

COMP1730/COMP6730

Programming for Scientists

Dictionaries and sets

Announcements (Project assignment)

- * **Project assignment [25%]**
 - 2D motion simulator of multi-planetary systems
 - Open, due on **15/10/2023, 11:55pm** (end of sem. week 10)
 - Read project specification carefully
 - Make sure you understand the requirements
 - Ask questions on the forum if unsure
 - Start early!
 - More on the assignment on tomorrow's lecture

Announcements (Accessing feedback in Wattle)

- ★ Assessment feedback can be found in Wattle's Gradebook, which can be accessed via the "Grades" link at the top of Wattle
- ★ You will find the feedback in the feedback column (automarker feedback, marking keys and comments from tutors are all combined)

Grade item	Calculated weight	Grade	Range	Percentage	Rank	Feedback
COMP1730/COMP6730 - Programming for Scientists - Sem 2 2023						
Homeworks						
Homework 1	13.33 %	2.00	0-2	100.00 %	1/466	#1: call unstack_tower_of_two ok; #2: call unstack_tower_of_three ok; Expr ignored at <code>unstack_tower_of_three</code> , line 3; wrong file name <code>unstack_tower_of_three</code> ; ID:2.0, TF:2.0; no comment
Homework 2	13.33 %	0.00	0-2	0.00 %	466/466	all stages failed (1:test #0: call neural_network(0, 0, 3.2, -0.1, 0.9, 0.7, -3.6, -1.0, -1.0, -1, -2.2, 0) returned incorrect answer 0.0; the expected answer is 0.5; 2:test #0: call neural_network(0, 0, 1, 1, 1, 1, 1, 0, 0, 0) returned incorrect answer 0.0; the expected answer is 0.5; 3:test #0: call neural_network(-1e+308, 0, -1e+308, 1e+308, 1e-308, -1e+308, -1e+308, 0, 1e-308, -1e+308, 1e+308, 1e+308) returned incorrect answer inf; the expected answer is 1.0); wrong file name <code>neural_network</code> ; nan; ID:2.0, FN:1.0, TF:0.0, ES:1.0; no comment
Homework 3	20.00 %	0.00	0-3	0.00 %	466/466	all stages passed; failed 0 out of 7 stages; wrong file name <code>neural_network</code> ; ID:2.0, FN:1.0, ES:1.0, VN:2.0, CC:0.0, DS:1.0, FD:2.0, CO:2.0, DRY:2.0
Homework 4	26.67 %	-	0-4	-	-	
Homework 5	26.67 %	-	0-4	-	-	



Lecture outline

- * Dictionaries: the Python `dict` type
- * Sets: the Python `set` type

Dictionaries (the concept of mapping)

- * A dictionary is **conceptually** a mapping
- * In general, a mapping can be defined as a relation between inputs and outputs that associates a **unique** (i.e., one and only one) output to each input
- * (Many) examples of mappings:
 - A continuous mathematical function $y = f(x)$
(mapping among x and y values)
 - A look-up index (e.g., mapping between words and page numbers in a book index, between names and phone numbers in a contact list, etc.)
 - ...
 - Even a Python list can be interpreted as a mapping between consecutive integers (list indices) and list elements

Dictionaries (definition)

- ★ A dictionary is a **mapping** where the correspondence among inputs and outputs is **explicitly** stored (in an efficient way)
- ★ Inputs are called keys and outputs are called values
- ★ Each key is associated to one and only one value (recall that dictionaries are mappings!)
- ★ If a given key is associated to a value, we refer to these two as a key–value pair, and say that dictionaries store key-value pairs
- ★ Keys can be values of any **immutable type** (e.g., integers, strings, tuples, etc.), while values can be of any type (i.e., mutable or immutable)
- ★ Dictionaries are also known as associative arrays, hash tables, symbol tables or simply maps in other programming languages

Common operations with dictionaries

- ★ What can you do with a dictionaries?
 - Create new, empty dictionary
 - Store a value with a key
 - Is a given key stored in the dictionary?
 - Look up the value stored for a given key
 - Remove key
 - Enumerate keys, values, or key–value pairs
- ★ A (common) misconception is to think that dictionaries are implemented using, e.g., two lists of the same length (one for the keys and another one for the values), or a single list of tuples
- ★ This is **NOT** the case: a hallmark of dictionaries is that key look-up takes (amortised) constant time (as, e.g., index look-up in lists)

Python's dict type

Two example ways of creating dictionaries:

1. Using dictionary literals:

```
adict = {}  
adict = dict()  
adict = { (12,2015) : 33.4, (6,2016) : 148.3 }  
adict = {"Australia":"Canberra", "Spain":"Madrid" }
```

- Dictionary (and set!) literals use curly brackets (i.e., { and })
- The literal can contain `key:value` pairs, which become the initial contents

2. Using a **dictionary comprehension** (example):

```
fin = open("data.tsv", "r")  
adict = {line.split("\t")[0]:line.split("\t")[1] for line in fin}  
fin.close()
```

Question: Can you tell what this latter code is doing?

Dictionary look-up read operations

- * Is a given key `key` stored in the dictionary `adict`? (True/False)

```
>>> key in adict
```

- * Dictionary look-up read (get value associated to given key):

```
>>> word_counter = { "be" : 2, "can" : 1 }
>>> word_counter["can"]
1
```

Same syntax as sequence indexing (but with any key type!)

- * Looking-up for nonexisting key produces a `KeyError` runtime error

```
>>> word_counter["the"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'the'
```

- * We can use `if key in adict:` as a guard against the error

Dictionary look-up write operations

- * Dictionary look-up write comes in two flavours
 - Add a new key-value pair
 - Update value associated to an existing key

```
>>> word_counter["the"] = 5    # Adds a new key--value pair
>>> word_counter
{'be': 2, 'can': 1, 'the': 5}
>>> word_counter["can"] += 1  # Updates value of existing key
>>> word_counter
{'be': 2, 'can': 2, 'the': 5}
```

- * **IMPORTANT:** As opposed to lists, no need to use, e.g., `list.append(element)`, to add new element. Indexing + assignment does the job (see example)

More on dictionaries

- * `dict` is a mutable type (as, e.g., lists, and NumPy arrays); see example in previous slide

- * Keys **MUST BE** *immutable* (*)

```
>>> alist = [1,0]
```

```
>>> adict = { alist : 2 }
```

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

- * A dictionary can contain keys of different (immutable) types
- * Stored values can be of any type (mutable or immutable)

More mutating dictionary operations

- * Removing keys:
 - `del adict[key]`
Removes `key` from `adict`
 - `adict.pop(key)`
Removes `key` from `adict` and returns the associated value
 - `adict.popitem()`
Removes an arbitrary `(key, value)` pair and returns it
- * `del` and `pop` cause a runtime error if `key` is not in dictionary;
`popitem` if it is empty

Iteration over dictionaries

- * `adict.keys()`, `adict.values()`, and `adict.items()` return *views* of the keys, values and key–value pairs
- * Views are iterable, but *not* sequences

```
for item in adict.items():  
    the_key = item[0]  
    the_value = item[1]  
    print(the_key, ':', the_value)
```

Programming examples

Codes will be uploaded after lecture

1. Counting frequency of items (i.e., to build a histogram):
 - Words in a file (or web page)
 - (Combinations of) values in a data table
2. Using dictionaries to represent and evaluate high degree polynomials with few monomials

Sets

- * A *set* is an **unordered** collection of (immutable) values without duplicates
- * Similar to dictionary but only keys (no values)
- * What can you do with a set?
 - Create a new set (empty or from an iterable)
 - Add or remove values
 - Is a given element in the set? (membership)
 - Mathematical operators: union, intersection, difference (note: not complement!)
 - Enumerate values

Python's set type

- * Set literals are written with `{ . . }`, but with elements only, not key–value pairs:

```
>>> aset = { 1, 'c', (2.5, 'b') }
```

- * `{ }` creates an empty dictionary, not a set!
- * A set can be created from any iterable:

```
>>> aset = set("AGATGATT")
```

```
>>> aset
```

```
{'T', 'A', 'G'}
```

- No duplicate elements in the set
- No order of elements in the set



Set operators

`elem in aset`

membership ($e \in A$)

`aset.issubset(bset)`

subset ($A \subseteq B$)

`aset | bset`

union ($A \cup B$)

`aset & bset`

intersection ($A \cap B$)

`aset - bset`

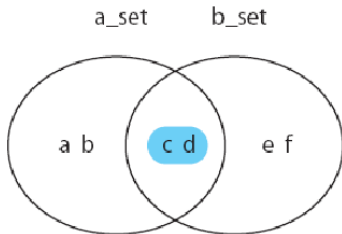
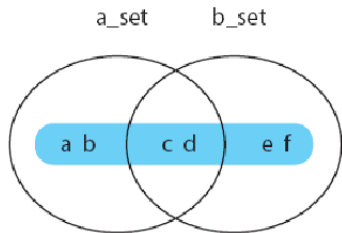
difference ($A \setminus B, A - B$)

`aset ^ bset`

symmetric difference

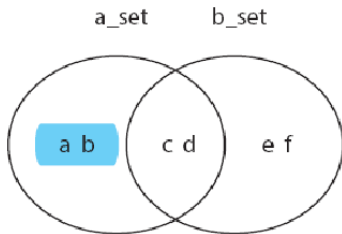
- ★ Set operators return a new result set, and do not modify the operands.
- ★ Also exist as methods (`aset.union(bset)`, `aset.intersection(bset)`, etc).

- * The union of `a_set` and `b_set` is the set of all elements that are in `a_set`, in `b_set`, or in both.
- * The intersection of `a_set` and `b_set` is the set of elements that are in both `a_set` and `b_set`.

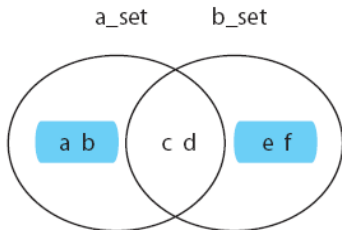


(Images from Punch & Enbody)

- ★ The difference of `a_set` and `b_set` is the set of elements in `a_set` that are not in `b_set`.

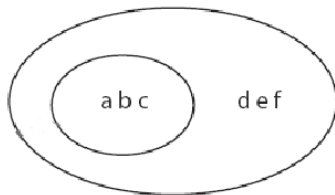


- ★ The symmetric difference of `a_set` and `b_set` is the set of elements that are in either but not in both.



(Images from Punch & Enbody)

- ★ a_set is a subset of b_set iff every element in a_set is also in b_set .
- ★ $A \subseteq B$ iff $A \cap B = A$.

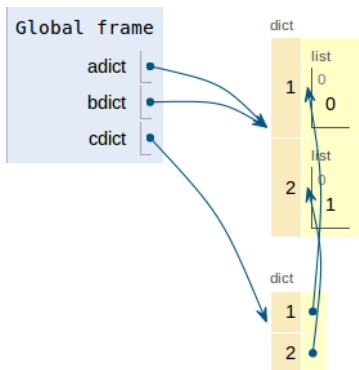


(Image from Punch & Enbody)

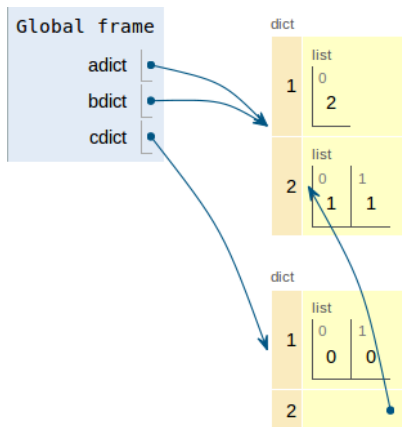
Copying

- * Dictionaries and sets are mutable objects
- * Like lists, dictionaries and sets store *references* to values
- * `dict.copy()` and `set.copy()` create a *shallow* copy of the dictionary or set
 - New dictionary / set, but containing references to the same values
 - Dictionary keys and set elements are immutable, so shared references do not matter
 - Values stored in a dictionary can be mutable

```
adict = {1:[0],2:[1]}  
bdict = adict  
cdict = adict.copy()  
bdict[1] = [2]  
cdict[1] = [0, 0]  
adict[2].append(1)
```



```
adict = {1:[0],2:[1]}
bdict = adict
cdict = adict.copy()
bdict[1] = [2]
cdict[1] = [0, 0]
adict[2].append(1)
```



Take-home messages

- * Dictionaries store mappings among keys (inputs) and values (outputs)
- * As opposed to sequences, allows for arbitrary/generalized indices
- * Implemented internally such that very fast lookup
- * Key-value pairs have no ordering (code that assumes an ordering is wrong!)
- * Set is different from dictionaries by having only keys (no values).