ECE326

PROGRAMMING LANGUAGES

Lecture 5 : Dictionary and File

Kuei (Jack) Sun
ECE
University of Toronto
Fall 2019

Dictionary

- As known as associative array
  - std::unordered_map in C++
- Collection of key value pairs
  - All keys must be unique
  - Unordered
    - You cannot sort a dictionary
- Implementation
  - Hash table
  - Search tree (e.g. red-black tree)

Dictionary

- { key1:value1, key2:value2, … }

```
>> eng2sp = {‘one’: ‘uno’, ‘two’: ‘dos’, ‘three’: ‘tres’}
>> eng2sp['one']
'uno'
>> eng2sp[4]                                # key not in dictionary
KeyError: 4

>> eng2sp['four'] = 'cuatro’                # add key-value pair
>> eng2sp
{'one': 'uno', 'two': 'dos', 'three': 'tres', 'four': 'cuatro'}

>> 'two' in eng2sp      # membership test on key
True
>> ‘dos’ in eng2sp      # cannot use to check if value exists
False
```

Key Requirement

- Dictionary key must be *hashable*
  - Related to *immutable*
  - Tuple is hashable if it has no reference to *mutable* objects

```
>> d = dict()      # creates an empty dictionary
>> d[1,2] = “hi”  # OK
>> a = [1, 2]
>> d[a] = “bye”   # not OK, list is mutable
TypeError: unhashable type: 'list‘
>> t = (1, a)
>> t
(1, [1, 2])
>> d[t] = “bye”   # not OK, tuple contains a mutable object
TypeError: unhashable type: 'list‘
```

Building Dictionary

- `zip` built-in function
  - Creates *n* *m*-tuples from *m* sequences of length *n*
    - Each element in tuple taken from same position of each sequence
  - *Lazy iterable*: computes as you loop through

```
>> a = zip(“abcd”, range(4), [1.5, 2.5, 3.5, 4.5])
>> a                    # zip is ‘lazy’ (so is range)
<zip object at 0x6ffffcc0888>
>> list(a)              # consume the iterable
[('a', 0, 1.5), ('b', 1, 2.5), ('c', 2, 3.5), ('d', 3, 4.5)]
```

- Build dictionary with list of 2-tuples (key-value pairs)

```
>> dict(zip(‘abcde’), range(5)))
{'a': 0, 'b': 1, 'c': 2, 'd': 3, 'e': 4}
```

Dictionary Methods

```
>> d = dict(zip(‘abcde’), range(5)))
>> d.get(‘f’, -1) # avoids KeyError by providing default value
-1

>> d.keys()
dict_keys(['a', 'b', 'c', 'd', 'e'])
>> d.values()
dict_values([0, 1, 2, 3, 4])

>> ‘’.join(c for c in d)      # loops through keys only
‘abcde’

>> for k, v in d.items():     # loops through (key, value)
..     print(k+str(v), end=' ')
a0 b1 c2 d3 e4
```

Dictionary Methods

```
>> d = dict(zip(‘abcdef’), range(6)))

>> del d[‘a’]              # remove key ‘a’ and its value
>> d
{'b': 1, 'c': 2, 'd': 3, 'e': 4, 'f': 6}

>> d.pop(‘c’)              # same as above and return its value
2
>> d
{'b': 1, 'd': 3, 'e': 4, 'f': 6}

>> d.update(dict(zip(‘abc’, range(6,9)))) # ‘b’ gets new value
>> d
{'b': 7, 'd': 3, 'e': 4, 'f': 6, 'a': 6, 'c': 8}
```

Dictionary Comprehension

- `{ K(x):V(x) for x in iterable }`
  - And the other two forms

Loop through unique characters only

Convention for "I don't care"

```
# make histogram of letters in string s
>> s = ‘mississauga’
>>  { c:s.count(c) for c in set(s) }
{'u': 1, 'g': 1, 'a': 2, 's': 4, 'i': 2, 'm': 1}

# keep only pairs where value > 3
>> import random
>> r = [ random.randint(0, 9) for __ in range(4) ]
>> d = dict(zip(“abcd”, r))
>> { k:v for k,v in d.items() if v > 3 }
{'b': 8, 'd': 5}
```

Function Arguments

- Keyword arguments
  - Specify an argument using parameter name
  - Useful for skipping over default arguments

```
def pizza(size=14, dough=“regular”, sauce=“tomato”, \
          cheese=“mozzarella”, toppings=[]):
    …

# dough is regular
mine = pizza(16, sauce=“alfredo”, cheese=“cheddar”)

# all arguments before toppings use default values
yours = pizza(toppings=[“bacon”, “pepperoni”, “salami”])
```

Variadic Function

- Allows you to take variable number of arguments
  - Both positional and/or keyword arguments

```
def foo(*args, **kwargs):
    print(args, kwargs)

>> foo(1, 2, bar=3, baz=4)
((1, 2), {'baz': 4, 'bar': 3})

# only accepts keyword arguments
def foo(**kwargs):
    print(kwargs)

>> foo(1, 2)
TypeError: foo() takes exactly 0 arguments (2 given)
```

Dynamic Programming

- Divide and conquer
  - Breaks down problem into sub-problems
  - Solve sub-problems and combine results to form solution
- Caches (saves) result of function for future reuse
  - Requires overlapping sub-problems
- Example: Fibonacci series
  - $F(n) = F(n-1) + F(n-2)$
    $= F(n-2) + F(n-3) + F(n-3) + F(n-4)$
    $= \ldots$
  - Lots of overlapping sub-problems!

Bottom-Up Approach

- Start from bottom-most unsolved sub-problem
- Solve its way up to the final solution

max() returns max value in the iterable

```
# bottom-up Fibonacci
def fibonacci(n):
    if n not in fibonacci.table:
        # start from smallest unsolved sub-problem
        mx = max(fibonacci.table) + 1
        for i in range(mx, n+1):     # ends at i == n
            fibonacci.table[i] = fibonacci.table[i-1] + \
                                 fibonacci.table[i-2]
    return fibonacci.table[n]

# static function variable!
fibonacci.table = { 0 : 1, 1 : 1 }   # f(0) = f(1) = 1
```

Memoization

- Same as dynamic programming, except top-down
- Advantage (over dynamic programming)
  - less computation if not all sub-problem needs to be solved
- Disadvantage
  - Recursion requires more memory than tabulation
- Example: Prolog (logic programming language)

```
?- fibonacci(100, F).
ERROR: Out of local stack
:- table fibonacci/2
?- fibonacci(100, F).
F = 573147844013817084101.
```

Top-Down Approach

- If sub-problem already solved, use result directly
- Else solve the sub-problem and add solution to table

```
# top-down Fibonacci
def fibonacci(n):                  # assume n >= 0
    if n in fibonacci.table:
        return fibonacci.table[n]
    else:
        v = fibonacci(n-1) + fibonacci(n-2)
        fibonacci.table[n] = v
        return v

fibonacci.table = { 0 : 1, 1 : 1 }
```

Pure Function

- Output solely determined by input to function
  - Also cannot have *side effects*
    - i.e. changing states outside of local environment
    - e.g. modifying non-local variables, perform I/O, etc.
  - Important in functional programming
- Referential transparency
  - Replacing expression by its corresponding value does not change program behaviour
  - Guaranteed from a pure function
- Requirement for memoisation/dynamic programming

Files

- Use open built-in function

```
>> f = open(“hello.txt”) # read-only mode, file must exist
>> h = open(“io.h”, “w”) # write-only mode, file will be wiped
```

- Reading text files

```
>> for line in f:          # file objects are iterable
..     print(line)
```

- Writing text files

```
>> h.write(“hello world\n”)   # add a new line to sentence
```

- Close file (after finished)

```
>> f.close()
```

Error Handling

- Deals with runtime error without crashing
- Need to disrupt normal execution flow
- Example: goto statement in C

```
    int i;
    Dir * d = malloc(sizeof(Dir)*NUM_DIRS);
    if (d == NULL) goto fail;
    for (i = 0; i < NUM_DIRS; i++) {
        if (!(d[i] = alloc_dir()))
            goto fail_d;
        /* do stuff with d[i] */
    }
return 0;
fail_d:
    for (i-- ; i >= 0; i--)
        free_dir(d[i]);
    free(d);
fail:
    return –ENOMEM;
```

Error Handling

- C++/Python: `try` statement
- Jumps to exception handler on error
  - May need to unwind stack frames (function calls)
  - Can be expensive (C++ compile option `--fno-exceptions`)

```
try:
    f = open(“hello.txt”)
except OSError as err:
    print(err)
else:                            # if no error occurs
    print(f.read())              # this reads everything
    f.close()
```

With Statement

- Some objects have pre-defined clean-up actions
  - Special `__exit__` method
- Makes code look much cleaner

```
# close called automatically when exiting block

with open(“hello.txt”) as f:
    print(f.read())

# Note: f still in scope here (but is closed)
```

User-Defined Exception

- Create a class derived from base Exception class
  - More on creating class in future lectures

```
class MyError(Exception):
    pass

>> raise MyError(“It’s bad”)       # raise your own exception
__main__.MyError: It's bad

pr = analyze_move(mv)
if pr > 1.0:                       # use built-in exception
    raise ValueError(“probability can’t be > 1!”)
```

Multiple Exceptions

```
def baz():
      try:
            foo()       # exceptions can be raised from inside
            bar()       # a function call for caller to handle
            …
      except (KeyError, ValueError):
            # deal with these two the same way
            return 0
      except OSError as err:
            print(err)
            return -1
      except:
            print(“unexpected exception!”)
            raise       # re-raise the exception to
      …                 # caller of this function
```