

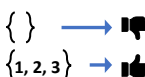

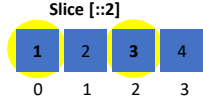
The Ultimate Python Cheat Sheet




Keywords

Keyword	Description	Code Examples
False, True	Boolean data type	False == (1 > 2) True == (2 > 1)
and, or, not	Logical operators → Both are true → Either is true → Flips Boolean	True and True # True True or False # True not False # True
break	Ends loop prematurely	while True: break # finite loop
continue	Finishes current loop iteration	while True: continue print("42") # dead code
class	Defines new class	class Coffee: # Define your class
def	Defines a new function or class method.	def say_hi(): print('hi')
if, elif, else	Conditional execution: - "if" condition == True? - "elif" condition == True? - Fallback: else branch	x = int(input("ur val:")) if x > 3: print("Big") elif x == 3: print("3") else: print("Small")
for, while	# For loop for i in [0,1,2]: print(i)	# While loop does same j = 0 while j < 3: print(j); j = j + 1
in	Sequence membership	42 in [2, 39, 42] # True
is	Same object memory location	y = x = 3 x is y # True [3] is [3] # False
None	Empty value constant	print() is None # True
lambda	Anonymous function	(lambda x: x+3)(3) # 6
return	Terminates function. Optional return value defines function result.	def increment(x): return x + 1 increment(4) # returns 5

Basic Data Structures

Type	Description	Code Examples
Boolean	The Boolean data type is either <code>True</code> or <code>False</code> . Boolean operators are ordered by priority: <code>not</code> → <code>and</code> → <code>or</code> 	<pre>## Evaluates to True: 1<2 and 0<=1 and 3>2 and 2>=2 and 1==1 and 1!=0 ## Evaluates to False: bool(None or 0 or 0.0 or '' or [] or {} or set()) Rule: None, 0, 0.0, empty strings, or empty container types evaluate to False</pre>
Integer, Float	An integer is a positive or negative number without decimal point such as 3. A float is a positive or negative number with floating point precision such as 3.1415926. Integer division rounds toward the smaller integer (example: 3//2==1).	<pre>## Arithmetic Operations x, y = 3, 2 print(x + y) # = 5 print(x - y) # = 1 print(x * y) # = 6 print(x / y) # = 1.5 print(x // y) # = 1 print(x % y) # = 1 print(-x) # = -3 print(abs(-x)) # = 3 print(int(3.9)) # = 3 print(float(3)) # = 3.0 print(x ** y) # = 9</pre> 
String	Python Strings are sequences of characters. String Creation Methods: 1. Single quotes >>> 'Yes' 2. Double quotes >>> "Yes" 3. Triple quotes (multi-line) >>> """Yes We Can""" 4. String method >>> str(5) == '5' True 5. Concatenation >>> "Ma" + "hatma" 'Mahatma' Whitespace chars: Newline \n, Space \s, Tab \t	<pre>## Indexing and Slicing s = "The youngest pope was 11 years" s[0] # 'T' s[1:3] # 'he' s[-3:-1] # 'ar' s[-3:] # 'ars' Slice [::2] 1 2 3 4 0 1 2 3 x = s.split() x[-2] + " " + x[2] + "s" # '11 popes' ## String Methods y = " Hello world\t\n " y.strip() # Remove Whitespace "HI".lower() # Lowercase: 'hi' "hi".upper() # Uppercase: 'HI' "hello".startswith("he") # True "hello".endswith("lo") # True "hello".find("ll") # Match at 2 "cheat".replace("ch", "m") # 'meat' ''.join(["F", "B", "I"]) # 'FBI' len("hello world") # Length: 15 "ear" in "earth" # True</pre> 

Complex Data Structures

Type	Description	Example
List	Stores a sequence of elements. Unlike strings, you can modify list objects (they're <i>mutable</i>).	<pre>l = [1, 2, 2] print(len(l)) # 3</pre> 
Adding elements	Add elements to a list with (i) <code>append</code> , (ii) <code>insert</code> , or (iii) list concatenation.	<pre>[1, 2].append(4) # [1, 2, 4] [1, 4].insert(1,9) # [1, 9, 4] [1, 2] + [4] # [1, 2, 4]</pre>
Removal	Slow for lists	<pre>[1, 2, 2, 4].remove(1) # [2, 2, 4]</pre>
Reversing	Reverses list order	<pre>[1, 2, 3].reverse() # [3, 2, 1]</pre>
Sorting	Sorts list using fast Timsort	<pre>[2, 4, 2].sort() # [2, 2, 4]</pre>
Indexing	Finds the first occurrence of an element & returns index. Slow worst case for whole list traversal.	<pre>[2, 2, 4].index(2) # index of item 2 is 0 [2, 2, 4].index(2,1) # index of item 2 after pos 1 is 1</pre>
Stack	Use Python lists via the list operations <code>append()</code> and <code>pop()</code>	<pre>stack = [3] stack.append(42) # [3, 42] stack.pop() # 42 (stack: [3]) stack.pop() # 3 (stack: [])</pre>
Set	An unordered collection of unique elements (<i>at-most-once</i>) → fast membership $O(1)$	<pre>basket = {'apple', 'eggs', 'banana', 'orange'} same = set(['apple', 'eggs', 'banana', 'orange'])</pre>

Type	Description	Example
Dictionary	Useful data structure for storing (key, value) pairs	<pre>cal = {'apple': 52, 'banana': 89, 'choco': 546} # calories</pre>
Reading and writing elements	Read and write elements by specifying the key within the brackets. Use the keys() and values() functions to access all keys and values of the dictionary	<pre>print(cal['apple'] < cal['choco']) # True cal['cappu'] = 74 print(cal['banana'] < cal['cappu']) # False print('apple' in cal.keys()) # True print(52 in cal.values()) # True</pre>
Dictionary Iteration	You can access the (key, value) pairs of a dictionary with the items() method.	<pre>for k, v in cal.items(): print(k) if v > 500 else '' # 'choco'</pre>
Membership operator	Check with the in keyword if set, list, or dictionary contains an element. Set membership is faster than list membership.	<pre>basket = {'apple', 'eggs', 'banana', 'orange'} print('eggs' in basket) # True print('mushroom' in basket) # False</pre>
List & set comprehension	List comprehension is the concise Python way to create lists. Use brackets plus an expression, followed by a for clause. Close with zero or more for or if clauses. Set comprehension works similar to list comprehension.	<pre>l = ['hi' + x for x in ['Alice', 'Bob', 'Pete']] # ['Hi Alice', 'Hi Bob', 'Hi Pete'] l2 = [x * y for x in range(3) for y in range(3) if x>y] # [0, 0, 2] squares = {x**2 for x in [0,2,4] if x < 4} # {0, 4}</pre>