The *Best Python Cheat Sheet

Just what you need

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	

[1]Soft keywords

Built-in

Built-in functions

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

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

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

Operator

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

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

Assignment expression

Assign and return value using the *walrus operator*.

```
if matching := pattern.search(data):
    do_something(matching)

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 to list
s  = (*iterable[, …])  # unpack to tuple
s  = {*iterable[, …]}  # unpack to set
d2 = {**d1[, …]}       # unpack to dict
```

Flow control

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

while <condition>:
    …
[else:                      # 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 expression 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 open('test.txt', 'w') as f:
...         f.write('Hello World!')
>>> 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

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

- 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 *builtins* module
Module (global)	Names defined in current module Code in global scope cannot access local variables
Enclosing (closure)	Names defined in any enclosing functions
Function (local)	Names defined in current function By default, has read-only access to module and enclosing function names By default, assignment creates a new local name *global <name>* grants read/write access to specified module name *nonlocal <name>* grants read/write access to specified name in closest enclosing function defining that name
Generator expression	Names contained within generator expression

Comprehension	Names contained within comprehension
Class	Names shared across all instances
Instance	Names contained within a specific instance
Method	Names contained within a specific instance method

- *globals()* - return *dict* of module scope variables
- *locals()* - return *dict* 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
...     global_name = 2
>>> write_global()
>>> print(global_name)
2
>>> def write_nonlocal():
...     closure_name = 1
...     def nested():
...         nonlocal closure_name
...         closure_name = 2
...     nested()
...     print(closure_name)
>>> write_nonlocal()
2
```

```
class C:
    class_name = 1
    def __init__(self):
        self.instance_name = 2
    def method(self):
        self.instance_name = 3
        C.class_name = 3
        method_name = 1
```

Sequence

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

x in s	True if any s[i] == x
x not in s	True if no s[i] == x
s1 + s2	Concatenate s1 and s2
s * n, n * s	Concatenate n copies of s
s.count(x)	Count of s[i] == x
len(s)	Count of items
min(s)	Smallest item
max(s)	Largest item

s.index(x[, start[, stop]])	Smallest i where s[i] == x, start/stop bounds search
reversed(s)	Iterator on s in reverse order (for string: reversed(list(s)))
sorted(s, cmp=func, key=getter, reverse=False)	New sorted list

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.

`s = (1, 'a', 3.0)` `s = 1, 'a', 3.0`	Create tuple
`s = (1,)`	Single-item tuple
`s = ()`	Empty tuple
`(1, 2, 3) == (1, 2) + (3,)`	Add makes new tuple
`(1, 2, 1, 2) == (1, 2) * 2`	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.

`s = [1, 'a', 3.0]` `s = list(range(3))`	Create list
`s[i] = x`	Replace item index i with x
`s[<slice>] = it`	Replace slice with iterable
`del s[<slice>]` `s[<slice>] = []`	Remove slice

`s.append(x)`
`s += x`
`s[len(s):len(s)] = [x]`
	Add item to end

`s.extend(it)` `s[len(s):len(s)] = it`	Add items from iterable to end
`s.insert(i, x)` `s[i:i] = [x]`	Insert item at index i
`s.remove(x)` `del s[s.index(x)]`	Remove first item where s[i] == x
`y = s.pop([i])`	Remove and return last item or indexed item
`s.reverse()`	Reverse items in place
`s.sort(cmp=func, key=getter, reverse=False)`	Sort items in place, default ascending

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.

`dict()` `{}`	Empty dict
`dict(<sequence|mapping>)` `{'d':4, 'a':2}`	Create from key:value pairs
`dict(**kwds)`	Create from keyword arguments
`dict(zip(keys, values))`	Create from sequences of keys and values
`dict.fromkeys(keys, value=None)`	Create from keys, all set to value

d.keys()	Iterable of keys
d.values()	Iterable of values
d.items()	Iterable of (key, value) pairs
d.get(key, default=None)	Get value for key, or default
d.setdefault(key, default=None)	Get value for key, add if missing
d.pop(key)	Remove and return value for key, raise KeyError if missing

d.popitem()	Remove and return (key, value) pair (last-in, first-out)
d.clear()	Remove all items
d.copy()	Shallow copy
d1.update(d2) d1 |= d2 3.9+	Add/replace key:value pairs from d2 to d1
d3 = d1 | d2 3.9+ d3 = {**d1, **d2}	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.

set()	Empty set
{1, 2, 3}	Create (note: {} creates empty dict - sad!)
set(iterable) {*iterable}	Create from iterable
frozenset(iterable=None)	Create frozen set
len(s)	Cardinality
v in s v not in s	Test membership
s1.issubset(s2)	True if s1 is subset of s2
s1.issuperset(s2)	True if s1 is superset of s2
s.add(v) *[mutable]*	Add element
s.remove(v) *[mutable]*	Remove element (KeyError if not found)
s.discard(v) *[mutable]*	Remove element if present
s.pop() *[mutable]*	Remove and return arbitrary element (KeyError if empty)

s.clear() *[mutable]*	Remove all elements
s1.intersection(s2[, s3…]) s1 & s2	New set of shared elements
s1.intersection_update(s2) *[mutable]*	Update elements to intersection with s2
s1.union(s2[, s3…]) s1 | s2	New set of all elements
s1.difference(s2[, s3…]) s1 - s2	New set of elements unique to s1
s1.difference_update(s2) *[mutable]*	Remove elements intersecting with s2
s1.symmetric_difference(s2) s1 ^ s2	New set of unshared elements
s1.symmetric_difference_update(s2) *[mutable]*	Update elements to symmetric difference with s2
s.copy()	Shallow copy
s.update(it1[, it2…]) *[mutable]*	Add elements from iterables

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*.

b'<str>'	Create from ASCII characters and \x00-\xff
bytes(<ints>)	Create from int sequence
bytes(<str>, 'utf-8') <str>.encode('utf-8')	Create from string
<int>.to_bytes(length, order, signed=False)	Create from int (order='big'|'little')
bytes.fromhex('<hex>')	Create from hex pairs (can be separated by whitespace)
<int> = <bytes>[<index>]	Return int in range 0 to 255
<bytes> = <bytes>[<slice>]	Return *bytes* even if only one element
list(<bytes>)	Return ints in range 0 to 255
<bytes_sep>.join(<byte_objs>)	Join *byte_objs* sequence with *bytes_sep* separator
str(<bytes>, 'utf-8') <bytes>.decode('utf-8')	Convert bytes to string
int.from_bytes(bytes, order, signed=False)	Return int from bytes (order='big'|'little')
<bytes>.hex(sep='', bytes_per_sep=2)	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, y=2) | f(1, 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
kwds = {'x': 3, 'y': 4}     # ** expands dictionary to keyword arguments
func(*args, **kwds)         # 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 by 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, **kwds)
if isinstance(obj, cls):
    obj.__init__(*args, **kwds)
```

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	__neg__(self)
+self	__pos__(self)
abs(self)	__abs__(self)
~self	__invert__(self) [bitwise]
self == other	__eq__(self) [default 'is', requires __hash__]
self != other	__ne__(self)
self < other	__lt__(self, other)
self <= other	__le__(self, other)
self > other	__gt__(self, other)
self >= other	__ge__(self, other)
item in self	__contains__(self, item)

bool(self)
if self:
if not self:
	__bool__(self)

bytes(self)	__bytes__(self)
complex(self)	__complex__(self)
float(self)	__float__(self)
int(self)	__int__(self)
round(self)	__round__(self[, ndigits])
math.ceil(self)	__ceil__(self)
math.floor(self)	__floor__(self)
math.trunc(self)	__trunc__(self)
dir(self)	__dir__(self)
format(self)	__format__(self, format_spec)
hash(self)	__hash__(self)
iter(self)	__iter__(self)
len(self)	__len__(self)
repr(self)	__repr__(self)
reversed(self)	__reversed__(self)
str(self) self(*args, **kwds)	__str__(self) __call__(self, *args, **kwds)
self[…]	__getitem__(self, key)
self[…] = 1	__setitem__(self, key, value)
del self[…]	__delitem__(self, key)
other[self]	__index__(self)
self.name	__getattribute__(self, name) __getattr__(self, name) [if AttributeError]
self.name = 1	__setattr__(self, name, value)
del self.name	__delattr__(self, name)
with self:	__enter__(self) __exit__(self, exc_type, exc_value, traceback)
await self	__await__(self)

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, **kwds):
        print(obj.__name__, args, kwds)
        return obj(*args, **kwds)
    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, **kwds):
        print(obj.__name__, args, kwds)
        return obj(*args, **kwds)
    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.

<substring> in s	True if string contains *substring*
s.startswith(<prefix>[, start[, end]])	True if string starts with *prefix*, optionally search bounded substring
s.endswith(<suffix>[, start[, end]])	True if string ends with *suffix*, optionally search bounded substring
s.strip(chars=None)	Strip whitespace from both ends, or passed characters
s.lstrip(chars=None)	Strip whitespace from left end, or passed characters
s.rstrip(chars=None)	Strip whitespace from right end, or passed characters
s.ljust(width, fillchar=' ')	Left justify with fillchar
s.rjust(width, fillchar=' ')	Right justify with fillchar
s.center(width, fillchar=' ')	Center with fillchar
s.split(sep=None, maxsplit=-1)	Split on whitespace, or *sep* str at most *maxsplit* times
s.splitlines(keepends=False)	Split lines on [\n\r\f\v\x1c-\x1e\x85\u2028\u2029] and \r\n
<separator>.join(<strings>)	Join sequence of *strings* with *separator* string
s.find(<substring>)	Index of first match or -1
s.index(<substring>)	Index of first match or raise ValueError

s.lower()	To lower case
s.upper()	To upper case
s.title()	To title case (The Quick Brown Fox)
s.capitalize()	Capitalize first letter
s.replace(old, new[, count])	Replace *old* with *new* at most *count* times
s.translate(<table>)	Use *str.maketrans(<dict>)* to generate table
chr(<int>)	Integer to Unicode character
ord(<str>)	Unicode character to integer
s.isdecimal()	True if [0-9], [٠-٩] or [۰-۹]
s.isdigit()	True if isdecimal() or [²³¹…]
s.isnumeric()	True if isdigit() or [¼½¾零〇一…]
s.isalnum()	True if isnumeric() or [a-zA-Z…]
s.isprintable()	True if isalnum() or [!#$%…]
s.isspace()	True if [\t\n\r\f\v\x1c-\x1f\x85\xa0…]
head, sep, tail = s.partition(<separator>)	Search for *separator* from start and split
head, sep, tail = s.rpartition(<separator>)	Search for *separator* from end and split
s.removeprefix(<prefix>) 3.9+	Remove *prefix* if present
s.removesuffix(<suffix>) 3.9+	Remove *suffix* if present

String escape

Sequence	Escape
Literal backslash	\\
Single quote	\'
Double quote	\"
Backspace	\b
Carriage return	\r

Sequence	Escape
Newline	\n
Tab	\t
Vertical tab	\v
Null	\0
Hex value	\xff
Octal value	\o77
Unicode 16 bit	\uxxxx
Unicode 32 bit	\Uxxxxxxxx
Unicode name	\N{name}

String formatting

Format	f-string	Output
Escape curly braces	f"{{}}"	'{}'
Expression	f"{6/3}, {'a'+'b'}" '{}, {}'.format(6/3, 'a'+'b')	'2, ab'
Justify left	f'{1:<5}'	'1 '
Justify center	f'{1:^5}'	' 1 '
Justify right	f'{1:>5}'	' 1'
Justify left with char	f'{1:.<5}'	'1....'
Justify right with char	f'{1:.>5}'	'....1'
Trim	f"{'abc':.2}"	'ab'
Trim justify left	f"{'abc':6.2}"	'ab '
ascii()	f'{v!a}'	ascii(v)
repr()	f'{v!r}'	repr(v)
str()	f'{v!s}'	str(v)
Justify left repr()	f"{'abc'!r:6}"	"'abc' "
Date format	f'{today:%d %b %Y}'	'21 Jan 1984'
Significant figures	f'{1.234:.2}'	'1.2'
Fixed-point notation	f'{1.234:.2f}'	'1.23'
Scientific notation	f'{1.234:.2e}'	'1.230e+00'
Percentage	f'{1.234:.2%}'	'123.40%'
Pad with zeros	f'{1.7:04}'	'01.7'
Pad with spaces	f'{1.7:4}'	' 1.7'
Pad before sign	f'{123456:+6}'	' +123'
Pad after sign	f'{123456:=+6}'	'+ 123'
Separate with commas	f'{123456:,}'	'123,456'
Separate with underscores	f'{123456:_}'	'123_456'
f'{1+1=}'	f'{1+1=}'	'1+1=2' (= prepends)
Binary	f'{164:b}'	'10100100'
Octal	f'{164:o}'	'244'
Hex	f'{164:X}'	'A4'
chr()	f'{164:c}'	'ÿ'

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)
^	Start of string (every line if MULTILINE)
$	End of string (every line if MULTILINE)
*	0 or more of preceding
+	1 or more of preceding
?	0 or 1 of preceding
*?, +?, ??	Same as *, + and ?, as few as possible
{m,n}	m to n repetitions
{m,n}?	m to n repetitions, as few as possible
[…]	Character set: e.g. '[a-zA-Z]'
[^…]	NOT character set
\	Escape chars '*?+&$|()', introduce special sequences
\\	Literal '\'

|	Or
(…)	Group
(?:…)	Non-capturing group
(?P<name>…)	Named group
(?P=name)	Match text matched by earlier group
(?=…)	Match next, non-consumptive
(?!…)	Non-match next, non-consumptive
(?<=…)	Match preceding, positive lookbehind assertion
(?<!…)	Non-match preceding, negative lookbehind assertion
(?(group)A|B)	Conditional match - A if group previously matched else B
(?letters)	Set flags for RE ('i','L', 'm', 's', 'u', 'x')
(?#…)	Comment (ignored)

Regex special sequences

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

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

Regex flags

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

(?a) | A | ASCII	ASCII-only match for \w, \W, \b, \B, \d, \D, \s, \S (default is Unicode)
(?i) | I | IGNORECASE	Case insensitive matching
(?L) | L | LOCALE	Apply current locale for \w, \W, \b, \B (discouraged)
(?m) | M | MULTILINE	Match every new line, not only start/end of string
(?s) | S | DOTALL	'.' matches ALL chars, including newline
(?x) | X | VERBOSE	Ignores whitespace outside character sets
DEBUG	Display expression debug info

Regex functions

compile(pattern[, flags=0])	Compiles Regular Expression Obj
escape(string)	Escape non-alphanumerics
match(pattern, string[, flags])	Match from start
search(pattern, string[, flags])	Match anywhere
split(pattern, string[, maxsplit=0])	Splits by pattern, keeping splitter if grouped
findall(pattern, string)	Non-overlapping matches as list of groups or tuples (>1)
finditer(pattern, string[, flags])	Iterator over non-overlapping matches
sub(pattern, repl, string[, count=0])	Replace count first leftmost non-overlapping; If repl is function, called with a MatchObj
subn(pattern, repl, string[, count=0])	Like sub(), but returns (newString, numberOfSubsMade)

Regex object

flags	Flags
groupindex	{group name: group number}
pattern	Pattern
match(string[, pos][, endpos])	Match from start of target[pos:endpos]
search(string[, pos][, endpos])	Match anywhere in target[pos:endpos]
split(string[, maxsplit=0])	See split() function
findall(string[, pos[, endpos]])	See findall() function
finditer(string[, pos[, endpos]])	See finditer() function
sub(repl, string[, count=0])	See sub() function
subn(repl, string[, count=0])	See subn() function

Regex match object

pos	pos passed to search or match
endpos	endpos passed to search or match
re	RE object

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
start(group), end(group)	Indices of start & end of group match (None if group exists but didn't contribute)
span(group)	(start(group), end(group)); (None, None) if group didn't contibute
string	String passed to match() or search()

Number

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

Mathematics

Also see built-in functions abs, pow, round, sum, min, max.

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

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 asssigned
   ├─ 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 insuffient 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 bytesarrays
      ├─ 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
PYTHONPATH	Augment default search path for module files
PYTHONSTARTUP	Module to execute before entering interactive prompt
PYTHONOPTIMIZE	Optimise execution (-O)
PYTHONWARNINGS	Set warning level [default/error/always/module/once/ignore] (-W)
PYTHONPROFILEIMPORTTIME	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
```