Sets, Logic, Computation: Python as a Playground

Reference: *Sets, Logic, Computation: An Open Introduction to Metalogic* (Richard Zach, Open Logic Project)

---

1. Overview of Python Collections

Python provides multiple collection types, including lists, tuples, dictionaries, and sets. Each of these serves a different purpose and has unique properties.

Type	Ordered	Duplicates	Mutable
list	✓	✓	✓
tuple	✓	✓	×
set	×	×	✓
dict	✓ (3.7+)	keys: ×	✓

Basic Collections

List A list is the most basic collection data type in Python. It is a **mutable**, **ordered** sequence, meaning that elements can be added, removed, or modified after the list is created.

```
mylist = [1, 2, 3]
```

Tuples A tuple is similar to a list but has a key distinction: it is **immutable**, meaning that once created, its elements cannot be changed. Tuples are useful when you need a fixed collection of elements that should not be altered.

```
mytuple = (1, 2, 3)
```

Sets A set is an **unordered** collection of unique elements. Unlike lists and tuples, sets do not allow duplicate values and do not maintain a specific order.

```
myset = {1, 2, 3}
```

Python also offers frozensets. A frozenset is similar to a set but is immutable.

Dictionaries A dictionary is a collection of key-value pairs. Each key is unique, and it maps to a specific value.

```
mydict = {'alice': 1, 'bob': 2, 'carol': 3}
```

Built-in Functions

Python provides built-in functions for collections. Two fundamental functions among them are `type()` and `len()`. The `type()` function returns the type of the specific object. The `len()` function returns the number of items in an object.

```
mylist = [1, 2, 3]

type(mylist)   # <class 'list'>
len(mylist)    # 3

myset = {1, 5, 7, 9, 10}
type(myset)    # <class 'set'>
len(myset)     # 5
```

---

2. Sets

2-1. What is a Set? (Extensionality)

A set is a **collection of objects**, considered as a single object. The objects making up the set are called **elements** or **members** of the set.

- If x is an element of set A: $x \in A$
- If x is not an element of set A: $x \notin A$
- A set with no elements is called the **empty set**, denoted $\emptyset$.

  Definition (Extensionality): If A and B are sets, then $A = B$ if and only if every element of A is also an element of B, and vice versa.

The key point of extensionality is that the identity of a set is determined **solely by its elements**. The order in which elements are listed, or how many times they appear, is irrelevant.

$$\{a, a, b\} = \{a, b\} = \{b, a\}$$

To prove that two sets are equal: show that whenever $x \in A$ then $x \in B$, and whenever $y \in B$ then $y \in A$.

2-2. Set Notation and Python

There are two ways to specify a set, each corresponding to a Python construct.

- **Extensional notation (roster notation):** List elements explicitly, e.g., $\{1, 2, 3\} \rightarrow$ corresponds to a Python set literal `{1, 2, 3}`
- **Intensional notation (set-builder notation):** Define by a property, e.g., $\{x : \varphi(x)\}$, read as "the set of all x such that $\varphi(x)$" $\rightarrow$ corresponds to a Python **set comprehension** `{x for x in ... if ...}`

Example: $\{x : x \leq 100 \wedge x \text{ is even}\} = \{2, 4, 6, \ldots, 100\}$

```
# Extensional notation
a = {1, 2, 3}
a = set([1, 2, 3])

# Intensional notation
{x for x in range(1, 101) if x % 2 == 0}    # $\rightarrow$ {2, 4, 6, ..
```

Note that while mathematical set-builder notation does not require specifying a universe, Python comprehensions always require an iterable to loop over (`range(1, 101)`). In Python, it is not possible to iterate over "all x" without a prior collection to draw from.

2-3. Extensionality and Python Sets

Now let's delve into the details of sets in Python. A set is an unordered collection of unique elements, meaning it does not allow duplicate values.

Create a Set To create a set in Python, you can use either curly braces `{}` or the built-in `set()` constructor.

```
a = {1, 2, 3}
a = set([1, 2, 3])
```

By extensionality, duplicate elements are automatically removed.

```
{1, 1, 2, 2, 2, 3, 3, 3}      # $\rightarrow$ {1, 2, 3}
{'a', 'b', 'a', 'a', 'c', 'b'} # $\rightarrow$ {'a', 'b', 'c'}
```

Create an Empty Set To create an empty set in Python, you must use the `set()` function. Using empty curly braces `{}` creates an empty dictionary, not an empty set.

```python
empty_set = set()

type(empty_set)    # <class 'set'>
len(empty_set)     # 0

e = {}
type(e)            # <class 'dict'>
```

Unordered In Python, sets are unordered collections, meaning their elements do not have a defined or predictable order. Sets do not maintain the order of elements as they are inserted.

```python
{1, 3, 2}   # output order may vary each time
```

---

3. Set Operations

3-1. Membership — $x \in A$

The mathematical expression $x \in A$ corresponds to membership testing in Python using the `in` operator. It checks whether the element x is part of the collection A.

```python
a = {1, 2, 3}
1 in a   # True
4 in a   # False
```

3-2. Size of a Set

To get the size of a set in Python, you can use the built-in `len()` function.

```python
a = {1, 2, 3}
len(a)   # 3
```

3-3. Subsets and Supersets

> **Definition (Subset):** If every element of a set A is also an element of B, then A is a **subset** of B, written $A \subseteq B$. If $A \subseteq B$ but $A \neq B$, we write $A \subsetneq B$ and say A is a **proper subset** of B.

Combining extensionality with the definition of subset gives us the following proposition.

Proposition: $A = B$ if and only if both $A \subseteq B$ and $B \subseteq A$.

Subset $A \subseteq B$ The mathematical expression $A \subseteq B$ corresponds to subset testing in Python. You can test if one set is a subset of another using the `issubset()` method or the `<=` operator. It checks whether all elements of A are also elements of B.

```
a = {1, 2}
b = {1, 2}
c = {1, 2, 3, 4}

a.issubset(b)   # True
a <= b          # True
```

```
a.issuperset(b)   # True
a >= b            # True
a.issuperset(c)   # False
```

Superset $A \supseteq B$

Proper Subset $A \subsetneq B$ The mathematical expression $A \subset B$ corresponds to checking in Python whether all elements of A are in B and the sets are not equal. This can be done using the `issubset()` method combined with a comparison or the `<` operator.

```
a < b                          # False (since a == b)
a.issubset(b) and a != b       # False
a < c                          # True
```

3-4. Union, Intersection, and Difference

```
a = {1, 2, 3}
b = {2, 3, 4, 5}
```

Definition (Union): $A \cup B$ is the set of all things which are elements of A, B, or both.

$$A \cup B = \{x : x \in A \vee x \in B\}$$

The mathematical expression $A \cup B$ is represented in Python using either the `union()` method or the `|` operator. This operation combines all unique elements from both sets.

> **Definition (Intersection):** $A \cap B$ is the set of all things which are elements of **both** A and B.
>
> $$A \cap B = \{x : x \in A \land x \in B\}$$
>
> Two sets are called **disjoint** if their intersection is empty.

The mathematical expression $A \cap B$ is represented in Python using either the `intersection()` method or the `&` operator. This operation returns a set containing elements that are common to both sets.

> **Definition (Difference):** $A \setminus B$ is the set of all elements of A which are not also elements of B.
>
> $$A \setminus B = \{x : x \in A \land x \notin B\}$$

The mathematical expression $A \setminus B$ or $A - B$ is represented in Python using either the `difference()` method or the `-` operator. This operation returns a set containing elements that are in A but not in B.

Operation	Notation	Python operator	Python method	Result
Union	$A \cup B$	`a | b`	`a.union(b)`	`{1, 2, 3, 4, 5}`
Intersection	$A \cap B$	`a & b`	`a.intersection(b)`	`{2, 3}`
Difference	$A \setminus B$	`a - b`	`a.difference(b)`	`{1}`

---

4. Power Sets

> **Definition (Power Set):** The set consisting of all subsets of a set A is called the **power set** of A, written $\wp(A)$.
>
> $$\wp(A) = \{B : B \subseteq A\}$$

Example: $\wp(\{a, b, c\}) = \{\emptyset,\ \{a\},\ \{b\},\ \{c\},\ \{a, b\},\ \{a, c\},\ \{b, c\},\ \{a, b, c\}\}$

A set with n elements has a power set with 2^n elements.

There is no built-in function in Python to generate a power set directly, but you can create one using the following custom implementation.

```python
def power_set(input_set):
    input_set = set(input_set)
    result = [set()]
    for item in input_set:
        new_subsets = [subset | {item} for subset in result]
        result.extend(new_subsets)
    return result

# Example usage
a = {1, 2, 3}
power_set(a)
# [set(), {1}, {2}, {1, 2}, {3}, {1, 3}, {2, 3}, {1, 2, 3}]
```

> **Tip:** In Python, it is impossible for a set to have other sets as elements. The most straightforward way to represent a power set is to use a list of sets.

---

5. Tuples and Ordered Pairs

5-1. Ordered Pairs

It follows from extensionality that sets have no order to their elements. When order matters, we use **ordered pairs** $\langle x, y \rangle$.

The key property of ordered pairs:

$$\langle a, b \rangle = \langle c, d \rangle \iff a = c \text{ and } b = d$$

Therefore, if $x \neq y$ then $\langle x, y \rangle \neq \langle y, x \rangle$. This contrasts with sets, where $\{x, y\} = \{y, x\}$.

> **Note (Wiener–Kuratowski definition):** In pure set theory, ordered pairs are defined as: $\langle a, b \rangle = \{\{a\}, \{a, b\}\}$

Ordered pairs and tuples in Python are closely related concepts used to represent collections of items in a specific order. An ordered pair is a set of two elements where the order matters. They are defined using parentheses.

```python
p = (1, 2)
```

Tuples are ordered sequences that can contain any number of elements, including ordered pairs. In Python, there is no specific "ordered pair" data type. Instead, Python uses tuples to represent ordered collections of elements, including pairs.

```
t = (1, 2, 3, 4)
```

5-2. Comparison

Two tuples are equal when they have the same length and all corresponding elements are equal.

```
(1, 2) == (1, 2)   # True
(1, 2) == (2, 1)   # False  ← order matters!
```

5-3. Cartesian Product — $A \times B$

> **Definition (Cartesian Product):** Given sets A and B, their Cartesian product $A \times B$ is defined by
>
> $$A \times B = \{\langle x, y\rangle : x \in A \text{ and } y \in B\}$$
>
> **Proposition:** If A has n elements and B has m elements, then $A \times B$ has $n \cdot m$ elements.

The product $A \times A$ is written A^2. More generally, the set of n-tuples from A is written A^n.

In Python, the Cartesian product of multiple sets or lists can be easily computed using the `itertools.product()` function from the `itertools` module.

```
from itertools import product

a = {'a', 'b'}
b = {1, 2}
list(product(a, b))
# [('a', 1), ('a', 2), ('b', 1), ('b', 2)]
```

For example, we can create a Cartesian product of nouns and verbs. Each tuple in the result represents a combination of a noun and a verb. This Cartesian product gives us all possible pairings of nouns and verbs.

```
import itertools
```

```
nouns = {'alice', 'boole', 'cantor'}
verbs = {'runs', 'jumps', 'sleeps'}

list(itertools.product(nouns, verbs))
# generates all 9 (noun, verb) combinations
```

---

6. Russell's Paradox

The set-builder notation $\{x : \varphi(x)\}$ is powerful, but **not every property defines a set**. The most famous example of this is Russell's Paradox.

Theorem (Russell's Paradox): There is no set $R = \{x : x \notin x\}$.

Proof: Suppose R exists. We can ask whether $R \in R$ or not.

- If $R \in R$, then by definition of R, we must have $R \notin R$. — Contradiction!
- If $R \notin R$, then by definition of R, we must have $R \in R$. — Contradiction!

Therefore R cannot exist. □

This paradox shows that set theory cannot be based on the naive principle that any property defines a set. Modern mathematics avoids this by adopting axiomatic systems such as ZFC (Zermelo–Fraenkel set theory), which precisely regulates the conditions under which sets exist.

In Python, sets cannot contain other sets — this is simply not allowed.

```
s = set()
s.add(s)   # TypeError: unhashable type: 'set'
```

While not directly related, Russell himself proposed **Type Theory** as one solution to this paradox. By classifying objects into hierarchical types, it logically prevents a set from containing itself as an element. Together with ZFC, Type Theory has had a lasting influence on modern mathematics and programming language design.

---

7. Example: Logic and Sets

A World

Here is a world:

$$H = \{\text{alice}, \text{boole}, \text{cantor}\}, \quad M = \{\text{alice}, \text{boole}, \text{cantor}, \text{pooh}, \text{simba}\}$$

```
human  = {'alice', 'boole', 'cantor'}
mortal = {'alice', 'boole', 'cantor', 'pooh', 'simba'}
```

In this world, there are two sets of beings: humans and mortals. The set of humans includes 'alice', 'boole', and 'cantor', representing the individuals who are defined as human. However, the set of mortals is larger, encompassing not only the humans but also 'pooh' and 'simba', who are mortal but not human.

Verifying Propositions

Proposition	Mathematical expression	Python code	Result
Alice is human	alice $\in H$	`'alice' in human`	`True`
All humans are mortal	$H \subseteq M$	`human <= mortal`	`True`
Not all mortals are human	$M \not\subseteq H$	`mortal <= human`	`False`
Non-human mortals	$M \setminus H$	`mortal - human`	`{'pooh', 'simba'}`
Humans and mortals	$H \cap M$	`human & mortal`	`{'alice', 'boole', 'cantor'}`
Humans = mortal humans	$H = H \cap M$	`human == human & mortal`	`True`

Alice is a human To check if 'alice' is a human in our defined world, we can use the `in` operator to test for membership in the human set.

```
'alice' in human   # True
```

All humans are mortal Since every element in the `human` set is also in the `mortal` set ($H \subseteq M$), it is true that all humans are mortal.

```
human <= mortal   # True
```

Not all mortals are human The elements 'pooh' and 'simba' exist in the `mortal` set but not in the `human` set. This shows that while all humans are mortal, not all mortals are human.

```
mortal <= human    # False
```

Non-human mortals The difference between the `mortal` set and the `human` set gives us the non-human mortals:

```
mortal - human    # {'pooh', 'simba'}
```

Humans and Mortality The intersection of the two sets confirms that all humans are indeed mortal:

```
human & mortal    # {'alice', 'boole', 'cantor'}

human == human & mortal    # True
```

Exercises

Exercise 1

Given the following sets, verify each statement, express it using set notation, and confirm with Python code.

$$P = \{\text{alice}, \text{bob}, \text{carol}, \text{pooh}\}, \quad S = \{\text{alice}, \text{bob}\}$$

```
person  = {'alice', 'bob', 'carol', 'pooh'}
student = {'alice', 'bob'}
```

(1) pooh is a student.

(2) All students are persons.

(3) All persons are students.

(4) Find the persons who are not students.

(5) Find the number of persons who are not students.

Exercise 2

Given the following sets, verify each statement, express it using set notation, and confirm with Python code.

$$V = \{a, e, i, o, u\}, \quad F = \{a, e, i\}, \quad B = \{o, u\}$$

```
vowels       = {'a', 'e', 'i', 'o', 'u'}
front_vowels = {'a', 'e', 'i'}
back_vowels  = {'o', 'u'}
```

(1) All front vowels are vowels.

(2) Not all vowels are front vowels.

(3) There is no vowel that is both a front vowel and a back vowel.

(4) Find the vowels that are not front vowels.

(5) The union of front vowels and back vowels equals the set of all vowels.

Glossary

Term	Wikipedia
Set	Set
Element / Member	Element
Empty set	Empty set
Extensionality	Axiom of extensionality
Extensional notation	Set-builder notation
Intensional notation	Set-builder notation
Subset	Subset
Proper subset	Subset
Superset	Subset
Power set	Power set
Union	Union
Intersection	Intersection
Difference	Complement
Disjoint sets	Disjoint sets
Ordered pair	Ordered pair
n-tuple	Tuple

Term	Wikipedia
Wiener–Kuratowski definition	Ordered pair
Cartesian product	Cartesian product
Russell's Paradox	Russell's paradox
ZFC (Zermelo–Fraenkel set theory)	ZFC
Type Theory	Type theory