




Lecture 6

Dictionaries
Exceptions





Dictionary

Dictionaries

- Allows you to associate **values** with **keys**.
 - (Classic example: phone book)
- Dictionaries are **unordered** collections of key-value pairs
- (**key**, **value**):
 - **Key**: **unique** identifier (where we can find our data)
 - **Value**: the data to find.
- Think about real dictionary: What is the key in this case?

Example

```
>>> student = {'name': "Mike", 'age': 19, 'courses': ["dsc10", "dsc20"]}
{'name': 'Mike', 'age': 19, 'courses': ['dsc10', 'dsc20']}
```

```
>>> student['name']
'Mike'
```

Values can be of any type.

Keys can be any *immutable* data type.

```
>>> bad_dict = {'name': "Mike"}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Example

```
>>> student = {'name': "Mike", 'age': 19, 'courses': ["dsc10", "dsc20"] }
```

```
>>> student['id'] # access the key that does not exist
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

KeyError: 'id'

```
>>> student.get('name') # return None or some default value. Use .get() instead
'Mike'
```

```
>>> student.get('id')
>>> # Nothing happens, none returned
```

```
>>> student.get('id', 'Not found')
'Not found'
```

Example

```
>>> student = {'name': "Mike", 'age': 19, 'courses': ["dsc10", "dsc20"]}
>>> student['id'] = 'A123456'           # adding new entry to a dictionary
>>> student
{'name': 'Mike', 'age': 19, 'courses': ['dsc10', 'dsc20'], 'id': 'A123456'}

# what will happen if the key already exists? The value will change
# update multiple values at the time. Use .update

>>> student.update({'name': 'Nick', 'id': 'A654321'})
>>> student
{'name': 'Nick', 'age': 19, 'courses': ['dsc10', 'dsc20'], 'id': 'A654321'}
```

Example

```
>>> student = {'name': "Mike", 'age': 19, 'courses': ["dsc10", "dsc20"]}
>>> del student['courses']      # remove courses. Use del
>>> student
{'name': 'Nick', 'age': 19, 'id': 'A654321'}

>>> student.keys()              # you can get all keys, values and pairs at once:
dict_keys(['name', 'age', 'id'])

>>> student.values()
dict_values(['Nick', 19, 'A654321'])

>>> student.items()
dict_items([('name', 'Nick'), ('age', 19), ('id', 'A654321')])
```

Question

```
confusion = {}  
confusion[1] = 1  
confusion['1'] = 2  
confusion[1.0] = 4  
  
sum = 0  
for k in confusion:  
    sum += confusion[k]  
  
print(sum)
```

Output?

- A: 2
- B: 4
- C: 6
- D: 7
- E: Error

Question

```
confusion = {}  
confusion[1] = 1  
confusion['1'] = 2  
confusion[1.0] = 4  
  
sum = 0  
for k in confusion:  
    sum += confusion[k]  
  
print(sum)
```

Output?

- A: 2
- B: 4
- C: 6 # 1 == 1.0 is True.
- D: 7
- E: Error

Question

```
numberGames = {}  
numberGames[(1,2,4)] = 8  
numberGames[(4,2,1)] = 10  
numberGames[(1,2)] = 12  
  
sum = 0  
for k in numberGames.keys():  
    sum += numberGames[k]  
  
print(len(numberGames) + sum)
```

Output?

- A: 8
- B: 12
- C: 24
- D: 30
- E: 33



Exceptions



Why?

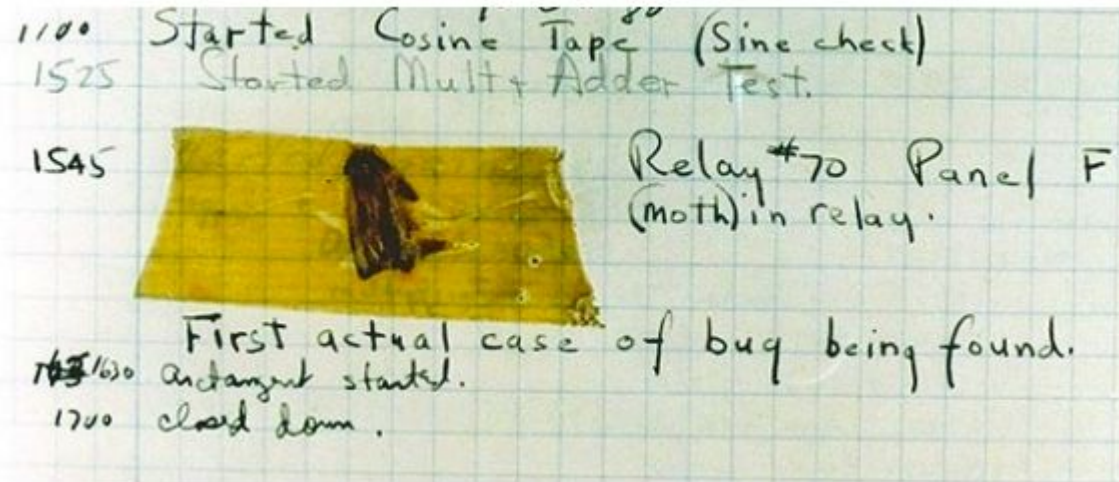
- An **exception** is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.
 - A function receives an argument value of an improper type
 - Some resource (such as a file) is not available
 - A network connection is lost in the middle of data transmission
- In general, when a Python script encounters a situation that it cannot cope with, it raises an **exception**.
- An **exception** is a Python *object* that represents an error.

Difference between `assert` and `exception`

- `try/except` blocks let you catch and manage exceptions. Exceptions can be triggered by `raise`, `assert`, and a large number of errors such as trying to index an empty list. `raise` is typically used when you have detected an error condition.
- `raise` and `assert` have a different philosophy. There are many "normal" errors in code that you detect and raise errors on. Perhaps a web site doesn't exist or a parameter value is out of range.
- Assertions are generally reserved for "I swear this cannot happen" issues that seem to happen anyway. It's more like runtime debugging than normal runtime error detection. Assertions can be disabled if you use the `-O` flag or run from `.pyo` files instead of `.pyc` files, so they should not be part of regular error detection.
- If production quality code raises an exception, then figure out what you did wrong. If it raises an `AssertionError`, you've got a bigger problem.

Handling Errors

- Sometimes, computer programs behave in non-standard ways
 - A function receives an argument value of an improper type
 - Some resource (such as a file) is not available
 - A network connection is lost in the middle of data transmission



Grace Hopper's Notebook, 1947, Moth found in a Mark II Computer

Exceptions

- A *built-in* mechanism in a programming language to declare and respond to exceptional conditions
- Python **raises** an **exception** whenever an error occurs
- Exceptions can be **handled** by the program, preventing the interpreter from halting (come to an abrupt stop)
- *Unhandled* exceptions will cause Python to halt execution and print a *stack trace*
 - *Long message that tells you what line has what error*

Raising Exceptions

Assert Statements

- Assert statements raise an exception of type `AssertionError`

assert <expression> <string>

- Assertions are designed to be used *liberally*.
- They can be ignored to increase efficiency by running Python with the "-O" flag; "O" stands for optimized

python3 -O

- Whether assertions are enabled is governed by a bool `__debug__`

(Demo)

Raise Statements

Exceptions are raised with a `raise` statement

```
raise <expression>
```

`<expression>` must evaluate to a subclass of `BaseException` or an instance of one.

Exceptions are constructed like any other object. E.g., `TypeError('Bad argument!')`

Raise Statements

Exceptions are raised with a `raise` statement

```
raise <expression>
```

<expressions> must evaluate to a subclass of `BaseException` or an instance of one

Exceptions are constructed like any other object. E.g., `TypeError('Bad argument!')`

`TypeError` -- A function was passed the wrong number/type of argument

`NameError` -- A name wasn't found

`KeyError` -- A key wasn't found in a dictionary

`RuntimeError` -- Catch-all for troubles during interpretation

Raise Statements

Exceptions are raised with a `raise` statement

```
raise <expression>
```

<expressions> must evaluate to a subclass of `BaseException` or an instance of one

Exceptions are constructed like any other object. E.g., `TypeError('Bad argument!')`

`TypeError` -- A function was passed the wrong number/type of argument

`NameError` -- A name wasn't found

`KeyError` -- A key wasn't found in a dictionary

`RuntimeError` -- Catch-all for troubles during interpretation

Try Statements

Try statements

Try statements handle exceptions

```
try:  
    <try suite>  
except <exception class> as <name>:  
    <except suite>  
...
```

Handling Exceptions

Exception handling can prevent a program from terminating

```
>>> try:  
    x = 1/0
```

Handling Exceptions

Exception handling can prevent a program from terminating

```
>>> try:  
    x = 1/0  
except ZeroDivisionError as e:
```


Handling Exceptions

Exception handling can prevent a program from terminating

```
>>> try:
    x = 1/0
except ZeroDivisionError as e:
    print('handling a', type(e))
```

Handling Exceptions

Exception handling can prevent a program from terminating

```
>>> try:
    x = 1/0
except ZeroDivisionError as e:
    print('handling a', type(e))
x = 0
```

Handling Exceptions

Exception handling can prevent a program from terminating

```
>>> try:
    x = 1/0
except ZeroDivisionError as e:
    print('handling a', type(e))
    x = 0

handling a <class 'ZeroDivisionError'>
>>> x
0
```

Try statements

Try statements handle exceptions

```
try:  
    <try suite>  
except <exception class> as <name>:  
    <except suite>  
...
```

Execution rule:

- The `<try suite>` is executed first
- If, during the course of executing the `<try suite>`, an exception is raised and
- the `<exception class>` is executed, with `<name>` bound to the exception

Handling Exceptions

Exception handling can prevent a program from terminating

```
>>> try:
    x = 1/0
except ZeroDivisionError as e:
    print('handling a', type(e))
    x = 0

handling a <class 'ZeroDivisionError'>
>>> x
0
```

Multiple try statements: Control jumps to the except suite of the most recent try statement that handles that type of exception

Handling Exceptions

Exception handling can prevent a program from terminating

```
>>> try:
    x = 1/0
except ZeroDivisionError as e:
    print('handling a', type(e))
    x = 0

handling a <class 'ZeroDivisionError'>
>>> x
0
```

(Demo)

WWPD?

How will the Python interpreter respond?

```
def invert(x):  
    inverse = 1/x # Raises a ZeroDivisionError if x is 0  
    print('Never printed if x is 0')  
    return inverse  
  
def invert_safe(x):  
    try:  
        return invert(x)  
    except ZeroDivisionError as e:  
        return str(e)
```

WWPD?

How will the Python interpreter respond?

```
def invert(x):  
    inverse = 1/x # Raises a ZeroDivisionError if x is 0  
    print('Never printed if x is 0')  
    return inverse
```

```
def invert_safe(x):  
    try:  
        return invert(x)  
    except ZeroDivisionError as e:  
        return str(e)
```

```
>>> invert_safe(1/0)
```

A: ZeroDivisionError: division by zero

B: Never printed if x is 0

C: 'division by zero' # str(e)

D: Something else

WWPD?

How will the Python interpreter respond?

```
def invert(x):  
    inverse = 1/x # Raises a ZeroDivisionError if x is 0  
    print('Never printed if x is 0')  
    return inverse
```

```
def invert_safe(x):  
    try:  
        return invert(x)  
    except ZeroDivisionError as e:  
        return str(e)
```

```
>>> try:  
...     invert_safe(0)  
... except ZeroDivisionError as e:  
...     print('Hello!')
```

A: ZeroDivisionError: division by zero

B: Never printed if x is 0

C: 'division by zero' # str(e)

D: Hello!

E: Something else

WWPD?

How will the Python interpreter respond?

```
def invert(x):  
    inverse = 1/x # Raises a ZeroDivisionError if x is 0  
    print('Never printed if x is 0')  
    return inverse  
  
def invert_safe(x):  
    try:  
        return invert(x)  
    except ZeroDivisionError as e:  
        return str(e)
```

```
>>> invert_safe(1/0)
```

More

```
>>> x = 17 + "Marina"
```

More

```
>>> x = 17 + "Marina"
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

More

try:

```
x = 17 + "Marina"
```

except:

```
print("Oouch")
```

Oouch

Pass: ignore and move on

try:

```
x = 17 + "Marina"
```

except:

```
pass
```

```
(cont.from here)
```

Nothing will happen and the code resumes its execution from `(cont. from here)`

Pass: ignore and move on

try:

```
x = 17 + "Marina"
```

except:

```
pass
```

Nothing will happen

Can be used with for/while loops as well

Difference between pass and continue

```
>>> a = [0, 1, 2]
>>> for element in a:
...     if not element:
...         pass
...     print element
...
0
1
2
>>> for element in a:
...     if not element:
...         continue
...     print element
...
1
2
```


What do you think will happen?

```
try:  
  
    a = 1/0  
  
except NameError as e:  
  
    print("printed error")
```

A: printed error

B: infinity

C: ZeroDivisionError: division by zero

D: ZeroDivisionError: division by zero AND
printed error

E: Something else