AttributeError</code>]
<code>self.name = 1</code>	<code>__setattr__(self, name, value)</code>
<code>del self.name</code>	<code>__delattr__(self, name)</code>
<code>with self:</code>	<code>__enter__(self)</code> <code>__exit__(self, exc_type, exc_value, traceback)</code>
<code>await self</code>	<code>__await__(self)</code>

Decorator

Decorator syntax passes a function or class to a callable and replaces it with the return value.

```

def show_call(obj):
    """
    Decorator that prints obj name and arguments each time obj is called.
    """
    def show_call_wrapper(*args, **kwargs):
        print(obj.__name__, args, kwargs)
        return obj(*args, **kwargs)
    return show_call_wrapper

@show_call # function decorator
def add(x, y):
    return x + y

# is equivalent to
add = show_call(add)

>>> add(13, 29)
add (13, 29) {}
42

@show_call # class decorator
class C:
    def __init__(self, a=None):
        pass

# is equivalent to
C = show_call(C)

>>> C(a=42)
C () {'a': 42}

```

```

# decorators optionally take arguments
def show_call_if(condition):
    """
    Apply show_call decorator only if condition is True.
    """
    return show_call if condition else lambda obj: obj

@show_call_if(False)
def add(x, y):
    return x + y

# is equivalent to
add = show_call_if(False)(add)

>>> add(13, 29)
42

@show_call_if(True)
def add(x, y):
    return x + y

>>> add(13, 29)
add (13, 29) {}
42

>>> add.__name__
'show_call_wrapper' # ugh! decorated function has different metadata

# @wraps decorator copies metadata of decorated object to wrapped object
# preserving original attributes (e.g. __name__)
from functools import wraps

def show_call_preserve_meta(obj):
    @wraps(obj)
    def show_call_wrapper(*args, **kwargs):
        print(obj.__name__, args, kwargs)
        return obj(*args, **kwargs)
    return show_call_wrapper

@show_call_preserve_meta
def add(x, y):
    return x + y

>>> add.__name__
'add'

```

Iterator

An iterator implements the `__iter__()` method, returning an iterable that implements the `__next__()` method. The `__next__()` method returns the next item in the collection and raises `StopIteration` when done.


```
class C:
    def __init__(self, items):
        self.items = items

    def __iter__(self):
        """Make class its own iterable."""
        return self

    def __next__(self):
        """Implement to be iterable."""
        if self.items:
            return self.items.pop()
        raise StopIteration
```

```
>>> c = C([13, 29])
>>> it = iter(c)      # get iterator
>>> next(it)          # get next item
29
>>> for item in c:    # iterate over C instance
...     print(item)
13
```

Generator

A function with a *yield* statement returns a generator iterator and suspends function processing. Each iteration over the generator iterator resumes function execution, returns the next yield value, and suspends again.

```
def gen():
    """Generator function"""
    for i in [13, 29]:
        yield i

>>> g = gen()
>>> next(g)           # next value
13
>>> for item in gen(): # iterate over values
...     print(item)
13
29
>>> list(gen())        # list all values
[13, 29]

def parent_gen():
    yield from gen()   # delegate yield to another generator

>>> list(parent_gen())
[13, 29]
```

Generator expression

```
# (<expression> for <name> in <iterable>[ if <condition>])
>>> g = (item for item in [13, 29] if item > 20)
>>> list(g)
[29]
```

String

Immutable sequence of characters.

<code><substring> in s</code>	True if string contains <i>substring</i>	<code>s.casefold()</code>	To lower case (aggressive)
<code>s.startswith(<prefix>[, start[, end]])</code>	True if string starts with <i>prefix</i> , optionally search bounded substring	<code>s.upper()</code>	To upper case
<code>s.endswith(<suffix>[, start[, end]])</code>	True if string ends with <i>suffix</i> , optionally search bounded substring	<code>s.title()</code>	To title case (The Quick Brown Fox)
<code>s.strip(chars=None)</code>	Strip whitespace from both ends, or passed characters	<code>s.capitalize()</code>	Capitalize first letter
<code>s.lstrip(chars=None)</code>	Strip whitespace from left end, or passed characters	<code>s.swapcase()</code>	Swap case
<code>s.rstrip(chars=None)</code>	Strip whitespace from right end, or passed characters	<code>s.replace(old, new[, count])</code>	Replace <i>old</i> with <i>new</i> at most <i>count</i> times
<code>s.ljust(width, fillchar=' ')</code>	Left justify with <i>fillchar</i>	<code>s.translate(<table>)</code>	Use <i>str.maketrans(<dict>)</i> to generate table
<code>s.rjust(width, fillchar=' ')</code>	Right justify with <i>fillchar</i>	<code>s.expandtabs(tabsize=8)</code>	Expand tabs to spaces
<code>s.center(width, fillchar=' ')</code>	Center with <i>fillchar</i>	<code>chr(<int>)</code>	Integer to Unicode character
<code>s.split(sep=None, maxsplit=-1)</code>	Split on whitespace, or <i>sep</i> str at most <i>maxsplit</i> times	<code>ord(<str>)</code>	Unicode character to integer
<code>s.splitlines(keepend=False)</code>	Split lines on <code>[\n\r\f\v\x1c-\x1e\x85\u2028\u2029]</code> and <code>\r\n</code>	<code><str>.encode(encoding='utf-8', errors='strict')</code>	Encode string to bytes
<code><separator>.join(<strings>)</code>	Join sequence of <i>strings</i> with <i>separator</i> string	<code>s.isalnum()</code>	True if <code>isnumeric()</code> or <code>[a-zA-Z...]</code> (>0 characters)
<code>s.format(*args, **kwargs)</code>	Substitute arguments into <code>{}</code> placeholders	<code>s.isalpha()</code>	True if <code>[a-zA-Z...]</code> (>0 characters)
<code>s.format_map(mapping)</code>	Substitute mapping into <code>{}</code> placeholders	<code>s.isdecimal()</code>	True if <code>[0-9]</code> , <code>[0-9]</code> or <code>[9-.]</code> (>0 characters)
<code>s.find(<substring>)</code>	Index of first match or -1	<code>s.isdigit()</code>	True if <code>isdecimal()</code> or <code>[²₃1...</code> (>0 characters)
<code>s.rfind(<substring>)</code>	Index of last match or -1	<code>s.isidentifier()</code>	True if valid Python name (including keywords)
<code>s.index(<substring>)</code>	Index of first match or raise <code>ValueError</code>	<code>s.islower()</code>	True if all characters are lower case (>0 characters)
<code>s.rindex(<substring>)</code>	Index of last match or raise <code>ValueError</code>	<code>s.isnumeric()</code>	True if <code>isdigit()</code> or <code>[₁₂₃零〇一...]</code> (>0 characters)
<code>s.count(<substring>[, start[, end]])</code>	Count instances of <i>substring</i> , optionally search bounded substring	<code>s.isprintable()</code>	True if <code>isalnum()</code> or <code>[!#\$%...]</code> (>0 characters)
<code>s.lower()</code>	To lower case	<code>s.isspace()</code>	True if <code>[\t\n\r\f\v\x1c-\x1f\x85\xa0...]</code> (>0 characters)
		<code>s.istitle()</code>	True if string is title case (>0 characters)

<code>s.isupper()</code>	True if all characters are upper case (>0 characters)	<code>head, sep, tail = s.rpartition(<separator>)</code>	Search for <i>separator</i> from end and split
<code>head, sep, tail = s.partition(<separator>)</code>	Search for <i>separator</i> from start and split	<code>s.removeprefix(<prefix>)</code> 3.9+	Remove <i>prefix</i> if present
		<code>s.removesuffix(<suffix>)</code> 3.9+	Remove <i>suffix</i> if present

String escape

Sequence	Escape
Literal backslash	<code>\\</code>
Single quote	<code>\'</code>
Double quote	<code>\"</code>
Backspace	<code>\b</code>
Carriage return	<code>\r</code>
Newline	<code>\n</code>
Tab	<code>\t</code>
Vertical tab	<code>\v</code>
Null	<code>\0</code>
Hex value	<code>\xff</code>
Octal value	<code>\o77</code>
Unicode 16 bit	<code>\uxxxx</code>
Unicode 32 bit	<code>\Uxxxxxxxx</code>
Unicode name	<code>\N{name}</code>

String formatting

Format	f-string	Output
Escape curly braces	<code>f"{{ }}"</code>	<code>'{'</code>
Expression	<code>f"{6/3}, {'a'+ 'b'}"</code> <code>'{ }, {'}'.format(6/3, 'a'+ 'b')</code>	<code>'2, ab'</code>
Justify left	<code>f'{1:<5}'</code>	<code>'1 '</code>
Justify center	<code>f'{1:^5}'</code>	<code>' 1 '</code>
Justify right	<code>f'{1:>5}'</code>	<code>' 1'</code>
Justify left with char	<code>f'{1:.<5}'</code>	<code>'1....'</code>
Justify right with char	<code>f'{1:.>5}'</code>	<code>'....1'</code>
Trim	<code>f"{'abc':.2}"</code>	<code>'ab'</code>
Trim justify left	<code>f"{'abc':6.2}"</code>	<code>'ab '</code>
<code>ascii()</code>	<code>f'{v!a}'</code>	<code>ascii(v)</code>
<code>repr()</code>	<code>f'{v!r}'</code>	<code>repr(v)</code>
<code>str()</code>	<code>f'{v!s}'</code>	<code>str(v)</code>
Justify left <code>repr()</code>	<code>f"{'abc'!r:6}"</code>	<code>"'abc' "</code>
Date format	<code>f'{today:%d %b %Y}'</code>	<code>'21 Jan 1984'</code>

Format	f-string	Output
Significant figures	<code>f'{1.234:.2}'</code>	<code>'1.2'</code>
Fixed-point notation	<code>f'{1.234:.2f}'</code>	<code>'1.23'</code>
Scientific notation	<code>f'{1.234:.2e}'</code>	<code>'1.230e+00'</code>
Percentage	<code>f'{1.234:.2%}'</code>	<code>'123.40%'</code>
Pad with zeros	<code>f'{1.7:04}'</code>	<code>'01.7'</code>
Pad with spaces	<code>f'{1.7:4}'</code>	<code>' 1.7'</code>
Pad before sign	<code>f'{123:+6}'</code>	<code>' +123'</code>
Pad after sign	<code>f'{123:+=6}'</code>	<code>'+ 123'</code>
Separate with commas	<code>f'{123456:,}'</code>	<code>'123,456'</code>
Separate with underscores	<code>f'{123456:}_'</code>	<code>'123_456'</code>
<code>f'{1+1=}'</code>	<code>f'{1+1=}'</code>	<code>'1+1=2' (= prepends)</code>
Binary	<code>f'{164:b}'</code>	<code>'10100100'</code>
Octal	<code>f'{164:o}'</code>	<code>'244'</code>
Hex	<code>f'{164:X}'</code>	<code>'A4'</code>
<code>chr()</code>	<code>f'{164:c}'</code>	<code>'ÿ'</code>

Regex

Standard library `re` module provides Python regular expressions.

```
>>> import re
>>> my_re = re.compile(r'name is (?P<name>[A-Za-z]+)')
>>> match = my_re.search('My name is Douglas.')
>>> match.group()
'name is Douglas'
>>> match.group(1)
'Douglas'
>>> match.groupdict()['name']
'Douglas'
```

Regex syntax

.	Any character (newline if DOTALL)		Or
^	Start of string (every line if MULTILINE)	(...)	Group
\$	End of string (every line if MULTILINE)	(?:...)	Non-capturing group
*	0 or more of preceding	(?P<name>...)	Named group
+	1 or more of preceding	(?P=name)	Match text matched by earlier group
?	0 or 1 of preceding	(?=...)	Match next, non-consumptive
*?, +?, ??	Same as *, + and ?, as few as possible	(?!...)	Non-match next, non-consumptive
{m,n}	m to n repetitions	(?<=...)	Match preceding, positive lookbehind assertion
{m,n}?	m to n repetitions, as few as possible	(?<!...)	Non-match preceding, negative lookbehind assertion
[...]	Character set: e.g. '[a-zA-Z]'	(?(group)A B)	Conditional match - A if group previously matched else B
[^...]	NOT character set	(? letters)	Set flags for RE ('i', 'L', 'm', 's', 'u', 'x')
\	Escape chars '*?+&\$ ()', introduce special sequences	(?#...)	Comment (ignored)
\\	Literal '\'		

Regex special sequences

<code>\<n></code>	Match by integer group reference starting from 1
<code>\A</code>	Start of string
<code>\b</code>	Word boundary (see flag: <code>ASCII LOCALE</code>)
<code>\B</code>	Not word boundary (see flag: <code>ASCII LOCALE</code>)
<code>\d</code>	Decimal digit (see flag: <code>ASCII</code>)
<code>\D</code>	Non-decimal digit (see flag: <code>ASCII</code>)

Regex flags

Flags modify regex behaviour. Pass to regex functions (e.g. `re.A` | `re.ASCII`) or embed in regular expression (e.g. `(?a)`).

<code>(?a)</code> <code>A</code> <code>ASCII</code>	ASCII-only match for <code>\w</code> , <code>\W</code> , <code>\b</code> , <code>\B</code> , <code>\d</code> , <code>\D</code> , <code>\s</code> , <code>\S</code> (default is Unicode)
<code>(?i)</code> <code>I</code> <code>IGNORECASE</code>	Case insensitive matching
<code>(?L)</code> <code>L</code> <code>LOCALE</code>	Apply current locale for <code>\w</code> , <code>\W</code> , <code>\b</code> , <code>\B</code> (discouraged)

<code>\s</code>	Whitespace [<code>\t\n\r\f\v</code>] (see flag: <code>ASCII</code>)
<code>\S</code>	Non-whitespace (see flag: <code>ASCII</code>)
<code>\w</code>	Alphanumeric (see flag: <code>ASCII LOCALE</code>)
<code>\W</code>	Non-alphanumeric (see flag: <code>ASCII LOCALE</code>)
<code>\Z</code>	End of string

<code>(?m)</code> <code>M</code> <code>MULTILINE</code>	Match every new line, not only start/end of string
<code>(?s)</code> <code>S</code> <code>DOTALL</code>	'.' matches ALL chars, including newline
<code>(?x)</code> <code>X</code> <code>VERBOSE</code>	Ignores whitespace outside character sets
<code>DEBUG</code>	Display expression debug info

Regex functions

<code>compile(pattern[, flags=0])</code>	Compiles Regular Expression Object
<code>escape(string)</code>	Escape non-alphanumerics
<code>match(pattern, string[, flags])</code>	Match from start
<code>search(pattern, string[, flags])</code>	Match anywhere
<code>split(pattern, string[, maxsplit=0])</code>	Splits by pattern, keeping splitter if grouped

<code>findall(pattern, string)</code>	Non-overlapping matches as list of groups or tuples (>1)
<code>finditer(pattern, string[, flags])</code>	Iterator over non-overlapping matches
<code>sub(pattern, repl, string[, count=0])</code>	Replace count first leftmost non-overlapping; If repl is function, called with a <code>MatchObj</code>
<code>subn(pattern, repl, string[, count=0])</code>	Like <code>sub()</code> , but returns (<code>newString</code> , <code>numberOfSubsMade</code>)

Regex object

<code>flags</code>	Flags
<code>groupindex</code>	{group name: group number}
<code>pattern</code>	Pattern
<code>match(string[, pos][, endpos])</code>	Match from start of target[pos:endpos]
<code>search(string[, pos][, endpos])</code>	Match anywhere in target[pos:endpos]

<code>split(string[, maxsplit=0])</code>	See <code>split()</code> function
<code>findall(string[, pos[, endpos]])</code>	See <code>findall()</code> function
<code>finditer(string[, pos[, endpos]])</code>	See <code>finditer()</code> function
<code>sub(repl, string[, count=0])</code>	See <code>sub()</code> function
<code>subn(repl, string[, count=0])</code>	See <code>subn()</code> function

Regex match object

pos	pos passed to search or match	start(group), end(group)	Indices of start & end of group match (None if group exists but didn't contribute)
endpos	endpos passed to search or match	span(group)	(start(group), end(group)); (None, None) if group didn't contribute
re	RE object	string	String passed to match() or search()
group([g1, g2, ...])	One or more groups of match One arg, result is a string Multiple args, result is tuple If gi is 0, returns the entire matching string If 1 <= gi <= 99, returns string matching group (None if no such group) May also be a group name Tuple of match groups Non-participating groups are None String if len(tuple)==1		

Number

bool([object]) True, False	Boolean, see <code>__bool__</code> special method
int([float str bool]) 5	Integer, see <code>__int__</code> special method
float([int str bool]) 5.1, 1.2e-4	Float (inexact, compare with <code>math.isclose(<float>, <float>)</code>) See <code>__float__</code> special method
complex(real=0, imag=0) 3 - 2j, 2.1 + 0.8j	Complex, see <code>__complex__</code> special method
fractions.Fraction(<numerator>, <denominator>)	Fraction
decimal.Decimal([str int])	Decimal (exact, set precision: <code>decimal.getcontext().prec = <int></code>)
bin([int]) 0b101010 int('101010', 2) int('0b101010', 0)	Binary
hex([int]) 0x2a int('2a', 16) int('0x2a', 0)	Hex

Mathematics

```
from math import (e, pi, inf, nan, isinf, isnan,
                  sin, cos, tan, asin, acos, atan, degrees, radians,
                  log, log10, log2)
```

Also: built-in functions (abs, max, min, pow, round, sum)

Statistics

```
from statistics import mean, median, variance, stdev, quantiles, groupby
```

Random

```
>>> from random import random, randint, choice, shuffle, gauss, triangular, seed
>>> random() # float inside [0, 1)
0.42
>>> randint(1, 100) # int inside [<from>, <to>]
42
>>> choice(range(100)) # random item from sequence
42
```

Time

The *datetime* module provides immutable hashable *date*, *time*, *datetime*, and *timedelta* classes.

Time formatting

Code	Output
%a	Day name short (Mon)
%A	Day name full (Monday)
%b	Month name short (Jan)
%B	Month name full (January)
%c	Locale datetime format
%d	Day of month [01,31]
%f	Microsecond [000000,999999]
%H	Hour (24-hour) [00,23]
%I	Hour (12-hour) [01,12]
%j	Day of year [001,366]
%m	Month [01,12]
%M	Minute [00,59]
%p	Locale format for AM/PM
%S	Second [00,61]. Yes, 61!
%U	Week number (Sunday start) [00(partial),53]
%w	Day number [0(Sunday),6]
%W	Week number (Monday start) [00(partial),53]
%x	Locale date format
%X	Locale time format
%y	Year without century [00,99]
%Y	Year with century (2023)
%Z	Time zone ('' if no TZ)
%z	UTC offset (+HHMM/-HHMM, '' if no TZ)
%%	Literal '%'

Exception

```
try:
    ...
except [<Exception>[ as e]]:
    ...
except: # catch all
    ...
else: # if no exception
    ...
finally: # always executed
    ...

raise <exception>[ from <exception|None>]

try:
    1 / 0
except ZeroDivisionError:
    # from None hides exception context
    raise TypeError("Hide ZeroDivisionError") from None
```


BaseException	Base class for all exceptions
└ BaseExceptionGroup	Base class for groups of exceptions
└ GeneratorExit	Generator close() raises to terminate iteration
└ KeyboardInterrupt	On user interrupt key (often 'CTRL-C')
└ SystemExit	On sys.exit()
└ Exception	Base class for errors
└ ArithmeticError	Base class for arithmetic errors
└ FloatingPointError	Floating point operation failed
└ OverflowError	Result too large
└ ZeroDivisionError	Argument of division or modulo is 0
└ AssertionError	Assert statement failed
└ AttributeError	Attribute reference or assignment failed
└ BufferError	Buffer operation failed
└ EOFError	input() hit end-of-file without reading data
└ ExceptionGroup	Group of exceptions raised together
└ ImportError	Import statement failed
└ ModuleNotFoundError	Module not able to be found
└ LookupError	Base class for lookup errors
└ IndexError	Index not found in sequence
└ KeyError	Key not found in dictionary
└ MemoryError	Operation ran out of memory
└ NameError	Local or global name not found
└ UnboundLocalError	Local variable value not assigned
└ OSError	System related error
└ BlockingIOError	Non-blocking operation will block
└ ChildProcessError	Operation on child process failed
└ ConnectionError	Base class for connection errors
└ BrokenPipeError	Write to closed pipe or socket
└ ConnectionAbortedError	Connection aborted
└ ConnectionRefusedError	Connection denied by server
└ ConnectionResetError	Connection reset mid-operation
└ FileExistsError	Trying to create a file that already exists
└ FileNotFoundError	File or directory not found
└ InterruptedError	System call interrupted by signal
└ IsADirectoryError	File operation requested on a directory
└ NotADirectoryError	Directory operation requested on a non-directory
└ PermissionError	Operation has insufficient access rights
└ ProcessLookupError	Operation on process that no longer exists
└ TimeoutError	Operation timed out
└ ReferenceError	Weak reference used on garbage collected object
└ RuntimeError	Error detected that doesn't fit other categories
└ NotImplementedError	Operation not yet implemented
└ RecursionError	Maximum recursion depth exceeded
└ StopAsyncIteration	Iterator __anext__() raises to stop iteration
└ StopIteration	Iterator next() raises when no more values
└ SyntaxError	Python syntax error
└ IndentationError	Base class for indentation errors
└ TabError	Inconsistent tabs or spaces
└ SystemError	Recoverable Python interpreter error
└ TypeError	Operation applied to wrong type object
└ ValueError	Operation on right type but wrong value
└ UnicodeError	Unicode encoding/decoding error
└ UnicodeDecodeError	Unicode decoding error
└ UnicodeEncodeError	Unicode encoding error
└ UnicodeTranslateError	Unicode translation error
└ Warning	Base class for warnings
└ BytesWarning	Warnings about bytes and bytearrays
└ DeprecationWarning	Warnings about deprecated features
└ EncodingWarning	Warning about encoding problem
└ FutureWarning	Warnings about future deprecations for end users
└ ImportWarning	Possible error in module imports
└ PendingDeprecationWarning	Warnings about pending feature deprecations
└ ResourceWarning	Warning about resource use
└ RuntimeWarning	Warning about dubious runtime behavior
└ SyntaxWarning	Warning about dubious syntax
└ UnicodeWarning	Warnings related to Unicode
└ UserWarning	Warnings generated by user code

Execution

```
$ python [-bBdEhiIOqsSuvVWx?] [-c command | -m module-name | script | - ] [args]
$ python --version
Python 3.10.12
$ python --help[-all] # help-all [3.11+]
# Execute code from command line
$ python -c 'print("Hello, world!")'
# Execute __main__.py in directory
$ python <directory>
# Execute module as __main__
$ python -m timeit -s 'setup here' 'benchmarked code here'
# Optimise execution
$ python -O script.py

# Hide warnings
PYTHONWARNINGS="ignore"
# OR
$ python -W ignore foo.py
# OR
import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)
```

```
# module of executed script is assigned __name__ '__main__'
# so to run main() only if module is executed as script
if __name__ == '__main__':
    main()
```

Environment variables

PYTHONHOME	Change location of standard Python libraries	PYTHONOPTIMIZE	Optimise execution (-O)
PYTHONPATH	Augment default search path for module files	PYTHONWARNINGS	Set warning level [default/error/always/module/once/ignore] (-W)
PYTHONSTARTUP	Module to execute before entering interactive prompt	PYTHONPROFILEIM PORTTIME	Show module import times (-X)

sitecustomize.py / usercustomize.py

Before `__main__` module is executed Python automatically imports:

- `sitecustomize.py` in the system site-packages directory
- `usercustomize.py` in the user site-packages directory

```
# Get user site packages directory
$ python -m site --user-site

# Bypass sitecustomize.py/usercustomize.py hooks
$ python -S script.py
```