Sciencestical Day-5: **100** handy Python **shortcuts**

1. Swapping variables

In [1]:
```
a, b = 1, 2
a, b = b, a
print(f"Swapped values: a = {a}, b = {b}")
```

```
Swapped values: a = 2, b = 1
```

2. List comprehensions

In [2]:
```
squares = [x**2 for x in range(10)]
print(f"Squares: {squares}")
```

```
Squares: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

3. Dictionary comprehensions

In [3]:
```
square_dict = {x: x**2 for x in range(10)}
print(f"Square dict: {square_dict}")
```

```
Square dict: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

4. Set comprehensions

In [4]:
```
square_set = {x**2 for x in range(10)}
print(f"Square set: {square_set}")
```

```
Square set: {0, 1, 64, 4, 36, 9, 16, 49, 81, 25}
```

5. Inline if-else

In [5]:
```
age = 18
status = "adult" if age >= 18 else "minor"
print(f"Status: {status}")
```

```
Status: adult
```

6. Multiple assignment

In [6]:
```
name, age, city = "Bernard", 27, "Groton"
print(f"Name: {name}, Age: {age}, City: {city}")
```

```
Name: Bernard, Age: 27, City: Groton
```

7. Unpacking sequences

In [7]:
```
data = ('Akaawase', 27, 'Makurdi')
name, age, city = data
print(f"Name: {name}, Age: {age}, City: {city}")
```

```
Name: Akaawase, Age: 27, City: Makurdi
```

8. Merging dictionaries (Python 3.9+)

In [8]:
```
dict1 = {'a': 1, 'b': 2}
dict2 = {'b': 3, 'c': 4}
merged_dict = dict1 | dict2
print(f"Merged dictionary: {merged_dict}")
```

```
Merged dictionary: {'a': 1, 'b': 3, 'c': 4}
```

9. Using zip to iterate over multiple sequences

In [9]:
```
names = ["Akaawase", "Bob", "Charlie"]
ages = [27, 25, 35]
for name, age in zip(names, ages):
    print(f"{name} is {age} years old")
```

```
Akaawase is 27 years old
Bob is 25 years old
Charlie is 35 years old
```

10. Enumerate for index and value

In [10]:
```
fruits = ['apple', 'banana', 'cherry']
for index, fruit in enumerate(fruits):
    print(f"Index {index}: {fruit}")
```

```
Index 0: apple
Index 1: banana
Index 2: cherry
```

11. Lambda functions for short anonymous functions

In [11]:
```
add = lambda x, y: x + y
print(f"Sum: {add(5, 3)}")
```

```
Sum: 8
```

12. Using * to unpack arguments

In [12]:
```
def multiply(x, y, z):
    return x * y * z

nums = (2, 3, 4)
print(f"Product: {multiply(*nums)}")
```

```
Product: 24
```

13. Using ** to unpack keyword arguments

In [13]:
```
def greet(name, greeting="Hello"):
    return f"{greeting}, {name}!"

params = {"name": "Teryima", "greeting": "Hi"}
print(greet(**params))
```

```
Hi, Teryima!
```

14. List slicing

In [14]:
```
lst = [1, 2, 3, 4, 5]
print(f"Sliced list (1:3): {lst[1:3]}")
print(f"Sliced list (start to 3): {lst[:3]}")
print(f"Sliced list (3 to end): {lst[3:]}")
print(f"Reversed list: {lst[::-1]}")
```

```
Sliced list (1:3): [2, 3]
Sliced list (start to 3): [1, 2, 3]
Sliced list (3 to end): [4, 5]
Reversed list: [5, 4, 3, 2, 1]
```

15. Using Counter for frequency counting

In [15]:
```
from collections import Counter
words = ["apple", "banana", "apple", "orange", "banana", "apple"]
word_count = Counter(words)
print(f"Word count: {word_count}")
```

```
Word count: Counter({'apple': 3, 'banana': 2, 'orange': 1})
```

16. ChainMap for combining multiple dictionaries

In [16]:
```
from collections import ChainMap
dict1 = {'a': 1, 'b': 2}
dict2 = {'b': 3, 'c': 4}
combined = ChainMap(dict1, dict2)
print(f"Combined ChainMap: {combined}")
```

```
Combined ChainMap: ChainMap({'a': 1, 'b': 2}, {'b': 3, 'c': 4})
```

17. Named tuples for simple classes

In [17]:
```
from collections import namedtuple
Point = namedtuple('Point', 'x y')
p = Point(11, 22)
print(f"Point: {p}, x: {p.x}, y: {p.y}")
```

```
Point: Point(x=11, y=22), x: 11, y: 22
```

18. Using itertools for efficient looping

In [18]:
```
from itertools import permutations, combinations
print(f"Permutations of 'AB': {list(permutations('AB'))}")
print(f"Combinations of 'AB': {list(combinations('AB', 1))}")
```

```
Permutations of 'AB': [('A', 'B'), ('B', 'A')]
Combinations of 'AB': [('A',), ('B',)]
```

19. Context managers for resource management

In [19]:
```
with open('data/example.txt', 'w') as f:
    f.write("Hello, world!")
```

20. f-strings for formatted strings (Python 3.6+)

In [20]:
```
name = "Akaawase"
age = 27
print(f"My name is {name} and I am {age} years old.")
```

```
My name is Akaawase and I am 27 years old.
```

21. Ternary conditional operator

In [21]:
```
result = "Even" if age % 2 == 0 else "Odd"
print(f"Age is {result}")
```

```
Age is Odd
```

22. Flatten a list of lists

In [22]:
```
nested_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flat_list = [item for sublist in nested_list for item in sublist]
print(f"Flat list: {flat_list}")
```

```
Flat list: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

23. Transpose a matrix

In [23]:
```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
transposed = list(zip(*matrix))
print(f"Transposed matrix: {transposed}")
```

```
Transposed matrix: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

24. Using defaultdict for missing keys

In [24]:
```
from collections import defaultdict
dd = defaultdict(int)
dd['key'] += 1
print(f"defaultdict: {dd}")
```

```
defaultdict: defaultdict(<class 'int'>, {'key': 1})
```

25. Using get() with dictionaries

In [25]:
```
d = {'a': 1, 'b': 2}
value = d.get('c', 0)
print(f"Value for key 'c': {value}")
```

```
Value for key 'c': 0
```

26. Using slice object

In [26]:
```
data = [0, 1, 2, 3, 4, 5, 6]
s = slice(1, 5, 2)
print(f"Sliced data: {data[s]}")
```

```
Sliced data: [1, 3]
```

27. Using any() and all()

In [27]:
```
conditions = [True, False, True]
print(f"Any true: {any(conditions)}")
print(f"All true: {all(conditions)}")
```

```
Any true: True
All true: False
```

28. Using enumerate to create a dict

In [28]:
```
elements = ['a', 'b', 'c']
enum_dict = dict(enumerate(elements))
print(f"Enumerate dict: {enum_dict}")
```

```
Enumerate dict: {0: 'a', 1: 'b', 2: 'c'}
```

29. String join method

In [29]:
```
words = ['Hello', 'Linkedin']
sentence = ' '.join(words)
print(f"Sentence: {sentence}")
```

```
Sentence: Hello Linkedin
```

30. Reverse a string

In [30]:
```
reversed_str = 'hello'[::-1]
print(f"Reversed string: {reversed_str}")
```

```
Reversed string: olleh
```

31. Check for palindrome

In [31]:
```
word = "racecar"
is_palindrome = word == word[::-1]
print(f"Is palindrome: {is_palindrome}")
```

```
Is palindrome: True
```

32. Using filter with lambda

In [32]:
```
nums = [1, 2, 3, 4, 5, 6]
even_nums = list(filter(lambda x: x % 2 == 0, nums))
print(f"Even numbers: {even_nums}")
```

```
Even numbers: [2, 4, 6]
```

33. Using map with lambda

In [33]:
```
squared_nums = list(map(lambda x: x ** 2, nums))
print(f"Squared numbers: {squared_nums}")
```

```
Squared numbers: [1, 4, 9, 16, 25, 36]
```

34. Using reduce for cumulative operation

In [34]:
```
from functools import reduce
product = reduce(lambda x, y: x * y, nums)
print(f"Product of numbers: {product}")
```

```
Product of numbers: 720
```

35. Using sorted with custom key

In [35]:
```
words = ["apple", "banana", "cherry"]
sorted_words = sorted(words, key=lambda x: len(x))
print(f"Sorted by length: {sorted_words}")
```

```
Sorted by length: ['apple', 'banana', 'cherry']
```

36. Checking for subset and superset

In [36]:
```
set_a = {1, 2, 3}
set_b = {1, 2}
is_subset = set_b.issubset(set_a)
is_superset = set_a.issuperset(set_b)
print(f"Is subset: {is_subset}, Is superset: {is_superset}")
```

```
Is subset: True, Is superset: True
```

37. Using itertools.chain for flattening lists

In [37]:
```
from itertools import chain
nested = [[1, 2, 3], [4, 5], [6, 7, 8]]
flat = list(chain(*nested))
print(f"Flattened list: {flat}")
```

```
Flattened list: [1, 2, 3, 4, 5, 6, 7, 8]
```

38. Using a generator expression

In [38]:
```
gen_exp = (x**2 for x in range(10))
print(f"Generator expression: {list(gen_exp)}")
```

```
Generator expression: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

39. Reading a file line by line

In [39]:
```
with open('data/example.txt', 'r') as f:
    lines = f.readlines()
print(f"File lines: {lines}")
```

```
File lines: ['Hello, world!']
```

40. Writing multiple lines to a file

In [40]:
```
lines_to_write = ["Line 1", "Line 2", "Line 3"]
with open('data/example0.txt', 'w') as f:
    f.writelines('\n'.join(lines_to_write))
```

41. Simple class with **str** method

In [41]:
```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name}, {self.age} years old"

person = Person("Akaawase", 27)
print(person)
```

```
Akaawase, 27 years old
```

42. Using dataclasses (Python 3.7+)

In [42]:
```
from dataclasses import dataclass

@dataclass
class Car:
    make: str
    model: str
    year: int

car = Car(make="Toyota", model="Corolla", year=2020)
print(car)
```

```
Car(make='Toyota', model='Corolla', year=2020)
```

43. Using **slots** to save memory

In [43]:
```
class PointWithSlots:
    __slots__ = ['x', 'y']

    def __init__(self, x, y):
        self.x = x
        self.y = y

p = PointWithSlots(10, 20)
print(f"Point with slots: ({p.x}, {p.y})")
```

```
Point with slots: (10, 20)
```

44. Using @property decorator

In [44]:
```python
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if value >= 0:
            self._radius = value
        else:
            raise ValueError("Radius must be non-negative")

circle = Circle(5)
print(f"Circle radius: {circle.radius}")
circle.radius = 10
print(f"Updated circle radius: {circle.radius}")
```

```
Circle radius: 5
Updated circle radius: 10
```

45. Using collections.deque for fast appends and pops

In [45]:
```python
from collections import deque
dq = deque([1, 2, 3])
dq.appendleft(0)
dq.append(4)
print(f"Deque: {dq}")
```

```
Deque: deque([0, 1, 2, 3, 4])
```

46. Using bisect to maintain a sorted list

In [46]:
```python
import bisect
sorted_list = [1, 2, 4, 5]
bisect.insort(sorted_list, 3)
print(f"Sorted list with insort: {sorted_list}")
```

```
Sorted list with insort: [1, 2, 3, 4, 5]
```

47. Using heapq for a priority queue

In [47]:
```python
import heapq
heap = [3, 1, 4, 1, 5, 9]
heapq.heapify(heap)
print(f"Heapified list: {heap}")
heapq.heappush(heap, 2)
print(f"Heap after push: {heap}")
smallest = heapq.heappop(heap)
print(f"Smallest element: {smallest}")
```

```
Heapified list: [1, 1, 4, 3, 5, 9]
Heap after push: [1, 1, 2, 3, 5, 9, 4]
Smallest element: 1
```

48. Using lru_cache for memoization

In [48]:
```python
from functools import lru_cache

@lru_cache(maxsize=None)
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(f"Fibonacci(10): {fibonacci(10)}")
```

```
Fibonacci(10): 55
```

49. Using partial to fix arguments

In [49]:
```
from functools import partial

def power(base, exponent):
    return base ** exponent

square = partial(power, exponent=2)
print(f"Square of 4: {square(4)}")
```

```
Square of 4: 16
```

50. Using namedtuple for structured data

In [50]:
```
from collections import namedtuple

Person = namedtuple('Person', 'name age')
alice = Person(name="Alice", age=30)
print(f"Namedtuple person: {alice}")
```

```
Namedtuple person: Person(name='Alice', age=30)
```

51. Using frozenset for immutable sets

In [51]:
```
immutable_set = frozenset([1, 2, 3])
print(f"Frozenset: {immutable_set}")
```

```
Frozenset: frozenset({1, 2, 3})
```

52. Using islice for slicing iterators

In [52]:
```
from itertools import islice

iterable = iter(range(10))
sliced = list(islice(iterable, 2, 6))
print(f"Sliced iterator: {sliced}")
```

```
Sliced iterator: [2, 3, 4, 5]
```

53. Using groupby to group elements

In [53]:
```
from itertools import groupby

data = sorted([('a', 1), ('b', 2), ('a', 3)])
grouped = {k: list(v) for k, v in groupby(data, key=lambda x: x[0])}
print(f"Grouped data: {grouped}")
```

```
Grouped data: {'a': [('a', 1), ('a', 3)], 'b': [('b', 2)]}
```

54. Using compress to filter by a mask

In [54]:
```
from itertools import compress
data = range(10)
selectors = [1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
filtered = list(compress(data, selectors))
print(f"Filtered by mask: {filtered}")
```

```
Filtered by mask: [0, 2, 4, 6, 8]
```

55. Using combinations_with_replacement

In [55]:
```
from itertools import combinations_with_replacement

comb_wr = list(combinations_with_replacement('ABC', 2))
print(f"Combinations with replacement: {comb_wr}")
```

```
Combinations with replacement: [('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'B'), ('B', 'C'), ('C', 'C')]
```

56. Simple recursive function

In [56]:
```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)

print(f"Factorial(5): {factorial(5)}")
```

```
Factorial(5): 120
```

57. Using contextlib for simple context managers

In [57]:
```python
from contextlib import contextmanager

@contextmanager
def simple_context():
    print("Entering")
    yield
    print("Exiting")

with simple_context():
    print("Inside context")
```

```
Entering
Inside context
Exiting
```

58. Using counter with most_common

In [58]:
```python
from collections import Counter

counter = Counter("abracadabra")
print(f"Most common elements: {counter.most_common(3)}")
```

```
Most common elements: [('a', 5), ('b', 2), ('r', 2)]
```

59. Using defaultdict for list

In [59]:
```python
d = defaultdict(list)
d['key'].append('value')
print(f"defaultdict with list: {d}")
```

```
defaultdict with list: defaultdict(<class 'list'>, {'key': ['value']})
```

60. Using tarfile to compress files

In [60]:
```python
import tarfile

with tarfile.open('data/sample.tar.gz', 'w:gz') as tar:
    tar.add('data/example0.txt')
```

61. Using zipfile to compress files

In [61]:
```python
import zipfile

with zipfile.ZipFile('data/sample.zip', 'w') as zipf:
    zipf.write('data/example0.txt')
```

62. Using pathlib for file paths

In [62]:
```python
from pathlib import Path

path = Path('data/example.txt')
print(f"File name: {path.name}")
print(f"File suffix: {path.suffix}")
print(f"File stem: {path.stem}")
```

```
File name: example.txt
File suffix: .txt
File stem: example
```

63. Using tempfile for temporary files

In [63]:
```python
import tempfile

with tempfile.TemporaryFile() as tempf:
    tempf.write(b'Some data')
    tempf.seek(0)
    print(f"Temporary file content: {tempf.read()}")
```

```
Temporary file content: b'Some data'
```

64. Using Enum for enumerations

In [64]:
```
from enum import Enum

class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3

print(f"Color.RED: {Color.RED}")
```

```
Color.RED: Color.RED
```

65. Using type hints for function annotations

In [65]:
```
def greet(name: str) -> str:
    return f"Hello, {name}"

print(greet("Bernard"))
```

```
Hello, Bernard
```

66. Using dataclasses for boilerplate code

In [66]:
```
from dataclasses import dataclass

@dataclass
class Point:
    x: int
    y: int

point = Point(10, 20)
print(f"Dataclass point: {point}")
```

```
Dataclass point: Point(x=10, y=20)
```

67. Using **repr** for object representation

In [67]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"Point({self.x}, {self.y})"

p = Point(10, 20)
print(p)
```

```
Point(10, 20)
```

68. Using **str** for readable object representation

In [68]:
```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name}, {self.age} years old"

print(Person("Alex", 60))
```

```
Alex, 60 years old
```

69. Using slots to save memory

In [69]:
```
class Point:
    __slots__ = ['x', 'y']

    def __init__(self, x, y):
        self.x = x
        self.y = y

p = Point(10, 20)
print(p.x, p.y)
```

```
10 20
```

70. Using generators for memory efficiency

In [70]:
```python
def generate_numbers(n):
    for i in range(n):
        yield i

gen = generate_numbers(10)
print(list(gen))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

71. Using asyncio for asynchronous programming

In [71]:
```python
import nest_asyncio
import asyncio

nest_asyncio.apply()

async def say_hello():
    print("Hello")
    await asyncio.sleep(1)
    print("World")

asyncio.run(say_hello())
```

```
Hello
World
```

72. Using dataclass for simple data containers

In [72]:
```python
from dataclasses import dataclass

@dataclass
class Point:
    x: int
    y: int

p = Point(10, 20)
print(p)
```

```
Point(x=10, y=20)
```

73. Using collections.Counter for counting

In [73]:
```python
from collections import Counter

words = ["apple", "banana", "apple"]
counter = Counter(words)
print(counter)
```

```
Counter({'apple': 2, 'banana': 1})
```

74. Using itertools.chain for flattening

In [74]:
```python
from itertools import chain

lists = [[1, 2, 3], [4, 5], [6, 7, 8]]
flat = list(chain.from_iterable(lists))
print(flat)
```

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

75. Using partial for function customization

In [75]:
```python
from functools import partial

def power(base, exponent):
    return base ** exponent

square = partial(power, exponent=2)
print(square(4))
```

```
16
```

76. Using map for element-wise operations

In [76]:
```
numbers = [1, 2, 3]
squares = list(map(lambda x: x**2, numbers))
print(squares)
```

```
[1, 4, 9]
```

77. Using filter for filtering elements

In [77]:
```
numbers = [1, 2, 3, 4, 5, 6]
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(evens)
```

```
[2, 4, 6]
```

78. Using reduce for aggregation

In [78]:
```
from functools import reduce

numbers = [1, 2, 3, 4, 5]
total = reduce(lambda x, y: x + y, numbers)
print(total)
```

```
15
```

79. Using namedtuple for structured data

In [79]:
```
from collections import namedtuple

Point = namedtuple('Point', ['x', 'y'])
p = Point(10, 20)
print(p)
```

```
Point(x=10, y=20)
```

80. Using frozenset for immutable sets

In [80]:
```
fs = frozenset([1, 2, 3])
print(fs)
```

```
frozenset({1, 2, 3})
```

81. Using heapq for priority queue

In [81]:
```
import heapq

numbers = [5, 7, 9, 1, 3]
heapq.heapify(numbers)
print(numbers)
```

```
[1, 3, 9, 7, 5]
```

82. Using bisect for binary search

In [82]:
```
import bisect

numbers = [1, 2, 4, 5]
bisect.insort(numbers, 3)
print(numbers)
```

```
[1, 2, 3, 4, 5]
```

83. Using itertools.islice for slicing iterators

In [83]:
```
from itertools import islice

iterable = range(10)
sliced = list(islice(iterable, 2, 6))
print(sliced)
```

```
[2, 3, 4, 5]
```

84. Using zip_longest for padding

In [84]:
```
from itertools import zip_longest

a = [1, 2]
b = [3, 4, 5]
zipped = list(zip_longest(a, b, fillvalue=0))
print(zipped)
```

```
[(1, 3), (2, 4), (0, 5)]
```

85. Using permutations for permutations

In [85]:
```
from itertools import permutations

perm = list(permutations('ABC'))
print(perm)
```

```
[('A', 'B', 'C'), ('A', 'C', 'B'), ('B', 'A', 'C'), ('B', 'C', 'A'), ('C', 'A', 'B'), ('C', 'B', 'A')]
```

86. Using combinations for combinations

In [86]:
```
from itertools import combinations

comb = list(combinations('ABC', 2))
print(comb)
```

```
[('A', 'B'), ('A', 'C'), ('B', 'C')]
```

87. Using combinations_with_replacement

In [87]:
```
from itertools import combinations_with_replacement

comb_wr = list(combinations_with_replacement('ABC', 2))
print(comb_wr)
```

```
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'B'), ('B', 'C'), ('C', 'C')]
```

88. Using lru_cache for memoization

In [88]:
```
from functools import lru_cache

@lru_cache(maxsize=None)
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(10))
```

```
55
```

89. Using contextlib for context managers

In [89]:
```
from contextlib import contextmanager

@contextmanager
def simple_context():
    print("Entering")
    yield
    print("Exiting")

with simple_context():
    print("Inside")
```

```
Entering
Inside
Exiting
```

90. Using deque for fast appends/pops

In [90]:
```python
from collections import deque

dq = deque([1, 2, 3])
dq.appendleft(0)
dq.append(4)
print(dq)
```

```
deque([0, 1, 2, 3, 4])
```

91. Using Enum for enumerations

In [91]:
```python
from enum import Enum

class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3

print(Color.RED)
```

```
Color.RED
```

92. Using pathlib for file paths

In [92]:
```python
from pathlib import Path

path = Path('data/example.txt')
print(path.name, path.suffix)
```

```
example.txt .txt
```

93. Using tempfile for temp files

In [93]:
```python
import tempfile

with tempfile.TemporaryFile() as tempf:
    tempf.write(b'Some data')
    tempf.seek(0)
    print(tempf.read())
```

```
b'Some data'
```

94. Using type hints for annotations

In [94]:
```python
def greet(name: str) -> str:
    return f"Hello, {name}"

print(greet("Akaawase"))
```

```
Hello, Akaawase
```

95. Using dataclass for data containers

In [95]:
```python
from dataclasses import dataclass

@dataclass
class Point:
    x: int
    y: int

p = Point(10, 20)
print(p)
```

```
Point(x=10, y=20)
```

96. Using collections.ChainMap for multiple dicts

In [96]:
```python
from collections import ChainMap

dict1 = {'a': 1}
dict2 = {'b': 2}
chain = ChainMap(dict1, dict2)
print(chain)
```

```
ChainMap({'a': 1}, {'b': 2})
```

97. Using tarfile for compression

In [97]:
```
import tarfile

with tarfile.open('data/sample.tar.gz', 'w:gz') as tar:
    tar.add('data/example0.txt')
```

98. Using zipfile for compression

In [98]:
```
import zipfile

with zipfile.ZipFile('data/sample.zip', 'w') as zipf:
    zipf.write('data/example.txt')
```

99. Using **slots** to save memory

In [99]:
```
class Point:
    __slots__ = ['x', 'y']

    def __init__(self, x, y):
        self.x = x
        self.y = y

p = Point(10, 20)
print(p.x, p.y)
```

```
10 20
```

100. Using generators for memory efficiency

In [100…
```
def generate_numbers(n):
    for i in range(n):
        yield i

gen = generate_numbers(10)
print(list(gen))

if __name__ == "__main__":
    print("This script contains 100 handy Python shortcuts.")
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
This script contains 100 handy Python shortcuts.
```

In [ ]: