

A tuple is a list that can't change, which is what lets it be a dict key. Examples use `t = (10, 20, 30, 40)`.

CREATE

<code>1, 2</code>	→ <code>(1, 2)</code>
the comma makes a tuple, not the brackets	
<code>(42,)</code>	→ <code>(42,)</code>
one item needs the trailing comma	
<code>()</code>	→ <code>()</code>
empty	
<code>tuple("abc")</code>	→ <code>('a', 'b', 'c')</code>
any iterable to a tuple	
<code>tuple([1, 2])</code>	→ <code>(1, 2)</code>
freeze a list	
<code>tuple(range(3))</code>	→ <code>(0, 1, 2)</code>

READ

<code>t = (10, 20, 30, 40)</code>	
<code>t[0]</code>	→ <code>10</code>
first	
<code>t[-1]</code>	→ <code>40</code>
last	
<code>t[1:3]</code>	→ <code>(20, 30)</code>
a slice is a tuple too	
<code>t[::-1]</code>	→ <code>(40, 30, 20, 10)</code>
reversed	
<code>len(t)</code>	→ <code>4</code>
<code>30 in t</code>	→ <code>True</code>
<code>t.count(20)</code>	→ <code>1</code>
<code>count()</code> and <code>index()</code> are the only methods	
<code>t.index(30)</code>	→ <code>2</code>
<code>min(t), max(t), sum(t)</code>	→ <code>(10, 40, 100)</code>

BUILD A NEW ONE

<code>t = (10, 20, 30, 40)</code>	
<code>(1, 2) + (3,)</code>	→ <code>(1, 2, 3)</code>
+ joins into a new tuple	
<code>(0,) * 3</code>	→ <code>(0, 0, 0)</code>
<code>t[1:] + (99,) + t[2:]</code>	→ <code>(10, 99, 30, 40)</code>
"change" an item by building a new tuple	
<code>list(t)</code>	→ <code>[10, 20, 30, 40]</code>
lots of edits? convert to a list	
<code>sorted((3, 1, 2))</code>	→ <code>[1, 2, 3]</code>
<code>sorted()</code> always returns a list	
<code>tuple(sorted((3, 1, 2)))</code>	→ <code>(1, 2, 3)</code>
sorted, back as a tuple	

COMPARE

<code>(1, 2) == (1, 2)</code>	→ <code>True</code>
<code>(1, 9) < (2, 0)</code>	→ <code>True</code>
item by item, left to right	
<code>(1, 2) < (1, 2, 0)</code>	→ <code>True</code>
a shorter prefix sorts first	
<code>max(["b", 2], ("a", 9))</code>	→ <code>('b', 2)</code>
first items decide, ties go to the next	

UNPACK

<code>x, y = (3, 4)</code>	
<code>y</code>	
→ <code>4</code>	
one name per item	
<code>first, *rest = (1, 2, 3, 4)</code>	
<code>rest</code>	
→ <code>[2, 3, 4]</code>	
* collects the rest, as a list	
<code>*init, last = (1, 2, 3, 4)</code>	
<code>init</code>	
→ <code>[1, 2, 3]</code>	
<code>name, _, age = ("Ada", "x", 36)</code>	
<code>age</code>	
→ <code>36</code>	
_ for an item you don't need	
<code>(a, b), c = (1, 2), 3</code>	
<code>b</code>	
→ <code>2</code>	
nested shapes unpack too	
<code>a, b = 1, 2</code>	
<code>a, b = b, a</code>	
<code>a</code>	
→ <code>2</code>	
swap without a temporary variable	
<code>for i, ch in enumerate("ab"):</code>	
<code>print(i, ch)</code>	
loops hand you tuples to unpack	

FUNCTIONS

<code>def min_max(xs):</code>	
<code>return min(xs), max(xs)</code>	
return several values as one tuple	
<code>lo, hi = min_max([3, 1, 2])</code>	
<code>hi</code>	
→ <code>3</code>	
unpack the result straight away	
<code>divmod(17, 5)</code>	→ <code>(3, 2)</code>
built-ins do it too	
<code>list(zip('ab', (1, 2)))</code>	
→ <code>[('a', 1), ('b', 2)]</code>	
zip pairs things up as tuples	
<code>list({'a': 1}.items())</code>	→ <code>[('a', 1)]</code>
dict items are (key, value) tuples	
<code>pow(*(2, 10))</code>	→ <code>1024</code>
* spreads a tuple into arguments	
<code>def total(*nums):</code>	
<code>return sum(nums)</code>	
*nums gathers the extra arguments as a tuple	
<code>total(1, 2, 3)</code>	→ <code>6</code>

AS KEYS AND SET MEMBERS

<code>dist = {"Paris", "Rome"}: 1106</code>	
<code>dist[("Paris", "Rome")]</code>	→ <code>1106</code>
a tuple can be a dict key	
<code>len({(0, 0), (0, 0), (1, 2)})</code>	→ <code>2</code>
or a set member: handy for grid points	
<code>tuple(sorted(['b', 'a']))</code>	→ <code>('a', 'b')</code>
a list can't be a key: make it a tuple	
<code>{[1, 2]: 'x'}</code>	→ raises <code>TypeError</code>
lists are unhashable	
<code>hash((1, [2]))</code>	→ raises <code>TypeError</code>
only works if every item is hashable too	

NAMEDTUPLE

<code>from collections import namedtuple</code>	
<code>Point = namedtuple("Point", ["x", "y"])</code>	
<code>p = Point(3, 4)</code>	
<code>p</code>	→ <code>Point(x=3, y=4)</code>
prints with its field names	
<code>p.x</code>	→ <code>3</code>
read by name	
<code>p[1]</code>	→ <code>4</code>
or by position	
<code>p._replace(x=0)</code>	→ <code>Point(x=0, y=4)</code>
a copy with one field changed	
<code>p._asdict()</code>	→ <code>{'x': 3, 'y': 4}</code>
<code>p == (3, 4)</code>	→ <code>True</code>
still a plain tuple underneath	
<code>Point._fields</code>	→ <code>('x', 'y')</code>
the field names	

TYPING.NAMEDTUPLE

<code>from typing import NamedTuple</code>	
<code>class Point(NamedTuple):</code>	
<code> x: int</code>	
<code> y: int = 0</code>	
the same, with types and defaults	
<code>Point(5)</code>	→ <code>Point(x=5, y=0)</code>

TUPLE OR LIST?

<code>point = (3, 4)</code>	
tuple: fixed size, each slot means something	
<code>scores = [7, 9, 4]</code>	
list: a growing pile of the same kind	

WATCH OUT

<code>t = (10, 20, 30, 40)</code>	
<code>("a")</code>	→ <code>'a'</code>
no comma, no tuple: write ("a,")	
<code>x = 1,</code>	
<code>x</code>	
→ <code>(1,)</code>	
a stray trailing comma makes a tuple	
<code>t[0] = 99</code>	
<code>TypeError: tuples can't change; build a new one</code>	
<code>t.append(50)</code>	
<code>AttributeError: no append. Use t = t + (50,)</code>	
<code>a, b = (1, 2, 3)</code>	
<code>ValueError: too many values; a, b, *_ = drops the rest</code>	
<code>u = ([1], 2)</code>	
<code>u[0].append(9)</code>	
<code>u</code>	
→ <code>([1, 9], 2)</code>	
the tuple is fixed, but a list inside it can change	
<code>tuple("hi")</code>	→ <code>('h', 'i')</code>
tuple() splits a string: write ("hi",)	
<code>(1, 2) == [1, 2]</code>	→ <code>False</code>
a tuple never equals a list	