INF1-IP

Introduction to Programming

Lecture 12 – Collections 2: Tuples and collection operations

Matteo Degiacomi

THE UNIVERSITY of EDINBURGH
informatics

> Questions open from past lectures

- Why can a string be a dictionary key, but a list cannot?
- What happens when a list contains other lists?
- How do we loop over two lists at once?

Today:

- Hashability, and a fourth collection type: **tuples**
- `enumerate()`, `zip()`, `split()` and `join()`
- Two-dimensional lists and slicing
- Comprehensions: a loop in one line



The reason for this image will become apparent in two slides

> Finding an item: who is faster?

How long does it take, on average, to find whether an item exists in a list, a set, or a dictionary?

collection size	`List` (item exists)	`List` (item absent)	`set`	`dict`
1,000	6 µs	12 µs	0.040 µs	0.046 µs
100,000	600 µs	1,200 µs	0.040 µs	0.043 µs
1,000,000	6,000 µs	12,000 µs	0.042 µs	0.041 µs

With a million items, the sets and dictionaries are up to **290,000 times** faster!

Hashing a value



A **hash** is an integer computed from a value. Python's `hash()` returns it.

```
print(hash("Hello world!"))
print(hash(True))
print(hash(3.1415926535))
print(hash(-1), hash(-2))
```

Output

```
4300268424190684184
1
32649043022899 2003
-2 -2
```

- Equal values always have equal hashes
- Different values can share a hash (**hash collision**)
- *A hash can be used to quickly find where a value is stored in a set or dictionary.* A list must instead check its items *one by one*.

{ Keys must be hashable

```
capacities = {"G.07": 120, "2.12": 40}
print(capacities["G.07"])

positions = {[0, 1]: "Lecture room"}
```

```
120
TypeError: unhashable type: 'list'
```

Set members and dictionary keys must be **hashable**.

Dictionary *values* can be anything.

Type	Mutable?	Hashable?
`str`, `int`, `float`, `bool`	No	Yes
`list`, `set`, `dict`	Yes	No

Meet the tuple

A **tuple** is like a list, but **immutable**. **Predict:** what does each line print?

```
location = (55.9444, -3.1872)
print(location[0])
print(location[-1])

empty = ()
one_item = (5,)
not_a_tuple = (5)
print(type(empty), type(one_item), type(not_a_tuple))
```

```
55.9444
-3.1872
<class 'tuple'> <class 'tuple'> <class 'int'>
```

- Parentheses, items separated by commas
- Index and loop as with lists
- One item needs a comma: `(5,)`

Watch out: no changes after creation: `location[0] = 56` raises `TypeError`.

Unpacking

Several names, one per item, in order. **Predict:** which value goes into `credits`?

```
course = ("IP", "Introduction to Programming", 20)
code, title, credits = course
print(f"{code}: {title} ({credits} credits)")
```

```
IP: Introduction to Programming (20 credits)
```

Watch out: one name per item: `code, title = course` raises `ValueError`.

Unlike lists, tuples *can* be hashable

```
room_at_position = {
    (0, 0): "Entrance",
    (0, 1): "Lecture room"
}

print(room_at_position[(0, 1)])
```

```
Lecture room
```

```
room_at_position = {
    [0, 0]: "Entrance",
    [0, 1]: "Lecture room"
}
```

```
TypeError: unhashable type: 'list'
```

Rule: a tuple is hashable **only if every item it contains is hashable**.

{ Try it yourself #1: a fixed record

- Make a tuple `event` containing an event name, a room, and a capacity
- Unpack it into three names and print a sentence
- Extra: show that this tuple is hashable by using it as key in a dictionary

About 3 minutes · on your own

Example solution #1

```
event = ("Programming lecture", "F.21", 150)
name, room, capacity = event
print(f"{name} is in {room} with capacity {capacity}")

bookings = {event: "confirmed"}
print(bookings[event])
```

```
Programming lecture is in F.21 with capacity 150
confirmed
```

The tuple is hashable because every item is a string or an integer.

Numbering with enumerate()

`enumerate(names)` supplies tuples: `(0, 'tea'), (1, 'soup'), (2, 'cake')`

Predict: what does each loop print?

```
names = ["tea", "soup", "cake"]

for index in range(len(names)):
    print(index, names[index])

for index, name in enumerate(names):
    print(index, name)
```

Output

```
0 tea
1 soup
2 cake
0 tea
1 soup
2 cake
```

Pairing with zip()

`zip(names, prices)` supplies tuples: `('tea', 2.4), ('soup', 3.5), ...`

Predict: what does each loop print?

```
names = ["tea", "soup", "cake"]
prices = [2.4, 3.5, 2.8]

for name, price in zip(names, prices):
    print(f"{name}: £{price:.2f}")

for line, record in enumerate(zip(names, prices), start=1):
    name, price = record
    print(f"{line}. {name}: £{price:.2f}")
```

Output

```
tea: £2.40
soup: £3.50
cake: £2.80
1. tea: £2.40
2. soup: £3.50
3. cake: £2.80
```

Watch out:
`zip()` stops at the **shortest** list.

Two-dimensional lists

`grid[row][column]`: first pick the row, then the item in it. **Predict:** what is printed?

```
temperatures = [
    [12, 14, 13],
    [11, 15, 16],
    [10, 12, 14],
]
print(temperatures[0])
print(temperatures[1][2])

temperatures[2][0] = 9
print(temperatures)
```

```
[12, 14, 13]
16
[[12, 14, 13], [11, 15, 16], [9, 12, 14]]
```

A nested loop visits each row, then each value in it:

```
for daily_readings in temperatures:
    for reading in daily_readings:
        print(reading)
```

Taking a slice

`sequence[start:stop:step]`: **stop is excluded**, as in `range()`.

```
values = [10, 20, 30, 40, 50, 60]
print(values[1:4])
print(values[:3])
print(values[3:])
print(values[2:5:2])
print(values[::3])
print(values[::-1])
```

```
[20, 30, 40]
[10, 20, 30]
[40, 50, 60]
[30, 50]
[10, 40]
[60, 50, 40, 30, 20, 10]
```

```
text = "programming"
print(text[:7])
print(text[:100])
```

```
program
programming
```

- A slice is a **new** sequence!
- *In Python*, no error if `stop` goes past the end

From text to a list, and back

Predict: which spaces survive the split?

```
text = "  tea   soup cake  "
words = text.split()
print(words)

print(" | ".join(words))
```

```
['tea', 'soup', 'cake']
tea | soup | cake
```

- `text.split()`: splits on whitespace
- `text.split(",")`: splits at each comma
- `separator.join(strings)`: the reverse

Live: split `"  pear,apple,pear,banana  "` at its commas, strip each word, remove duplicates with a set, and join the words with `", "`.

{ Comprehensions: a loop in one line

Loop

```
squares = []
for number in range(1, 6):
    squares.append(number ** 2)
print(squares)
```

→ `[1, 4, 9, 16, 25]`

List comprehension

```
squares = [number ** 2
           for number in range(1, 6)]
print(squares)
```

→ `[1, 4, 9, 16, 25]`

Add an `if` clause to keep only some items:

```
even_squares = [number ** 2 for number in range(1, 11)
                if number % 2 == 0]
print(even_squares)
```

→ `[4, 16, 36, 64, 100]`

{ Set and dictionary comprehensions

Curly brackets: a **set**, or a **dictionary** with `key: value`.

```
lengths = {word: len(word) for word in ["list", "tuple", "set"]}
remainders = {number % 3 for number in range(10)}

print(lengths)
print(remainders)
```

```
{'list': 4, 'tuple': 5, 'set': 3}
{0, 1, 2}
```

A set keeps one copy of each remainder: `0`, `1` and `2`.

Note: a comprehension should stay short and simple. If it is hard to read, write the loop instead.

Try it yourself #2: three comprehensions

With `words = ["python", "", "loops", "data"]`, build:

- the cubes of the integers from 1 to 5
- the non-empty words, in upper case
- a dictionary from each non-empty word to its first character

Why must the third one skip empty strings?

About 3 minutes · on your own

Example solution #2

```
words = ["python", "", "loops", "data"]

cubes = [number ** 3 for number in range(1, 6)]
uppercase_words = [word.upper() for word in words if word != ""]
first_characters = {word: word[0] for word in words if word != ""}

print(cubes)
print(uppercase_words)
print(first_characters)
```

```
[1, 8, 27, 64, 125]
['PYTHON', 'LOOPS', 'DATA']
{'python': 'p', 'loops': 'l', 'data': 'd'}
```

Without the filter, `""[0]` raises `IndexError`: an empty string has no first character.

> Summary

- Set members and dictionary keys must be **hashable**
- Tuples: immutable sequences, hashable when all their items are
- Unpacking: `first, second = pair`
- `grid[row][column]` for two-dimensional lists
- `sequence[start:stop:step]`: stop is excluded
- `enumerate()` and `zip()` supply tuples; `split()` and `join()` convert text and lists
- `[expression for item in items if condition]`, with `{...}` for sets and dictionaries

Homework: the consolidation challenge at the end of the notebook.