

1. Recap
○○○○○○○
○○○○○○○

2. Error/Exception Handling
○○○○

3. File Input/Output
○○○○○

KOLT Python

Error Handling, File Input & Output

Ahmet Uysal

Monday 11th November, 2019

KOLT



1. Recap
○○○○○○○
○○○○○○○

2. Error/Exception Handling
○○○○

3. File Input/Output
○○○○○

Agenda

Python Data Model

Data Structures

1. Error/Exception Handling

2. File Input/Output



Mutability

Immutable:

An `object` with a fixed value.



Mutability

Immutable:

An **object** with a fixed value. Immutable objects include **numbers**, **strings** and **tuples**. Such an object cannot be altered.

Mutability

Immutable:

An **object** with a fixed value. Immutable objects include **numbers**, **strings** and **tuples**. Such an object cannot be altered. A new object has to be created if a different value has to be stored.

Mutability

Immutable:

An **object** with a fixed value. Immutable objects include **numbers**, **strings** and **tuples**. Such an object cannot be altered. A new object has to be created if a different value has to be stored.

```
a = 5  
a = 10  
a += 3
```

```
hello = 'hello'  
hallo = hello[0] + 'a' + hello[2:]
```

1. Recap

○○●○○○○○
○○○○○○○

2. Error/Exception Handling

○○○○

3. File Input/Output

○○○○○○

Python Data Model



Python Data Model

How did we represent data in Python?

Python Data Model

How did we represent data in Python? **Variables!**

Python Data Model

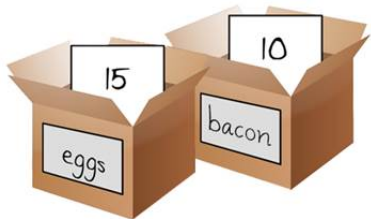
How did we represent data in Python? **Variables!**
How do they work?

Python Data Model

How did we represent data in Python? **Variables!**
How do they work? Do they store the data themselves?

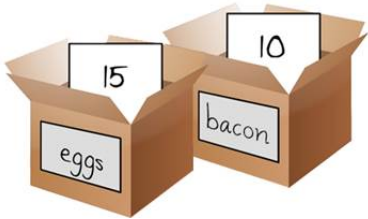
Python Data Model

How did we represent data in Python? **Variables!**
How do they work? Do they store the data themselves?



Python Data Model

How did we represent data in Python? **Variables!**
How do they work? Do they store the data themselves?



1. Recap

○○●○○○
○○○○○○

2. Error/Exception Handling

○○○○

3. File Input/Output

○○○○○○

Box Analogy



Box Analogy

```
my_age = 9  
my_age += 12  
print(my_age)      # => 21
```

Box Analogy

```
my_age = 9  
my_age += 12  
print(my_age)      # => 21
```

```
my_secret_box = [0, 1, 2]  
other_box = my_secret_box  
other_box.remove(2)  
print(my_secret_box)
```


Box Analogy

```
my_age = 9  
my_age += 12  
print(my_age)      # => 21
```

```
my_secret_box = [0, 1, 2]  
other_box = my_secret_box  
other_box.remove(2)  
print(my_secret_box)
```

Prints [0, 1]

Box Analogy

```
my_age = 9  
my_age += 12  
print(my_age)      # => 21
```

```
my_secret_box = [0, 1, 2]  
other_box = my_secret_box  
other_box.remove(2)  
print(my_secret_box)
```

Prints [0, 1]

Did we just changed inside of a closed box?

Box Analogy

```
my_age = 9  
my_age += 12  
print(my_age)      # => 21
```

```
my_secret_box = [0, 1, 2]  
other_box = my_secret_box  
other_box.remove(2)  
print(my_secret_box)
```

Prints [0, 1]

Did we just changed inside of a closed box?

Box analogy **does not** work!

Python Data Model

```
my_secret_box = [0, 1, 2]
```

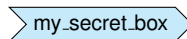
Python Data Model

```
my_secret_box = [0, 1, 2]
```

0	1	2
---	---	---

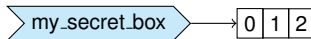
Python Data Model

```
my_secret_box = [0, 1, 2]
```



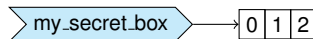
Python Data Model

```
my_secret_box = [0, 1, 2]
```



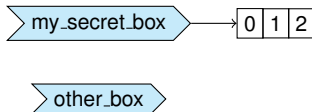
Python Data Model

```
my_secret_box = [0, 1, 2]  
other_box = my_secret_box
```



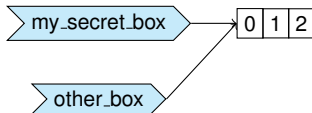
Python Data Model

```
my_secret_box = [0, 1, 2]  
other_box = my_secret_box
```



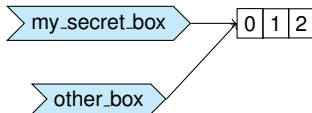
Python Data Model

```
my_secret_box = [0, 1, 2]  
other_box = my_secret_box
```



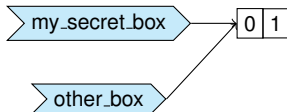
Python Data Model

```
my_secret_box = [0, 1, 2]  
other_box = my_secret_box  
other_box.remove(2)
```



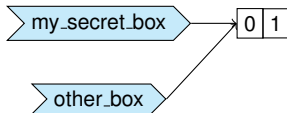
Python Data Model

```
my_secret_box = [0, 1, 2]  
other_box = my_secret_box  
other_box.remove(2)
```



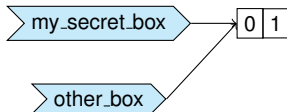
Python Data Model

```
my_secret_box = [0, 1, 2]
other_box = my_secret_box
other_box.remove(2)
print(my_secret_box)
```



Python Data Model

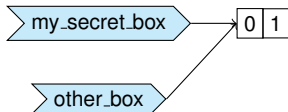
```
my_secret_box = [0, 1, 2]
other_box = my_secret_box
other_box.remove(2)
print(my_secret_box)
```



Variables are more like **labels** pointing to **values**!

Python Data Model

```
my_secret_box = [0, 1, 2]
other_box = my_secret_box
other_box.remove(2)
print(my_secret_box)
```



Variables are more like **labels** pointing to **values**!
Assignment links **variables** to **values**!

Object

Everything is an object in Python. Even though variables **do not** have `types`, each object has a **fixed** `type`.

↪ Values at the right side of our label analogy are objects!

Object

Everything is an object in Python. Even though variables **do not** have `types`, each object has a **fixed** `type`.

↪ Values at the right side of our label analogy are objects!

```
a = 5
```

Object

Everything is an object in Python. Even though variables **do not** have `types`, each object has a **fixed** `type`.

↪ Values at the right side of our label analogy are objects!

`a = 5`

5

Object

Everything is an object in Python. Even though variables **do not** have `types`, each object has a **fixed** `type`.

↪ Values at the right side of our label analogy are objects!

`a = 5`



Object

Everything is an object in Python. Even though variables **do not** have `types`, each object has a **fixed** `type`.

↪ Values at the right side of our label analogy are objects!

`a = 5`



Object

Everything is an object in Python. Even though variables **do not** have `types`, each object has a **fixed** `type`.

↪ Values at the right side of our label analogy are objects!

```
a = 5  
a = 10
```

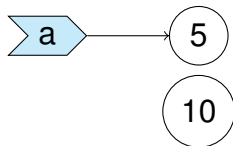


Object

Everything is an object in Python. Even though variables **do not** have `types`, each object has a **fixed** `type`.

↪ Values at the right side of our label analogy are objects!

```
a = 5  
a = 10
```

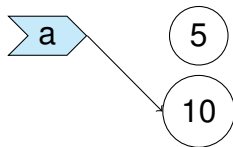


Object

Everything is an object in Python. Even though variables **do not** have `types`, each object has a **fixed** `type`.

↪ Values at the right side of our label analogy are objects!

```
a = 5  
a = 10
```

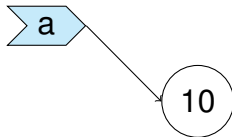


Object

Everything is an object in Python. Even though variables **do not** have `types`, each object has a **fixed** `type`.

↪ Values at the right side of our label analogy are objects!

```
a = 5  
a = 10
```

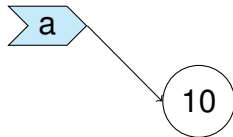


Object

Everything is an object in Python. Even though variables **do not** have `types`, each object has a **fixed** `type`.

↪ Values at the right side of our label analogy are objects!

```
a = 5  
a = 10  
a += 3
```

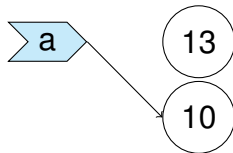


Object

Everything is an object in Python. Even though variables **do not** have `types`, each object has a **fixed** `type`.

↪ Values at the right side of our label analogy are objects!

```
a = 5  
a = 10  
a += 3
```

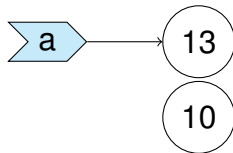


Object

Everything is an object in Python. Even though variables **do not** have `types`, each object has a **fixed** `type`.

↪ Values at the right side of our label analogy are objects!

```
a = 5  
a = 10  
a += 3
```

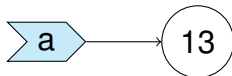


Object

Everything is an object in Python. Even though variables **do not** have `types`, each object has a **fixed** `type`.

↪ Values at the right side of our label analogy are objects!

```
a = 5  
a = 10  
a += 3
```

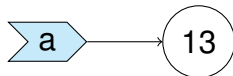


Object

Everything is an object in Python. Even though variables **do not** have `types`, each object has a **fixed** `type`.

↪ Values at the right side of our label analogy are objects!

```
a = 5  
a = 10  
a += 3  
print(a)
```



Object

Each object has an `identity`, this value can be obtained by using `id()` function.

`==` operator compares values, `is` operator compares identities.

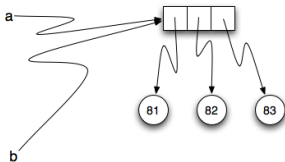
```
a = 1000
b = 1000
a == b    # => True
a is b    # => False
```

Almost always use `==` to compare values!

Aliasing & Cloning

- More than one variables can refer to **same object**!
- What if we want to clone/copy instead of aliasing?
- For lists, `list.copy()` \Rightarrow returns a shallow copy of the list.
- Shallow: only copy the references, not inner values.

• `>>> import copy`
`copy.copy(x)` : shallow copy, `copy.deepcopy(x)` : deepcopy



Tuples

- **Immutable** sequence(ordered) of elements.

Tuples

- **Immutable** sequence(ordered) of elements.
- Similar to `lists`, you can use **indexing**, **slicing**, and iterate over using `for` loops.

Tuples

- **Immutable** sequence(ordered) of elements.
- Similar to `lists`, you can use **indexing**, **slicing**, and iterate over using `for` loops.
- Elements cannot be added/removed/changed once the tuple is created.

Tuples

- **Immutable** sequence(ordered) of elements.
- Similar to `lists`, you can use **indexing**, **slicing**, and iterate over using `for` loops.
- Elements cannot be added/removed/changed once the tuple is created.
- How to create tuples?

Tuples

- **Immutable** sequence(ordered) of elements.
- Similar to `lists`, you can use **indexing**, **slicing**, and iterate over using `for` loops.
- Elements cannot be added/removed/changed once the tuple is created.
- How to create tuples? `my_tuple = (1, [1, 2], 'a')`

Tuples

- **Immutable** sequence(ordered) of elements.
- Similar to `lists`, you can use **indexing**, **slicing**, and iterate over using `for` loops.
- Elements cannot be added/removed/changed once the tuple is created.
- How to create tuples? `my_tuple = (1, [1, 2], 'a')`
- `len(my_tuple)` \Rightarrow

Tuples

- **Immutable** sequence(ordered) of elements.
- Similar to `lists`, you can use **indexing**, **slicing**, and iterate over using `for` loops.
- Elements cannot be added/removed/changed once the tuple is created.
- How to create tuples? `my_tuple = (1, [1, 2], 'a')`
- `len(my_tuple) ⇒ 3`
- `my_tuple.append(3) ⇒`

Tuples

- **Immutable** sequence(ordered) of elements.
- Similar to `lists`, you can use **indexing**, **slicing**, and iterate over using `for` loops.
- Elements cannot be added/removed/changed once the tuple is created.
- How to create tuples? `my_tuple = (1, [1, 2], 'a')`
- `len(my_tuple) ⇒ 3`
- `my_tuple.append(3) ⇒ AttributeError:`
`'tuple' object has no attribute 'append'`

Tuples

`() / tuple(): empty tuple,`

1. Recap

○○○○○○○
●●○○○○○

2. Error/Exception Handling

○○○○

3. File Input/Output

○○○○○○

Tuples

```
() / tuple(): empty tuple,  
(3):
```



Tuples

```
() / tuple(): empty tuple,  
(3): int 3,
```



Tuples

```
() / tuple(): empty tuple,  
(3): int 3,  
(3,):
```



Tuples

```
() / tuple(): empty tuple,  
(3): int 3,  
(3,): tuple containing 3
```



Tuples

() / tuple(): empty tuple,
(3): int 3,
(3,): tuple containing 3

```
my_list = [1, 2, 3]
my_tuple = ('a', my_list) # ('a', [1, 2, 3, 4])
my_list.append(4)
print(my_tuple)
my_list += [5, 6, 7] # my_list.extend(...)
print(my_tuple)
my_tuple += (1, 2) # my_tuple = my_tuple + (1, 2)
print(my_tuple)
```

Sets

- **Unordered** sequence of **unique** elements.

Sets

- **Unordered** sequence of **unique** elements.
- **Cannot** use **indexing/slicing**, can iterate with `for` loops.

Sets

- **Unordered** sequence of **unique** elements.
- **Cannot** use **indexing/slicing**, can iterate with `for` loops.
- **Mutable**, `add(element)`, `remove(element)` methods.

Sets

- **Unordered** sequence of **unique** elements.
- **Cannot** use **indexing/slicing**, can iterate with `for` loops.
- **Mutable**, `add(element)`, `remove(element)` methods.
- Python also has **immutable** sets: `frozenset`

Sets

- **Unordered** sequence of **unique** elements.
- **Cannot** use **indexing/slicing**, can iterate with `for` loops.
- **Mutable**, `add(element)`, `remove(element)` methods.
- Python also has **immutable** sets: `frozenset`
- How to create sets?

Sets

- **Unordered** sequence of **unique** elements.
- **Cannot** use **indexing/slicing**, can iterate with `for` loops.
- **Mutable**, `add(element)`, `remove(element)` methods.
- Python also has **immutable** sets: `frozenset`
- How to create sets? `my_set = {1, 2, 3, 4, 2}`
- How to create empty sets?

Sets

- **Unordered** sequence of **unique** elements.
- **Cannot** use **indexing/slicing**, can iterate with `for` loops.
- **Mutable**, `add(element)`, `remove(element)` methods.
- Python also has **immutable** sets: `frozenset`
- How to create sets? `my_set = {1, 2, 3, 4, 2}`
- How to create empty sets? `set()` (`{ }` is reserved for `dict`)

Sets

- **Unordered** sequence of **unique** elements.
- **Cannot** use **indexing/slicing**, can iterate with `for` loops.
- **Mutable**, `add(element)`, `remove(element)` methods.
- Python also has **immutable** sets: `frozenset`
- How to create sets? `my_set = {1, 2, 3, 4, 2}`
- How to create empty sets? `set()` (`{ }` is reserved for `dict`)
- Can compute set operations: **union**, **intersection**, **difference**, **symmetric difference**.

Sets

```
ceren = {'Marco', 'Irem', 'Sunduz'}
gul_sena = {'Gamze', 'Ata', 'Zeynep'}
hasan_can = {'Gamze', 'Berker', 'Cemre'}
ahmet = {'Irem', 'Demet', 'Ekin'}

# intersection &
print(gul_sena.intersection(hasan_can)) # => {'Gamze'}
print(ceren & gul_sena) # => set()
# union |
print(ceren.union(ahmet)) # => {'Ekin', 'Irem', 'Demet',
                                # 'Marco', 'Sunduz'}
print(hasan_can | ceren | gul_sena | ahmet) # => all names
# difference -
print((gul_sena - hasan_can)) # => {'Zeynep', 'Ata'}
# symmetric_difference ^
print(ceren.symmetric_difference(ahmet))
# => {'Marco', 'Ekin', 'Sunduz', 'Demet'}}
```

Dictionaries

- Collection of **key–value** pairs.



Dictionaries

- Collection of **key–value** pairs.
- Cannot use **indexing/slicing**, can iterate with `for` loops.

Dictionaries

- Collection of **key–value** pairs.
- **Cannot** use **indexing/slicing**, can iterate with `for` loops.
- In general, they are not **ordered**.

Dictionaries

- Collection of **key–value** pairs.
- **Cannot** use **indexing/slicing**, can iterate with `for` loops.
- In general, they are not **ordered**.
- However, in Python 3.7 pairs are guaranteed to be in insertion order.

Dictionaries

- Collection of **key–value** pairs.
- **Cannot** use **indexing/slicing**, can iterate with `for` loops.
- In general, they are not **ordered**.
- However, in Python 3.7 pairs are guaranteed to be in insertion order.
- In other words, we will get pairs in insertion order if we loop over the `dict`.

Dictionaries

- Collection of **key–value** pairs.
- **Cannot** use **indexing/slicing**, can iterate with `for` loops.
- In general, they are not **ordered**.
- However, in Python 3.7 pairs are guaranteed to be in insertion order.
- In other words, we will get pairs in insertion order if we loop over the `dict`.
- How to create dictionaries?

Dictionaries

- Collection of **key–value** pairs.
- **Cannot** use **indexing/slicing**, **can** iterate with `for` loops.
- In general, they are not **ordered**.
- However, in Python 3.7 pairs are guaranteed to be in insertion order.
- In other words, we will get pairs in insertion order if we loop over the `dict`.
- How to create dictionaries? `{ }/dict()`: empty dictionary

Dictionaries

- Collection of **key–value** pairs.
- **Cannot** use **indexing/slicing**, can iterate with `for` loops.
- In general, they are not **ordered**.
- However, in Python 3.7 pairs are guaranteed to be in insertion order.
- In other words, we will get pairs in insertion order if we loop over the `dict`.
- How to create dictionaries? `{ }/dict()`: empty dictionary
- `d = {'one': 1, 'two': 2, 'three': 3, 'four': 4}`

Dictionaries

- Collection of **key–value** pairs.
- **Cannot** use **indexing/slicing**, can iterate with `for` loops.
- In general, they are not **ordered**.
- However, in Python 3.7 pairs are guaranteed to be in insertion order.
- In other words, we will get pairs in insertion order if we loop over the `dict`.
- How to create dictionaries? `{ }/dict()`: empty dictionary
- `d = {'one': 1, 'two': 2, 'three': 3, 'four': 4}`
- How to access values?

Dictionaries

- Collection of **key–value** pairs.
- **Cannot** use **indexing/slicing**, can iterate with `for` loops.
- In general, they are not **ordered**.
- However, in Python 3.7 pairs are guaranteed to be in insertion order.
- In other words, we will get pairs in insertion order if we loop over the `dict`.
- How to create dictionaries? `{ }/dict()`: empty dictionary
- `d = {'one': 1, 'two': 2, 'three': 3, 'four': 4}`
- How to access values? `print(d['one'])` # \Rightarrow 1

Confused Section Leader Gul Sena

```
# I need a way to keep track of my students
my_students = {'Ayse': ['economics', 'freshman'],
               'Emir': ['psychology', 'master'],
               'Emirhan': ['business administration', 'junior'],
               'Furkan': ['law', 'junior'],
               'Mahsa': ['material science', 'phd'],
               'Meva': ['international relations', 'freshman']}

for student, info in my_students.items():
    print(f'{student} studies {info[0]}')
# Emir left my class :(
my_students.pop('Emir')
# someone new in my class
my_students['Canan'] = ['industrial engineering', 'junior']
# Ayse passed another year
my_students['Ayse'][1] = 'sophomore'
```

1. Recap

○○○○○○○
○○○○○○●

2. Error/Exception Handling

○○○○

3. File Input/Output

○○○○○○

Attendance



Attendance

Fill out the attendance form:

tiny.cc/kolt-python

Attendance

Fill out the attendance form:

tiny.cc/kolt-python

Password: **G2WIN**

Syntax Errors

What happens when you run a syntactically incorrect file?

Syntax Errors

What happens when you run a syntactically incorrect file?

```
for i in range(100)
print(i)
# SyntaxError: invalid syntax
```

Syntax Errors

What happens when you run a syntactically incorrect file?

```
for i in range(100)
print(i)
# SyntaxError: invalid syntax
```

```
while True:
print('Hello')
# IndentationError: expected an indented block
```

Easy to detect: Your code will not work :)

Runtime Exceptions

When a statement is **syntactically correct** does that mean we are safe?

Runtime Exceptions

When a statement is **syntactically correct** does that mean we are safe?

```
print(3/0)
```

Runtime Exceptions

When a statement is **syntactically correct** does that mean we are safe?

```
print(3/0) , int('hello')
```

Runtime Exceptions

When a statement is **syntactically correct** does that mean we are safe?

```
print(3/0) , int('hello') , 'hello'[2] = 'a'
```

Runtime Exceptions

When a statement is **syntactically correct** does that mean we are safe?

```
print(3/0) , int('hello') , 'hello'[2] = 'a'
```

How to be safe in these situations?

Runtime Exceptions

When a statement is **syntactically correct** does that mean we are safe?

```
print(3/0) , int('hello') , 'hello'[2] = 'a'
```

How to be safe in these situations?

- Put `if` checks everywhere?

Runtime Exceptions

When a statement is **syntactically correct** does that mean we are safe?

```
print(3/0) , int('hello') , 'hello'[2] = 'a'
```

How to be safe in these situations?

- Put `if` checks everywhere?
- Too much effort, and probably we cannot list every condition.

Runtime Exceptions

When a statement is **syntactically correct** does that mean we are safe?

```
print(3/0), int('hello'), 'hello'[2] = 'a'
```

How to be safe in these situations?

- Put `if` checks everywhere?
- Too much effort, and probably we cannot list every condition.
- Solution is `try-except-finally` blocks.

Try Except Blocks

```
try:
    <risky-statements>
    <risky-statements>
    <risky-statements>
    ...
except ValueError as valError:
    print('value error', valError)
except (RuntimeError, TypeError, NameError):
    print('One of the above errors, but not ValueError')
else:
    print('No errors')
finally:
    print('This always runs')
```


Try Except Blocks

```
def divide(x, y):  
    try:  
        result = x / y  
    except ZeroDivisionError:  
        print("division by zero!")  
    else:  
        print("result is", result)  
    finally:  
        print("executing finally clause")
```

Working With Files

Why might we want to work with files?

Working With Files

Why might we want to work with files?

- Work on **structured** data in large quantities.

Working With Files

Why might we want to work with files?

- Work on **structured** data in large quantities.
- Save the current state of the program for later retrieval

Working With Files

Why might we want to work with files?

- Work on **structured** data in large quantities.
- Save the current state of the program for later retrieval
 - How to add save/load functionality to game you have written?
- Save the result of your program.

Working With Files

Why might we want to work with files?

- Work on **structured** data in large quantities.
- Save the current state of the program for later retrieval
 - How to add save/load functionality to game you have written?
- Save the result of your program.
 - Save experiment results to a file.

Working With Files

Why might we want to work with files?

- Work on **structured** data in large quantities.
- Save the current state of the program for later retrieval
 - How to add save/load functionality to game you have written?
- Save the result of your program.
 - Save experiment results to a file.
- Keep logs for large systems.

Working With Files

Why might we want to work with files?

- Work on **structured** data in large quantities.
- Save the current state of the program for later retrieval
 - How to add save/load functionality to game you have written?
- Save the result of your program.
 - Save experiment results to a file.
- Keep logs for large systems.
- ...

Files In Python

Access to a `file` object using `open(filename, mode='r')` function

- **filename**: File name including the **file extension**. Ex: 'data.txt'

Files In Python

Access to a `file` object using `open(filename, mode='r')` function

- **filename**: File name including the **file extension**. Ex: 'data.txt'
- If you want to access/create a file outside of current **working directory**, you also need to include path. Ex: './FolderName/data.txt', 'C:/Users/AUYSAL16/Desktop/data.txt'

Files In Python

Access to a `file` object using `open(filename, mode='r')` function

- **filename**: File name including the **file extension**. Ex: 'data.txt'
- If you want to access/create a file outside of current **working directory**, you also need to include path. Ex: './FolderName/data.txt', 'C:/Users/AUYSAL16/Desktop/data.txt'
- **mode** denotes how the file will be used:

Files In Python

Access to a `file` object using `open(filename, mode='r')` function

- **filename**: File name including the **file extension**. Ex: 'data.txt'
- If you want to access/create a file outside of current **working directory**, you also need to include path. Ex: './FolderName/data.txt', 'C:/Users/AUYSAL16/Desktop/data.txt'
- **mode** denotes how the file will be used:
 - 'r': read mode, default

Files In Python

Access to a `file` object using `open(filename, mode='r')` function

- **filename**: File name including the **file extension**. Ex: 'data.txt'
- If you want to access/create a file outside of current **working directory**, you also need to include path. Ex: './FolderName/data.txt', 'C:/Users/AUYSAL16/Desktop/data.txt'
- **mode** denotes how the file will be used:
 - 'r': read mode, default
 - 'w': write mode, overrides the file contents if it already exists

Files In Python

Access to a `file` object using `open(filename, mode='r')` function

- **filename**: File name including the **file extension**. Ex: 'data.txt'
- If you want to access/create a file outside of current **working directory**, you also need to include path. Ex: './FolderName/data.txt', 'C:/Users/AUYSAL16/Desktop/data.txt'
- **mode** denotes how the file will be used:
 - 'r': read mode, default
 - 'w': write mode, overrides the file contents if it already exists
 - 'x': create & write mode, similar to write mode gives error if file already exists

Files In Python

Access to a `file` object using `open(filename, mode='r')` function

- **filename**: File name including the **file extension**. Ex: 'data.txt'
- If you want to access/create a file outside of current **working directory**, you also need to include path. Ex: './FolderName/data.txt', 'C:/Users/AUYSAL16/Desktop/data.txt'
- **mode** denotes how the file will be used:
 - 'r': read mode, default
 - 'w': write mode, overrides the file contents if it already exists
 - 'x': create & write mode, similar to write mode gives error if file already exists
 - 'a': append mode, adds content to the end of file

File Methods

How to read file content?

File Methods

How to read file content?

- First open the file `f = open('my_file.txt')`

File Methods

How to read file content?

- First open the file `f = open('my_file.txt')`
- `f.read()`: returns content of entire file as a string

File Methods

How to read file content?

- First open the file `f = open('my_file.txt')`
- `f.read()`: returns content of entire file as a string
- `f.readline()`: returns a single line from file

File Methods

How to read file content?

- First open the file `f = open('my_file.txt')`
- `f.read()`: returns content of entire file as a string
- `f.readline()`: returns a single line from file
- **for** line **in** f: \Rightarrow Iterate over all lines

File Methods

How to read file content?

- First open the file `f = open('my_file.txt')`
- `f.read()`: returns content of entire file as a string
- `f.readline()`: returns a single line from file
- **for** line **in** f: \Rightarrow Iterate over all lines
- `list(f)/f.readlines()`: read file lines to a list

File Methods

How to read file content?

- First open the file `f = open('my_file.txt')`
- `f.read()`: returns content of entire file as a string
- `f.readline()`: returns a single line from file
- **for** line **in** f: \Rightarrow Iterate over all lines
- `list(f)/f.readlines()`: read file lines to a list
- **Always** close the file when you are done: `f.close()`

File Methods

How to create/modify files?

- Open the file with a write enabled mode, e.g, `w`, `x`, `a`

File Methods

How to create/modify files?

- Open the file with a write enabled mode, e.g, `w`, `x`, `a`
- Ex: `f = open('my_file', 'w')`

File Methods

How to create/modify files?

- Open the file with a write enabled mode, e.g, `w`, `x`, `a`
- Ex: `f = open('my_file', 'w')`
- Use `f.write(string)` to write to file

File Methods

How to create/modify files?

- Open the file with a write enabled mode, e.g, `w`, `x`, `a`
- Ex: `f = open('my_file', 'w')`
- Use `f.write(string)` to write to file
- `file.write()` method **only** takes **str** values!

File Methods

How to create/modify files?

- Open the file with a write enabled mode, e.g, `w`, `x`, `a`
- Ex: `f = open('my_file', 'w')`
- Use `f.write(string)` to write to file
- `file.write()` method **only** takes **str** values!
- Close the file when you are done.

File Methods

How to create/modify files?

- Open the file with a write enabled mode, e.g, `w`, `x`, `a`
- Ex: `f = open('my_file', 'w')`
- Use `f.write(string)` to write to file
- `file.write()` method **only** takes **str** values!
- Close the file when you are done.
- `f.close()`

Context Managers

What if something bad happens before we close the file?

```
f = open('my_file.txt', 'r') as f:
    # Content of my_file.txt: '1,0,2'
values = f.read().split(',')
# What happens?
result = int(values[0]) / int(values[1])
f.close()
```

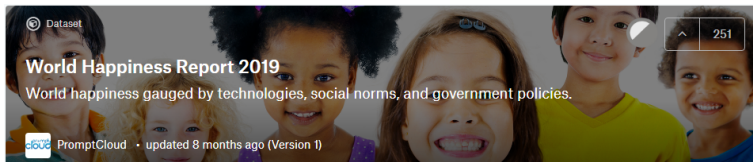
```
# Safer approach, file is closed
# even when we encounter an exception
with open('my_file.txt', 'w') as f:
    f.write('Hello, world!')
```

1. Recap
○○○○○○○
○○○○○○○

2. Error/Exception Handling
○○○○

3. File Input/Output
○○○○○●

Example: World Happiness Report 2019



Open repl.it to code along.