

Python

Aayush Bajaj

April 28, 2026

Contents

1	Data Types	1
2	Keywords	2
3	Time Complexities	2
3.1	lists	2
3.2	double-ended queue (deque)	3
3.3	dictionary	3
3.4	set	3
3.5	counter	4
3.6	heap / priority queue	4
3.7	sorted list	4
3.8	general traversals:	4
4	Operations (and their return types)	4
4.1	dict views	4
4.2	list mutation	4
4.3	set algebra	5
4.4	string essentials	5
4.5	numeric & iterator built-ins	5
5	REPL	5
5.1	the underscore variable	6
5.2	introspection	6
5.3	invocation flags	6
5.4	environment	6
5.5	alternatives	6

1 Data Types

category	types
text	<code>str</code>
numeric	<code>int</code> , <code>float</code> , <code>complex</code>
sequence	<code>list</code> , <code>tuple</code> , <code>range</code>
mapping	<code>dict</code>
set	<code>set</code> , <code>frozenset</code>
boolean	<code>bool</code>
binary	<code>bytes</code> , <code>bytearray</code> , <code>memoryview</code>
none	<code>NoneType</code>

2 Keywords

Python reserves 35 hard keywords (plus the soft keywords `match` and `case` for structural pattern matching since 3.10). Reserved keywords cannot be used as identifiers; soft keywords are only special in the relevant context.

category	keywords
values	<code>True</code> , <code>False</code> , <code>None</code>
logical	<code>and</code> , <code>or</code> , <code>not</code>
identity	<code>is</code>
membership	<code>in</code>
branching	<code>if</code> , <code>elif</code> , <code>else</code>
looping	<code>for</code> , <code>while</code> , <code>break</code> , <code>continue</code> , <code>pass</code>
pattern	<code>match</code> , <code>case</code>
functions	<code>def</code> , <code>return</code> , <code>lambda</code> , <code>yield</code>
classes	<code>class</code>
exceptions	<code>try</code> , <code>except</code> , <code>finally</code> , <code>raise</code> , <code>assert</code>
imports	<code>import</code> , <code>from</code> , <code>as</code>
scoping	<code>global</code> , <code>nonlocal</code>
async	<code>async</code> , <code>await</code>
context	<code>with</code>
deletion	<code>del</code>

soft
keywords
only
reserved
inside a
match
statement

A live list is available via `import keyword`; `keyword.kwlist` (and `keyword.softkwlist` for soft keywords).

3 Time Complexities ¹

3.1 lists

operation	time complexity
<code>sequence.append(item)</code>	$\mathcal{O}(1)$
<code>sequence.pop(item)</code>	$\mathcal{O}(1)$
<code>sequence.insert(0,item)</code>	$\mathcal{O}(n)$
<code>sequence.pop(item,0)</code>	$\mathcal{O}(n)$
<code>sequence[index]</code>	$\mathcal{O}(1)$
<code>sequence[index]=value</code>	$\mathcal{O}(1)$
<code>item in sequence</code>	$\mathcal{O}(n)$
<code>len(sequence)</code>	$\mathcal{O}(1)$
<code>sequence.extend(iterable)</code>	$\mathcal{O}(k)$
<code>sequence + other_sequence</code>	$\mathcal{O}(n + k)$
<code>sequence[index:index+k]</code>	$\mathcal{O}(k)$
<code>sequence.index(item)</code>	$\mathcal{O}(n)$
<code>sequence.count(item)</code>	$\mathcal{O}(n)$
<code>for item in sequence:</code>	$\mathcal{O}(n)$

slow
because
popping a
start
requires
shifting all
indices
down

¹information courtesy of <https://www.pythonmorsels.com/time-complexities/>

3.2 double-ended queue (deque)

operation	description	time complexity
<code>queue.append(item)</code>	Add item to right end	$\mathcal{O}(1)$
<code>queue.appendleft(item)</code>	Add item to left end	$\mathcal{O}(1)$
<code>queue.pop()</code>	Remove and return item from right end	$\mathcal{O}(1)$
<code>queue.popleft()</code>	Remove and return item from left end	$\mathcal{O}(1)$
<code>queue.extend(iterable)</code>	Add all elements from iterable to right end	$\mathcal{O}(k)$
<code>queue.extendleft(iterable)</code>	Add all elements from iterable to left (reversed)	$\mathcal{O}(k)$
<code>queue.remove(value)</code>	Remove first occurrence of value	$\mathcal{O}(n)$
<code>queue.rotate(n)</code>	Rotate n steps right (negative for left)	$\mathcal{O}(k)$
<code>queue.clear()</code>	Remove all elements	$\mathcal{O}(n)$
<code>queue.count(value)</code>	Count occurrences of value	$\mathcal{O}(n)$
<code>queue.index(value)</code>	Find index of first occurrence of value	$\mathcal{O}(n)$
<code>queue.reverse()</code>	Reverse elements in place	$\mathcal{O}(n)$
<code>item in queue</code>	Check if item exists in queue	$\mathcal{O}(n)$
<code>queue[0]</code> or <code>queue[-1]</code>	Access first or last element	$\mathcal{O}(1)$
<code>queue[i]</code>	Access element at index i	$\mathcal{O}(n)$
<code>for item in queue</code>	Iterate through all elements	$\mathcal{O}(n)$

3.3 dictionary

operation	time complexity
<code>mapping[key]=value</code>	$\mathcal{O}(1)$
<code>mapping[key]</code>	$\mathcal{O}(1)$
<code>mapping.get(key)</code>	$\mathcal{O}(1)$
<code>mapping.pop(key)</code>	$\mathcal{O}(1)$
<code>key in mapping</code>	$\mathcal{O}(1)$
<code>for k, v in mapping.items()</code>	$\mathcal{O}(n)$
<code>next(iter(mapping))</code>	$\mathcal{O}(1)$
<code>next(reversed(mapping))</code>	$\mathcal{O}(1)$
<code>value in mapping.values()</code>	$\mathcal{O}(n)$
<code>mapping.update(iterable)</code>	$\mathcal{O}(k)$

3.4 set

operation	time complexity
<code>my_set.add(item)</code>	$\mathcal{O}(1)$
<code>my_set.remove(item)</code>	$\mathcal{O}(1)$
<code>item in my_set</code>	$\mathcal{O}(1)$
<code>for item in my_set</code>	$\mathcal{O}(n)$
<code>set1 & set2</code>	$\mathcal{O}(n)$
<code>set1 set2</code>	$\mathcal{O}(n)$
<code>set1 ^ set2</code>	$\mathcal{O}(n)$
<code>set1 - set2</code>	$\mathcal{O}(n)$

3.5 counter

operation	time complexity
<code>counter[item]</code>	$\mathcal{O}(1)$
<code>counter.pop(item)</code>	$\mathcal{O}(1)$
<code>for k, v in counter.items():</code>	$\mathcal{O}(n)$
<code>for k, v in counter.most_common():</code>	$\mathcal{O}(n \log(n))$
<code>for k, v in counter.most_common(k):</code>	$\mathcal{O}(n \log(k))$
<code>counter.update(iterable)</code>	$\mathcal{O}(k)$
<code>counter.subtract(iterable)</code>	$\mathcal{O}(k)$
<code>counter.total()</code>	$\mathcal{O}(n)$

3.6 heap / priority queue

operation	time complexity
<code>heapq.heapify(sequence)</code>	$\mathcal{O}(n)$
<code>heapq.heappop(sequence)</code>	$\mathcal{O}(\log(n))$
<code>heapq.heappush(sequence, item)</code>	$\mathcal{O}(\log(n))$
<code>sequence[0]</code>	$\mathcal{O}(1)$

3.7 sorted list

operation	time complexity
<code>sorted_sequence = sorted(sequence)</code>	$\mathcal{O}(n \log(n))$
<code>sorted_sequence.index(item)</code>	$\mathcal{O}(n)$
<code>bisect.bisect(sorted_sequence, item)</code>	$\mathcal{O}(\log(n))$
<code>bisect.insort(sorted_sequence, item)</code>	$\mathcal{O}(n)$

3.8 general traversals:

operation	time complexity
<code>min(iterable)</code>	$\mathcal{O}(n)$
<code>max(iterable)</code>	$\mathcal{O}(n)$
<code>sorted(iterable)</code>	$\mathcal{O}(n \log(n))$
<code>heapq.nsmallest(k, iterable)</code>	$\mathcal{O}(n \log(k))$
<code>statistics.multimode(iterable)</code>	$\mathcal{O}(n)$

4 Operations (and their return types)

Many of the operations have been enumerated in the previous section on Time Complexities, but operations such as `.values()` on a **dict**, have not been discussed. This section will aggregate that information.

4.1 dict views

The view objects are **dynamic** — they reflect mutations to the underlying dict.

operation	returns	notes
<code>d.keys()</code>	<code>dict_keys</code>	set-like view of keys
<code>d.values()</code>	<code>dict_values</code>	view of values (not set-like — may dupe)
<code>d.items()</code>	<code>dict_items</code>	set-like view of (key, value) tuples
<code>d.copy()</code>	<code>dict</code>	shallow copy
<code>dict.fromkeys(it, v)</code>	<code>dict</code>	new dict with keys from <code>it</code> mapped to <code>v</code>

4.2 list mutation

In-place mutators on `list` return **None** — chaining `lst.sort().reverse()` is a common bug.

operation	returns	notes
<code>lst.sort(key=fn, reverse=True)</code>	<code>None</code>	stable, in-place
<code>lst.reverse()</code>	<code>None</code>	in-place
<code>lst.copy()</code>	<code>list</code>	shallow copy (same as <code>lst[:]</code>)
<code>lst.clear()</code>	<code>None</code>	empties in place
<code>sorted(lst, key=fn)</code>	<code>list</code>	non-mutating; returns new list
<code>reversed(lst)</code>	<code>list_reverseiterator</code>	lazy iterator

4.3 set algebra

Each of the named methods has an operator counterpart that requires both sides to be sets; the methods accept any iterable.

operation	returns	symbol
<code>a.union(b)</code>	<code>set</code>	$a \mid b$
<code>a.intersection(b)</code>	<code>set</code>	$a \& b$
<code>a.difference(b)</code>	<code>set</code>	$a - b$
<code>a.symmetric_difference(b)</code>	<code>set</code>	$a \wedge b$
<code>a.issubset(b)</code>	<code>bool</code>	$a \leq b$
<code>a.issuperset(b)</code>	<code>bool</code>	$a \geq b$
<code>a.isdisjoint(b)</code>	<code>bool</code>	—

4.4 string essentials

`str` is immutable — every method returns a new string.

operation	returns	notes
<code>s.split(sep)</code>	<code>list</code>	splits on whitespace if <code>sep</code> omitted
<code>sep.join(iterable)</code>	<code>str</code>	iterable must contain strings
<code>s.strip()</code>	<code>str</code>	also <code>lstrip</code> , <code>rstrip</code>
<code>s.replace(a, b)</code>	<code>str</code>	replaces all (or first <code>n</code>)
<code>s.startswith(p)</code>	<code>bool</code>	<code>p</code> may be a tuple of prefixes
<code>s.endswith(p)</code>	<code>bool</code>	suffix variant of the above
<code>s.find(sub)</code>	<code>int</code>	<code>-1</code> if absent (vs <code>index</code> which raises)
<code>s.format(*args, **kw)</code>	<code>str</code>	f-strings preferred for literals
<code>s.encode("utf-8")</code>	<code>bytes</code>	inverse of <code>bytes.decode</code>

4.5 numeric & iterator built-ins

operation	returns	notes
<code>abs(x)</code>	same numeric type	also magnitude for <code>complex</code>
<code>round(x, ndigits)</code>	<code>int</code> / <code>float</code>	banker's rounding
<code>divmod(a, b)</code>	<code>tuple</code>	$(a // b, a \% b)$
<code>enumerate(it, start=0)</code>	<code>enumerate</code>	yields (i, item)
<code>zip(*iters, strict=False)</code>	<code>zip</code>	<code>strict=True</code> raises on mismatch (3.10+)
<code>map(fn, *iters)</code>	<code>map</code>	lazy
<code>filter(fn, it)</code>	<code>filter</code>	lazy; <code>None</code> keeps truthy
<code>any(it)/all(it)</code>	<code>bool</code>	short-circuit
<code>sum(it, start=0)</code>	numeric	<code>math.fsum</code> for floats

5 REPL

Read-Eval-Print Loop — Python's interactive interpreter. Launch with `python` (or `python -i script.py` to run a script and then drop into a REPL with its globals already populated).

5.1 the underscore variable

In interactive mode, `_` is bound to the value of the last expression that wasn't `None`. Successive underscores `__` and `___` hold the previous two — useful for chaining when you forgot to assign:

```
1 >>> 2 + 2
2 4
3 >>> _ * 10
4 40
5 >>> _ + __
6 44
```

This binding is REPL-only. Inside a script, `_` is just the conventional name for “I don’t care about this value” (e.g. `for _ in range(n):`).

5.2 introspection

operation	description
<code>help(obj)</code>	interactive help / docstring
<code>dir(obj)</code>	list attributes (filter with <code>_</code> -prefix to drop dunder)
<code>type(obj)</code>	runtime type
<code>vars(obj)</code>	the <code>__dict__</code> of <code>obj</code>
<code>repr(obj)</code>	unambiguous string form (what the REPL prints)
<code>id(obj)</code>	CPython memory address — identity, not equality
<code>obj?</code>	IPython-only: docstring + signature
<code>obj??</code>	IPython-only: full source

5.3 invocation flags

command	effect
<code>python -i script.py</code>	run the script, then drop into the REPL with its namespace
<code>python -m module</code>	run a module as a script (e.g. <code>python -m http.server</code> , <code>python -m json.tool</code> , <code>python -m venv .venv</code>)
<code>python -c "code"</code>	execute a one-liner
<code>python -O</code>	strip <code>assert</code> and skip <code>__debug__</code> branches

5.4 environment

- `PYTHONSTARTUP`: path to a file run on every interactive REPL start — handy for default imports or aliases.
- `PYTHONDONTWRITEBYTECODE=1`: suppress `__pycache__` when fiddling with throwaway scripts.
- `PYTHONBREAKPOINT`: overrides what `breakpoint()` drops into (default `pdb.set_trace`; set to `ipdb.set_trace` for richer debugging or `0` to disable).
- exit with `exit()`, `quit()`, or `Ctrl-D` (`Ctrl-Z` `Enter` on Windows).

5.5 alternatives

- **IPython** — `?`/`??` introspection, magic commands (`%timeit`, `%run`, `%load_ext autoreload`; full list via `%lsmagic`), better tracebacks.
- **ptpython** — multiline editing, syntax highlighting, inline type hints.
- **bpython** — minimal cousin of IPython with autocomplete and rewind.
- `python -m asyncio` (3.8+) — REPL with top-level `await` for coroutine experimentation.