You have an N-element tuple or sequence that you would like to unpack into a collection of N variables.

| # A tuple of three elements my\_tuple = (1, 2, 3)  # Unpacking into three separate variables a, b, c = my\_tuple  print(a) # Output: 1 print(b) # Output: 2 print(c) # Output: 3 |
| --- |

| data = [ 'ACME', 50, 91.1, (2012, 12, 21) ] name, shares, price, date = data |
| --- |

| name, shares, price, (year, mon, day) = data |
| --- |

| data = [("Tokyo", 37400068), ("Delhi", 28514000), ("Shanghai", 25582000)] for city, population in data:  print(f"The population of {city} is {population}") |
| --- |

| for name, age in [('Alice', 24), ('Bob', 28)]:  print(f"{name} is {age} years old.") |
| --- |

If there is a mismatch in the number of elements, you’ll get an error. For example:

| p = (4, 5) x, y, z = p Traceback (most recent call last): File "", line 1, in ValueError: need more than 2 values to unpack |
| --- |

Unpacking actually works with any object that happens to be iterable, not just tuples or lists. This includes strings, files, iterators, and generators. For example:

| s = 'Hello'  a, b, c, d, e = s |
| --- |

When unpacking, you may sometimes want to discard certain values. Python has no special syntax for this, but you can often just pick a throwaway variable name for it. For example:

| data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]  \_, shares, price, \_ = data |
| --- |

Using \* to grab excess items

Defining function parameters with **\*args** to grab arbitrary excess arguments is a classic Python feature.

| a, b, \*rest = range(5) a, b, \*rest = range(3) a, b, \*rest = range(2) |
| --- |

In the context of parallel assignment, the \* prefix can be applied to exactly one variable, but it can appear in any position

| a, \*body, c, d = range(5) \*head, b, c, d = range(5) |
| --- |

Named tuples The collections.namedtuple function is a factory that produces subclasses of tuple enhanced with field names and a class name — which helps debugging.

| from collections import namedtuple  # Create a named tuple type Person = namedtuple('Person', 'name age gender') |
| --- |

| # Creating an instance of the Person named tuple person = Person(name="Alice", age=30, gender="Female")  print(person) # Output: Person(name='Alice', age=30, gender='Female') |
| --- |

| print(person.name) # Output: Alice print(person.age) # Output: 30 print(person[1]) # Also output: 30 |
| --- |