Chapter 02 - Lists and Dictionaries

May 8, 2021

0.1 Item 11: Know How to Slice Sequences

- be careful of slicling with negative numbers as you can end up with suprising results
- `somelist[-0:]` will result in `somelist[:]` and copy the orginal list
- result of slicing a `list` is a whole new `list`
- in assignments, slices replace the specified range in the orginal `list`
- you can shrink a list if you index a larger range and replace it with a shorter range
- simmilarly you can grow a list when indexing it
- if you assign to a slice with no start or end indexes, you replace the entire contents of the `list` with a copy of whats refrenced and not allocating a new `list`

0.2 Item 12: Avoid Striding and Slicing in a Single Expression

- in python, there is a special syntax to stride to form a list
- `somelist[start:end:strinde]`
- the syntax above lets you take every `nth` item when slicing a sequence

[1]:
```
x = ['red', 'orange', 'yellow', 'green', 'blue', 'purple']
odds = x[::2]
evens = x[1::2]

print(odds)
print(evens)
```

```
['red', 'yellow', 'blue']
['orange', 'green', 'purple']
```

- problem with stride is that it has unexpected behavior
- for example, if you are trying to reverse using it `x[::-1]`
- the syntax work for strings but as soon as the data is encoded as `UTF-8` it breakes

- in general avoid using a `stride` along with start and end indexes
- if you need to use it, perer making it a postive value and omit start and end indexes
- if you need to use stride, perfer making it a postive value anbd omit start and end indexes
- striding and then slicing creates an extra shallow copy of the data
- if your program cant afford the `time` or `memory` consider using `itertools` which is clearer to read and does not permit negative values for start, end, or stride

[2]:
```
y = x[::2] # ['a', 'c', 'e', 'g']
z = y[1:-1] # ['c', 'e']
```

0.3 Item 13: Perfer Catch-All Unpacking Over Slicing

- one limitation of unpacking is that you need to know the lenght of the sequence you are unpacking in advance
- the solution to the unpacking problem is below but it is noisy and error prone

[5]:
```
car_ages = [0, 9, 4, 8, 7, 20, 19, 1, 6, 15]
car_ages_descending = sorted(car_ages, reverse=True)

oldest = car_ages_descending[0]
second_oldest = car_ages_descending[1]
others = car_ages_descending[2:]

print(oldest, second_oldest, others)
```

```
20 19 [15, 9, 8, 7, 6, 4, 1, 0]
```

- to correct this, python has introduces catch-all unpacking through a `starred expression`
- a starred expression may appear in any position
- the catch is that you need to have atleast one required part or you get an error
- you also cant use multiple catch-all expressions in a single-level unpacking pattern

[8]:
```
oldest, second_oldest, *others = car_ages_descending
print(oldest, second_oldest, others)

*others, second_youngest, youngest = car_ages_descending
print(youngest, second_youngest, others)
```

```
20 19 [15, 9, 8, 7, 6, 4, 1, 0]
0 1 [20, 19, 15, 9, 8, 7, 6, 4]
```

- you can use multiple starred expressions in an unpacking assignment statement, as long as they're catch-alls for different parts of the multilevel structure being packed
- it is not reccomended

[9]:
```
car_inventory = {
 'Downtown': ('Silver Shadow', 'Pinto', 'DMC'),
 'Airport': ('Skyline', 'Viper', 'Gremlin', 'Nova'),
}
((loc1, (best1, *rest1)),
 (loc2, (best2, *rest2))) = car_inventory.items()
print(f'Best at {loc1} is {best1}, {len(rest1)} others')
print(f'Best at {loc2} is {best2}, {len(rest2)} others')
```

```
Best at Downtown is Silver Shadow, 2 others
Best at Airport is Skyline, 3 others
```

- starred expressions becomes `list` instances in all cases
- if there are no leftovers, then the leftover becomes an empty list

[10]:
```
short_list = [1, 2]
first, second, *rest = short_list
print(first, second, rest)
```

```
1 2 []
```

- you can also unpack arbitrary iterators with the unpacking syntax
- below is an example a generator that yields the rows of a CSV file containing all car orders from the dealership this week

[11]:
```
def generate_csv():
    yield ('Date', 'Make' , 'Model', 'Year', 'Price')
```

- processing the results of the generator using indexes and slices is ok but requires multiple lines and its visually noisy

[13]:
```
all_csv_rows = list(generate_csv())
header = all_csv_rows[0]
rows = all_csv_rows[1:]

print('CSV Header:', header)
print('Row count: ', len(rows))
```

```
CSV Header: ('Date', 'Make', 'Model', 'Year', 'Price')
Row count:  0
```

- unpacking with a starred expression makes it easy to process the first row- the header separately from the rest of the iterators contents

[14]:
```
it = generate_csv()
header, *rows = it

print('CSV Header:', header)
print('Row count: ', len(rows))
```

```
CSV Header: ('Date', 'Make', 'Model', 'Year', 'Price')
Row count:  0
```

- the problem with unpacking an iterator risks the potential of using up all of the memory on your computer and causing your program to crash

0.4 Item 14: Sort by Complex Criteria Using the `key` Parameter

- you cant sort classes because the `sort` method tries to call comparision special methods that are not define dby the class
- but there might still be some attribute on an object that you wold like to use for sorting
- that is where the `key` parameter can be used
- the `key` function should be a comparable value to use in place of an item for sorting purposes
- with the lambda function passed as the `key` parameter you can access attributes of items

[17]:
```
class Tool:
    def __init__(self, name, weight):
        self.name = name
        self.weight = weight

    def __repr__(self):
        return f'Tool({self.name!r}, {self.weight})'

tools = [
    Tool('level', 3.5),
    Tool('hammer', 1.25),
    Tool('screwdriver', 0.5),
    Tool('chisel', 0.25),
]

print('Unsorted:', repr(tools))
tools.sort(key=lambda x: x.weight)
print('\nSorted: ', tools)
```

```
Unsorted: [Tool('level', 3.5), Tool('hammer', 1.25), Tool('screwdriver', 0.5),
Tool('chisel', 0.25)]

Sorted:  [Tool('chisel', 0.25), Tool('screwdriver', 0.5), Tool('hammer', 1.25),
Tool('level', 3.5)]
```

- for strings you might also want to process them such as using lowercase before sorting

[18]:
```
places = ['home', 'work', 'New York', 'Paris']
places.sort()

print('Case sensitive: ', places)
places.sort(key=lambda x: x.lower())
print('Case insensitive:', places)
```

```
Case sensitive:  ['New York', 'Paris', 'home', 'work']
Case insensitive: ['home', 'New York', 'Paris', 'work']
```

- sometimes you need to sort by multiple criteria
- the simplest solution in Python is to use the `tuple` type
- Tuples are immutable sequences of arbitrary python values
- tuples are comparable by default and have a natural ordering, meaning they implement all of the special methods such a `__lt__` that are required by the `sort` method
- `Tuples` impement these special method comparators by iterating over each position in the `tuple` and comparing the corespondig values one index at a time
- in the first position in the tuples being compared are equal, then the tuple comparision will move on to the second position

[19]:
```
saw = (5, 'circular saw')
jackhammer = (40, 'jackhammer')
assert not (jackhammer < saw) # Matches expectations
```

[20]:
```
drill = (4, 'drill')
sander = (4, 'sander')
assert drill[0] == sander[0] # Same weight
assert drill[1] < sander[1] # Alphabetically less
assert drill < sander # Thus, drill comes first
```

- you can take advantage of this `tuple` comparision in order to sort the list by multiple parameters
- one limitation of having the key function return a `tuple` is that the direction of sorting for all critera must be the same, either all ascending or decending order
- providing `reversed=True` will effect all the tuple criteria

[24]:
```
power_tools = [
 Tool('drill', 4),
 Tool('circular saw', 5),
 Tool('jackhammer', 40),
 Tool('sander', 4),
]

power_tools.sort(key=lambda x: (x.weight, x.name))
print(power_tools)
print("")

power_tools.sort(key=lambda x: (x.weight, x.name),
 reverse=True) # Makes all criteria descending
print(power_tools)
```

```
[Tool('drill', 4), Tool('sander', 4), Tool('circular saw', 5),
Tool('jackhammer', 40)]

[Tool('jackhammer', 40), Tool('circular saw', 5), Tool('sander', 4),
Tool('drill', 4)]
```

- for numerical values its possible to mix sorting directions by using the `unary` minus operator in the `key` function
- this negates one of the values in the returned `tuple` effectively reversing its sort order while leaving the other in tact

[25]:
```
power_tools.sort(key=lambda x: (-x.weight, x.name))
print(power_tools)
```

```
[Tool('jackhammer', 40), Tool('circular saw', 5), Tool('drill', 4),
Tool('sander', 4)]
```

- if the `key` lambda is giving you an error, just combine multiple sorting

0.5 Item 15: Be Cautious When Relying on `dict` Insertion Ordering

- after `Python 3.6` dictionaries perserve insertion order
- a concenquence of this is that keyword arguments to functions – including the `**kwargs` catch-all parameter previously would come though an seemingly random order which made it hard to debug function calls

- classes also use the `dict` type for their instance dictionaries
- so you can assume that the order that the values are perserved

[26]:
```
class MyClass:
     def __init__(self):
        self.alligator = 'hatchling'
        self.elephant = 'calf'

a = MyClass()

for key, value in a.__dict__.items():
    print(f'{key} = {value}')
```

```
alligator = hatchling
elephant = calf
```

- there is still the `OrderedDict` class that perserves ordering
- if you need to handle a high rate of key insertions and popitem calls, `OrderdedDict` may be a better fit than the standard `dict`

- order is not always maintained
- python makes it easy for programmers to define their own custom container types that emulate the standard `protocols` matching `list`, `dict`, and the other types
- they inherit form `collections.abc`
- python is not statically typed, so modt code relies on `duck typing`, where an object's behavior is its de facto type, instead of rigid class hierarchies

- lets say you wanted a program to show the results of a contest for the cutest baby animal

[34]:
```
votes = {
 'otter': 1281,
 'polar bear': 587,
 'fox': 863,
}
```

- we can define a function to process this voting and save the rank of each animal name into a provided empty dictionary

[28]:
```
def populate_ranks(votes, ranks):
    names = list(votes.keys())
    names.sort(key=votes.get, reverse=True)

    for i, name in enumerate(names, 1):
        ranks[name] = i
```

- we also need a function that will tell me what animal won the contest
- this function works by assuming that `populate_ranks` will assign the contest of the `ranks` dictionary in ascending order, meaning that the first key must be the winner

[32]:
```
def get_winner(ranks):
    return next(iter(ranks))
```

- we can check that our program runs correctly

[33]:
```
ranks = {}
populate_ranks(votes, ranks)
print(ranks)
winner = get_winner(ranks)
print(winner)
```

```
{'otter': 1, 'fox': 2, 'polar bear': 3}
otter
```

- lets sat the requirements for the program have changed
- the `UI` element that shows the results should be in alphabetical order instead of `rank` order
- to accomplicsh this, I can use the `collections.abc` built-in module to define a new dictionary-like class that `iterates` its contents in alphabetical order

[45]:
```
from collections.abc import MutableMapping

class SortedDict(MutableMapping):

    def __init__(self):
        self.data = {}

    def __getitem__(self, key):
        return self.data[key]

    def __setitem__(self, key, value):
        self.data[key] = value

    def __setitem__(self, key, value):
        self.data[key] = value

    def __delitem__(self, key):
        del self.data[key]

    def __iter__(self):
        keys = list(self.data.keys())
        keys.sort()
        for key in keys:
            yield key

    def __len__(self):
```

```
        return len(self.data)
```

- we can use `SortedDict` instance in place of a standard dict with the functions form before and no erros will be raised since the class conforms to the protocal of a standard dictionary
- howerver the results are incorrect

[46]:
```
sorted_ranks = SortedDict()
populate_ranks(votes, sorted_ranks)
print(sorted_ranks.data)
winner = get_winner(sorted_ranks)
print(winner)
```

```
{'otter': 1, 'fox': 2, 'polar bear': 3}
fox
```

- the problem is that the implementation of `get_winner` assumes that the dictonary's iteration is in insertion order to match `populate_ranks`
- this code is using the `SortedDict` instead of `dict`, so the assumption is no longer true
- thus the value returned for the winner is `fox`, which is true alphabetically

- one solutions are to either reimplement the `get_winner` function to no longer assume that the ranks dictionary has a specific iteration order
- this is the most conserative and robust solution

[47]:
```
def get_winner(ranks):
    for name, rank in ranks.items():
        if rank == 1:
            return name

winner = get_winner(sorted_ranks)
print(winner)
```

```
otter
```

- another approach is to add an explicit check to the top of the function to ensure that the type of `ranks` matches my expectations and to raise an expection if not
- this solution likely has better runtime performance than the more conserative approach

[48]:
```
def get_winner(ranks):
    if not isinstance(ranks, dict):
        raise TypeError('must provide a dict instance')
    return next(iter(ranks))


try:
    get_winner(sorted_ranks)
except TypeError:
    print('ust provide a dict instance')
```

```
ust provide a dict instance
```

- the third alternative is to use type annotations to enforce that the value passed to `get_winner` is a dict instance and not a `MutableMapping` with dictionary-like behavior
- you can run the `mypy` tool in strict mode on an annotated version

[50]:
```
# would have thrown an error if mypy was implented

from typing import Dict, MutableMapping
def populate_ranks(votes: Dict[str, int],
                   ranks: Dict[str, int]) -> None:

    names = list(votes.keys())
    names.sort(key=votes.get, reverse=True)

    for i, name in enumerate(names, 1):
        ranks[name] = i

def get_winner(ranks: Dict[str, int]) -> str:
    return next(iter(ranks))


votes = {
 'otter': 1281,
 'polar bear': 587,
 'fox': 863,
}

sorted_ranks = SortedDict()
populate_ranks(votes, sorted_ranks)
print(sorted_ranks.data)
winner = get_winner(sorted_ranks)
print(winner)
```

```
{'otter': 1, 'fox': 2, 'polar bear': 3}
fox
```

- there are three ways to be careful about dictionary-like classes: write code that doesnt rely on insertion, ordering, explicitly check for the `dict type` at runtime

0.6 Perfer `get` over `in` and `KeyError` to Handle Missing Dictionary Keys

- the three `fundemental` operations for interacting with dictionaries are `accessing`, `assigning` and `deleting` keys and their associated values
- the contents of dictionaries are dynamic and thus its entirely possible, even likely that when you try to access or delte a key it wont be possible
- you could use `try/except` but that is not the best approach

[51]:
```
counters = {
 'pumpernickel': 2,
```

```
 'sourdough': 1,
}

key = 'pumpernickel'
count = counters.get(key, 0)
counters[key] = count + 1
```

- you should generally use `Counter` class if you have to count keys
- if dictionaries have a more complex type, such as a `list`, do the following

[53]:
```
votes = {
 'baguette': ['Bob', 'Alice'],
 'ciabatta': ['Coco', 'Deb'],
}

key = 'brioche'
who = 'Elmer'

if key in votes:
    names = votes[key]
else:
    votes[key] = names = []

names.append(who)
print(votes)
```

```
{'baguette': ['Bob', 'Alice'], 'ciabatta': ['Coco', 'Deb'], 'brioche':
['Elmer']}
```

[54]:
```
try:
    names = votes[key]
except KeyError:
    votes[key] = names = []
    names.append(who)
```

- you could also use the `get` method to fetch a `list` value when the key is present, or do one fetch and one assignment if the key isn't present

[56]:
```
names = votes.get(key)
if names is None:
    votes[key] = names = []

names.append(who)
```

- that above can be shortened to one line which improve readability

[58]:
```
if (names := votes.get(key)) is None:
    votes[key] = names = []
```

```
names.append(who)
```

- the `dict` type also provides the `setdefault` method to help shorten this pattern even further
- `setdefault` tries to fetch the value of akey in the dictionary
- if the key is not present, the method assigns that key to the default value provided
- and then the method returnd the value for that key: either the originally present value or the newly inserted default value
- `setdefault` is used below to implement the same logic as in the `get` example above

[62]:
```
# readibility is not obvious, so you might not want to go with approach

names = votes.setdefault(key, [])
names.append(who)
```

- another problem with the top approach is that the default value passed to `setdefault` is assigned directly into the dictionary when the key is missing instead of being cipied
- inother words, you get an empty array if nothing was provided to the `value` default parameter
- checking for this mistake leads to overhead which leads to worse performance

[61]:
```
data = {}
key = 'foo'
value = []
data.setdefault(key, value)
print('Before', data)
value.append('hello')
print('After: ', data)
```

```
Before {'foo': []}
After:  {'foo': ['hello']}
```

0.7 Item 17: Perfer `defaultdict` over `setdefault` to Handle Missing Items in Internal State

- when you work with a dictionary you did not create, you can handle missing key in multiple ways (i.e. item 16, perfer `get` over `keyerror`)
- there are cases howerver we need to use `setdefault`

- lets say I want to keep track of the cities I have visited in countries around the world

[63]:
```
visits = {
 'Mexico': {'Tulum', 'Puerto Vallarta'},
 'Japan': {'Hakone'},
}
```

- I can use the `setdefault` method to add new citites to the sets, whether the country name is already present in the dictionary or not
- this approach is shorter than using the `get` method

[64]:
```
visits.setdefault('France', set()).add('Arles') # Short
```

```
if (japan := visits.get('Japan')) is None: # Long
    visits['Japan'] = japan = set()

japan.add('Kyoto')

print(visits)
```

```
{'Mexico': {'Tulum', 'Puerto Vallarta'}, 'Japan': {'Hakone', 'Kyoto'}, 'France':
{'Arles'}}
```

- what about situations when you do control the creation of the dictionary being accessed
- this is generally the case when youre using a dictionary instance to keep track of the internal state of a class, for example
- below is the example above wrapped in a class weith helper methods to access the dynamic inner state stored in a dictionary

[66]:
```
class Visits:
    def __init__(self):
        self.data = {}

    def add(self, country, city):
        city_set = self.data.setdefault(country, set())
        city_set.add(city)
```

- the new class hides the complexities of calling `setdefault` correctly and it provides a nicer interface for the programmer

[67]:
```
visits = Visits()
visits.add('Russia', 'Yekaterinburg')
visits.add('Tanzania', 'Zanzibar')
print(visits.data)
```

```
{'Russia': {'Yekaterinburg'}, 'Tanzania': {'Zanzibar'}}
```

- the implementation of the `Visits.add` method is not idea
- the `setdefault` method is still confusingly named, which makes it more difficult for a new reader of the code to immediately understand whats happening
- the implementation is also not effecient because it constructs a new `set` instance on every call, regardless of whether the given country was already present in the data dictionary
- the `defaultdict` calss from `collections` simplifies the common use case by automatically storing a default value when a key doesent exits

[68]:
```
from collections import defaultdict

class Visits:
    def __init__(self):
        self.data = defaultdict(set)

    def add(self, country, city):
```

```
        self.data[country].add(city)

visits = Visits()
visits.add('England', 'Bath')
visits.add('England', 'London')
print(visits.data)
```

```
defaultdict(<class 'set'>, {'England': {'London', 'Bath'}})
```

- note we avoid having to call the `setdefault` method which could be costly considering how many times we call it
- If you're creating a dictionary to manage an arbitrary set of potential keys, then you should prefer using a defaultdict instance from the collections built-in module if it suits your problem.
- If a dictionary of arbitrary keys is passed to you, and you don't control its creation, then you should prefer the get method to access its items. However, it's worth considering using the setdefault method for the few situations in which it leads to shorter code

0.8 Item 18: Know How To Construct Key-Dependend Default Values with 'missing

- there are times when neither `setdefault` not `defaultdict` is the right fit
- imagine I want to write a program to manage social network profile pictures on the filesystem
- I need a dictionary to map the profile picture pathnames to open file handles so I can read and write those images as needed
- below, this is done by using a normal `dict` instance and checking for the presence of keys, using the `get` method and an assignment expression

[71]:
```
pictures = {}
path = 'profile_1234.png'
if (handle := pictures.get(path)) is None:
    try:
        handle = open(path, 'a+b')
    except OSError:
        print(f'Failed to open path {path}')
        raise
    else:
        pictures[path] = handle
handle.seek(0)
image_data = handle.read()
```

- when the file exists in the dictionary, this code make only a signle dictionary access
- if the file handle does not exist, the dictionary is accessed once by `get` and then it is assigned in the `else` clase
- the call to the read method stands clearly seprate from the code that calls `open` and handles exceptions

- we could use `KeyError` but that would require more dictionary accesses and levels of nesting
- we could also use `setdefault` method

[72]:
```
try:
    handle = pictures.setdefault(path, open(path, 'a+b'))
except OSError:
    print(f'Failed to open path {path}')
    raise
else:
    handle.seek(0)
    image_data = handle.read()
```

- the problems with the code above is that the `open` built-in function to create the file handle is always called, even when the path is already present in the dictionary
- this results in additional file handle that may confilict with the existing open handles in the same programs
- exceptions may be raised by the open call and need to be handeled but it may not be possible to differentate tem from exceptions that may be raised by the `setdefault` call on the same line

- if your trying to manage internal state, another assumption you might make is that `defaultdict` could be used for keeping track of these profile pictures

[77]:
```
from collections import defaultdict

def open_picture(profile_path):
    try:
        return open(profile_path, 'a+b')
    except OSError:
        print(f'Failed to open path {profile_path}')
        raise

pictures = defaultdict(open_picture)
try:
    handle = pictures[path]
    handle.seek(0)
    image_data = handle.read()
except TypeError:
    print('open_picture() missing 1 required positional argument')
```

```
open_picture() missing 1 required positional argument
```

- the problem is that `defaultdcit` expects that the function passed to its constructor doesn't require any arguments
- this means that the helper function that defaultdict calls does not know which specific key is being accessed, which eliminates my ability to call `open`

- this problem is common enough that there is a built-in solutin
- you cann subclass the `dict` type and implement the `__missing__` special method to dadd custom logic for handling missing keys
- there we define a new class that takes advantage of the same `open_picture` hemper method defined above

[78]:
```
class Pictures(dict):
    def __missing__(self, key):
        value = open_picture(key)
        self[key] = value
        return value

pictures = Pictures()
handle = pictures[path]
handle.seek(0)
image_data = handle.read()
```

- when the pictures[path] dictionary access finds that the path key isent present in the dictionary, the `__missing__` method is called
- this method must create the new default value for the key, insert it into the dictionary and return it to the caller
- subsequent access of the same path will not call `__missing__` since the corresponding items are already present

- The setdefault method of dict is a bad fit when creating the default value has high computational cost or may raise exceptions.
- the function passed to `defaultdict` must have the default value depend on the key being accesesd
- you can define your own `dict` subclass with a `__missing__` method in order to construct default values that must know which key was being accessed