Dictionaries

- Rather than accessing values by numbers (e.g., shopping_list[2]), it is also possible to access values by strings, e.g.

```
login = {
    'service': 'Camosun',
    'user_ID': 'me@camosun.ca',
    'password': 'myCamosunPassword'
}
```

  - ***login['service']*** returns *Camosun*
  - ***login['password']*** returns *myCamosunPassword*

Dictionaries

- Dictionaries can also be nested, e.g.

```
logins = {
    'Camosun': {
        'user_ID': 'me@camosun.ca',
        'password': 'myCamosunPassword'
    },
    'GitHub': {
        'user_ID': 'me@camosun.ca',
        'password': 'myGitHubPassword'
    }
}
```

  - ***logins['Camosun']*** returns a dictionary

  - ***logins['Camosun']['password']*** returns *myCamosunPassword*

Dictionaries

- To be on the safe side, we should either test if a key is in the dictionary, or provide a default value if the key is missing
  - ***if 'service' in login:***
    print(login['service'])
  - ***print(login.get('somekey', 'keynotfound'))***

Other Dictionary Commands

- Assuming *dct* is a dictionary:
  - ***dct.keys()*** -- a list containing all the keys of the dictionary
  - ***dct.values()*** -- a list containing all the values of the dictionary
  - ***dct.items()*** -- a list containing all the items of the dictionary in tuple form (more on that later)

Exercises

- Write a function that accepts a string and then tallies how often each character appears. For example, *Hello World* results in

```
H 1
e 1
l 3
o 2
  1
W 1
r 1
d 1
```

Tuples

- Python supports tuples, allowing one constant or variable to hold multiple values e.g.,
- ***street = ("123", "Main Street", "West")***
- ***(number, name, direction) = street***
- *number* will be *123*, *name* will be *Main Street*, *direction* will be *West*
- Can also use bracket notation, e.g. *street[1]* is *Main Street*

Tuples and Dictionaries

- When iterating through dictionaries, tuples provide an easy way to access both key and value pairs at the same time. Instead of:

```
for ch in t:
    print(ch, t[ch])
```

- we can use:

```
for k, v in t.items():
    print(k, v)
```

Exercises

- Write a function that accepts a string and then prints out the character that appears most often

reduce

- The ***map*** function applies an operation to every element in a list in isolation

- In some cases, we want to carry over the result of a previous operation to the next element (e.g., when summing all elements)

- For that, we can use the ***reduce*** function (from *functools*)

```
from functools import reduce
lst = [1, 2, 3, 4]
print(reduce(lambda x, y: x+y, lst))  # prints 10

dct = {'A': 1, 'B': 2, 'C': 3, 'D': 4}
print(reduce(lambda x, y: x + y, dct.values()))  # prints 10
```

- ***values()*** creates a list of all dictionary values

Ternary Operator

- Just like in C and Java, Python provides a ternary operator:
  _ _ _ _ ***if*** _____ ***else*** . . . .
- If ____ is true, then _ _ _ _ is evaluated, else . . . . is evaluated, e.g.,
- ***x if x > y else y***
- This is equivalent to ***max(x, y)***

Using *reduce* to Find the Maximum

- For lists and dictionaries:

```
lst = [1, 2, 3, 4]
print(reduce(lambda x, y: x if x > y else y, lst)) # prints 4

dct = {'A': 1, 'B': 2, 'C': 3, 'D': 4}
print(reduce(lambda x, y: x if x > y else y, dct.values()))  # prints
4
```

- For dictionaries, if we want to find the corresponding key:

```
dct = {'A': 1, 'B': 2, 'C': 3, 'D': 4}
print(reduce(lambda x, y: x if dct[x] > dct[y] else y, dct.keys()))
# prints D
```

sum and *max*

- Summing all values or finding the maximum is so common that dedicated functions exist:

```
sum(lst)  # 10
sum(dct.values())  # 10

max(lst)  # 4
max(dct.values())  # 4

print(max(dct, key=lambda x: dct[x]))  # D
```

Exercises

- Write a function that accepts a string and then prints out the character that appears most often; use the ***max*** function for this
- Write a function that accepts a string and then prints out all the characters that appear more than once (e.g., *lo* in case of *Hello World*); use the ***reduce*** function for this. Note that ***reduce(lambda x, y: operation, dictionary, initial_value))*** can be used to inject an initial value.

String Formatting

- There are many ways to format strings in Python. Here are some approaches:

```
name = input('What is your name? ')
print('Hello', name)  # , only works in a print
print('Hello ' + name)
print(f'Hello {name}')
print('Hello {}'.format(name))
print('Hello %s' % name)  # old
```

String Formatting

- Note that in case of variables that are not strings, some approaches must/can be modified:

```
value = 1/3
print('Value:', value)  # , only works in a print
print('Value: ' + str(value))
print(f'Value: {value}')
print('Value: {:0.2}'.format(value))  # the  :0.2 is optional; prints
0.33
```

Classes

- Python also supports classes:

```
class Treasure:
def __init__(self):
    self.value = 10

    def __str__(self):
        return f'${self.value}'


t = Treasure()
print(t.value)  # prints 10
print(t)  # prints $10
```

Classes

```
class Tile:
    def __init__(self, name: str, treasure: Treasure = None):
        self.name = name
        self.treasure = treasure

    def __str__(self):
        return f'{self.name}({self.treasure})'

    def remove_treasure(self):
        self.treasure = None


s = Tile("start")
print(s)      # prints start(None)

g = Tile("goal", Treasure())  # Treasure from previous slide
print(g)      # prints goal($10)
```

Classes

```
class Player:
    def __init__(self, name):
        self.name = name
        self.score = 0

    def visit(self, tile: Tile):  # visit a tile to pick up the treasure
        self.score += tile.treasure.value
        tile.remove_treasure()

    def __str__(self):
        return f'{self.name}: {self.score}'


p1 = Player('1')
print(p1)  # prints Player 1: 0
p1.visit(g)  # g from previous slide
print(g)  # prints goal(None) since treasure was picked up
print(p1)  # prints Player 1: 10
```

Instances

- Class instances are passed by reference:

```
class a_class:
    def __init__(self):
        self.a_variable = 'a value'


def a_function(a_parameter: a_class):
    a_parameter.a_variable = 'another value'


example = a_class()
print(example.a_variable)  # prints a value
a_function(example)
print(example.a_variable)  # prints another value
```

Data Attributes

- Can add data attributes (i.e., instance variables) on the fly:

```
p1.wins = True
print(p1.wins)  # prints True
```

Class vs. Instance Variables

```
class Treasure:
    name = 'Jewel'  # class variable

    def __init__(self):
        self.value = 10  # instance variable

    def update(self, name, value):
        Treasure.name = name
        self.value = value


t1 = Treasure()
t2 = Treasure()
print(t1.name, t1.value)  # prints Jewel 10
print(t2.name, t2.value)  # prints Jewel 10

t1.update('Gold', 100)
print(t1.name, t1.value)  # prints Gold 100
print(t2.name, t2.value)  # prints Gold 10 since name is a class var
```

Class vs. Instance Variables

- Avoid the following mistake:

```
class TreasureChest:
    treasures = []


t1 = TreasureChest()
t2 = TreasureChest()
t1.treasures.append('Gold')
print(t2.treasures)  # prints ['Gold']; but we don't want that
```

Class vs. Instance Variables

- Instead, use:

```
class TreasureChest:
    def __init__(self):
        self.treasures = []


t1 = TreasureChest()
t2 = TreasureChest()
t1.treasures.append('Gold')
print(t2.treasures)  # prints []
```

Class vs. Instance Variables

- Also watch out for this behaviour:

```
class Treasure:
    value = 10

t1 = Treasure()
t1.value = t1.value + 10  # instance variable!
print(t1.value)  # prints(20)
print(Treasure.value)  # prints(10)
Treasure.value += 1
print(Treasure.value)  # prints(11)
print(t1.value)  # prints(20)
```

- When an instance variable has not yet been defined, it takes on the value of the same-named class variable (if available)

Inheritance

- Inheritance is also supported:

```
class DesertTile(Tile):
    def __init__(self, name: str, treasure: Treasure = None):
        self.damage = 5
        super().__init__(name, treasure)
```

Modules

- Various ways exist to import variables, functions, and classes from other Python files

- ***import*** **_____**, e.g.,

import random
random.randrange(10)

- ***from*** **_____** ***import*** ..., e.g.,

from random import randrange
randrange(10)

- The latter form is recommended

- Avoid ***from*** **_____** ***import ****; this can lead to unexpected name clashes

Modules

- Code following ***if __name__ == "__main__":*** is not run when imported; only when running the file directly
- A directory can be turned into a package of different modules by including a file named *__init__.py*

Exercises

- On your own:
  - Work on the questions in the *Python* section of *Practice Questions and Solutions*

Lab 2

- Using classes in different modules (files), create a basic treasure hunting application:
  - 2D board
  - randomly-placed treasures
  - multiplayer
  - players can move 1 tile at a time; no turn order

Lab 3

- Using *pytest*, write code that tests your game code
- Create a GitHub Action that will run those tests every time code is committed to the main branch

Key Skills

- Write programs in Python using
  - various data types and operators,
  - selection statements (if, match)
  - repetition-related statements (while, for in, break, else, continue)
  - exception handling
  - file I/O
  - functions
  - lists
  - dictionaries
  - tuples
  - lambdas and maps, comprehensions, reduce, sum, max
  - classes and modules