**Iterables**

* A central abstraction in python is the notion of an **iterable**: an object from which you can fetch a sequence of other objects.  
  The act of fetching a sequence from an iterable object is known as **iteration**.
* **What are Comprehensions in Python**:   
  A concise syntax for describing lists, sets, or dictionaries in a declarative or functional style.   
  This shorthand is readable and expressive meaning that comprehensions are very effective at communicating intent to human readers.
* **List Comprehension:**
* List comprehension is enclosed in square brackets just like a literal list, but instead of literal elements it contains a fragment of declarative code, which describes how to construct the elements of the list.
* General form of list comprehensions: [expr(item) for item in iterable].  
  That is, for each item in the iterable object on the right, we evaluate the expression on the left, and use that as the next element of this new list. The expression on the left is almost always in terms of the item, but that is not mandatory.
* The source object can be any iterable object such as a tuple.   
  The expression can be any Python expression (which may or may not be in terms of the item)
* The type of object produced by list comprehensions is a regular list

|  |  |
| --- | --- |
| **List Comprehension** | **Equivalent for loop** |
| In [**8**]: words="An enhanced Interactive Python."  In [**9**]: print(words)  An enhanced Interactive Python.  In [**10**]: [len(word) for word in words.split()]  Out[**10**]: [2, 8, 11, 7]  In [**17**]: type([len(word) for word in words.split()])  Out[**17**]: list | In [**14**]: lengths=[]  In [**15**]: for word in words.split():      ...: lengths.append(len(word))  In [**16**]: print(lengths)  [2, 8, 11, 7] |

* **Set Comprehension:**
* Set supports similar comprehension syntax using curly braces instead of square brackets.   
  Note that the resulting set will not be stored in a meaningful order since sets are unordered containers.

Use this to remove duplicates.

* General form of Set comprehensions: {expr(item) for item in iterable}

|  |
| --- |
| **Set Comprehension** |
| In [**18**]: {len(word) for word in words.split()}  Out[**18**]: {2, 7, 8, 11}  In [**19**]: type({len(word) for word in words.split()})  Out[**19**]: set |

* **Dictionary Comprehension:**
* Dictionary Comprehension also uses curly braces but is distinguished from the set comprehension by the fact that we now provide two colon separated expressions for the key and value, which will be evaluated in tandem for each item.
* General form of Set comprehensions: {key\_expr:value\_expr for item in iterable}

|  |
| --- |
| **Dictionary Comprehension** |
| In [**20**]: {word:len(word) for word in words.split()}  Out[**20**]: {'An': 2, 'enhanced': 8, 'Interactive': 11, 'Python.': 7}  In [**21**]: type({word:len(word) for word in words.split()})  Out[**21**]: dict |

* One use for a dictionary comprehension is to invert a dictionary so we can perform efficient lookups in the opposite direction.

Note: Dictionary comprehensions do not work directly on dict sources. Use dict.items() to get keys and values from dict sources, and then use tuple unpacking to access the key and values separately.

In [**22**]: from pprint import pprint as pp

    ...: d1={word:len(word) for word in words.split()}

In [**23**]: pp(d1)

{'An': 2, 'Interactive': 11, 'Python.': 7, 'enhanced': 8}

In [**25**]: invertedd1={v:k for k,v in d1.items()}

In [**26**]: pp(invertedd1)

{2: 'An', 7: 'Python.', 8: 'enhanced', 11: 'Interactive'}

* **Caution**: If Dictionary comprehension produce some identical keys, later keys will override earlier keys.
* **Limit on expression complexity in Comprehensions:** There is no limit to the complexity of the expression we can use in any of the comprehensions, we should avoid going overboard and extract complex expressions into separate functions to preserve readability.
* **Filter predicates in Comprehension:** All three types of collection comprehension support an optional filtering clause, which allows us to choose which items of the source are evaluated by the expression on the left.

General form of List expression with Filter predicates: [expr(item) for item in iterable if predicate(item)]

In [**28**]: [x\*x for x in range(1,11) if x%2==0]

Out[**28**]: [4, 16, 36, 64, 100]

* Comprehensions are often more readable than the alternative, however sometimes a long or complex comprehension may be less readable than the equivalent for loop. There's no hard and fast rule about when one form should be preferred, but we should be conscientious when writing our code. We should try to choose the best form for your situation.
* **Comprehensions should ideally be Purely Functional:** They should have no side effects. If we need to create side effects such as printing to the console during iteration, use another construct such as a for loop instead.
* **Iterable and Iterator Protocol:**
* Comprehensions and for loops are the most frequently used language features for performing iteration. That is, taking items one-by-one from a source and doing something with each in turn. However, both comprehensions and for loops iterate over the whole sequence by default whereas sometimes more fine-grain control is needed. This is provided by **iterable and iterator objects.**
* The **iterable protocol** allows us to pass an iterable object, usually a collection or stream of objects such as a list, to the **built-in iter() function to get an iterator** for the iterable object. In short, Iterable is any object that returns and iterator when passed to iter function.
* **Iterator objects support the iterator protocol**, which requires that we can pass the iterator object to the built-in **next() function** to fetch the next value from the underlying collection.
* **Example**: In the below example we ask our iterable object to give us an iterator using the built-in iter function, and then request a value from the iterator using the next function. Each call to next moves the iterator through the sequence

In [**29**]: iterable=["spring","summer","autumn","winter"]

In [**30**]: type(iterable)

Out[**30**]: list

In [**31**]: iterator=iter(iterable)

In [**32**]: type(iterator)

Out[**32**]: list\_iterator

In [**33**]: next(iterator)

Out[**33**]: 'spring'

In [**34**]: next(iterator)

Out[**34**]: 'summer'

In [**35**]: next(iterator)

Out[**35**]: 'autumn'

In [**36**]: next(iterator)

Out[**36**]: 'winter'

In [**37**]: next(iterator)

Traceback (most recent call last):

File "<ipython-input-37-4ce711c44abc>", line 1, in <module>

next(iterator)

StopIteration

Note that when we reach the end, python raises StopIteration Exception.

* Higher-level iteration constructs such as for loops and comprehensions are built directly upon this lower-level iteration protocol.
* **Generators:**
* Python generators provide the means for describing iterable series with code and functions. These sequences are evaluated lazily meaning they only compute the next value on demand. This important property allows them to model infinite sequences of values with no definite end such as streams of data from a sensor or active log files. Ex: Simple code to send events to event Hub.
* Generators are defined by any Python function which uses the yield keyword at least once (May have many times or has it as part of a loop) in its definition. They may also contain the return keyword with no arguments.(This return can be useful if we want to terminate the stream based on some condition.) And just like any other function, there's an implicit return at the end of the definition.

In [**38**]: def gen123():

...: yield 1

...: yield 2

...: yield 3

...:

...:

...: g=gen123()

In [**39**]: type(g)

Out[**39**]: generator

In [**40**]: print(g)

<generator object gen123 at 0x0000020E121FED68>

* **Generators are in fact Python iterators**, so we can use the standard ways of working with iterators to retrieve or yield successive values from the sequence. To retrieve the next value from an iterator, we use the built- in next function passing the iterator or generator in this case to the function.

Because generators are iterators, they can be used in all the usual Python constructs which expect iterators such as for loops.

Like iterators if we call next after the last Item we get StopIteration Exception.

In [**41**]: next(g)

Out[**41**]: 1

In [**42**]: next(g)

Out[**42**]: 2

In [**43**]: next(g)

Out[**43**]: 3

In [**44**]: next(g)

Traceback (most recent call last):

File "<ipython-input-44-e734f8aca5ac>", line 1, in <module>

next(g)

StopIteration

* **Each call to the generator function returns a new generator object.** This means that each generator can be advanced independently.

In [**46**]: g1=gen123()

    ...: g2=gen123()

In [**47**]: next(g1)

Out[**47**]: 1

In [**48**]: next(g1)

Out[**48**]: 2

In [**49**]: next(g2)

Out[**49**]: 1

* **Following shows how generators work internally.** When the generator g is created none of the code within the generator body has yet been executed. When we request the first value the generator body runs up to and including the first yield statement. The code executes just far enough to literally yield the next value. When we call next(g) again, execution of the generator function resumes at the point it left off and continues running until the next yield. After the final value is returned, the next request causes the generator function to execute until it returns at the end of the function body, which in turn raises the expected StopIteration exception.

In [**60**]: def gen123():

    ...: for i in range(3):

    ...: print("About to yield "+str(i))

    ...: yield i\*i

In [**61**]: g=gen123()

In [**62**]: next(g)

About to yield 0

Out[**62**]: 0

In [**63**]: next(g)

About to yield 1

Out[**63**]: 1

In [**64**]: next(g)

About to yield 2

Out[**64**]: 4

* Note that generator functions, which resume execution each time the next value is requested, can maintain state in local variables. This means we can have counters defined inside the generator. We can use these stateful local variables to check conditions and may be exit the generator function.

Following shows the counter local variable which maintains state:

In [**94**]: def take(count,iterable1):

    ...: counter=0

    ...: for i in iterable1:

    ...: if counter == count:

    ...: return

    ...: else:

    ...: yield i

    ...: counter = counter+1

In [**95**]: list1=["Mumbai","Delhi","Hyder","Bang"]

    ...: g1=take(2,list1)

In [**96**]: for j in g1:

    ...: print(j)

Mumbai

Delhi

* **Generators are lazy** meaning that computation only happens just in time when the next result is requested. This interesting and useful property of generators means that they can be used to model infinite sequences. Ex: Simulating continuous events to be sent to Azure Event Hub Since values are only produced as requested by the caller and since no data structure needs to be built to contain the elements of the sequence, generators can safely be used to produce never ending or just very large sequences like sensor readings, mathematical sequences such as primes or factorials, or perhaps the contents of multi-terabyte files.
* **Generator Comprehension:**
* Generator expressions are a cross between comprehensions and generator functions. They use a similar syntax as comprehensions, but they result in the creation of a generator object, which produces the specified sequence lazily.
* The syntax for generator expressions is very similar to list comprehensions (expr(item) for item in iterable) delimited by parentheses instead of the brackets used for list comprehensions.
* Generator expressions are useful for situations where you want the lazy evaluation of generators with the declarative concision of comprehensions.

In [**8**]: millionsquares=(i\*i for i in range(100001))

In [**9**]: print(millionsquares)

<generator object <genexpr> at 0x00000262E5C95C00>

In [**10**]: list(millionsquares)[-10:]

Out[**10**]:

[9998200081,

9998400064,

9998600049,

9998800036,

9999000025,

9999200016,

9999400009,

9999600004,

9999800001,

10000000000]

In [**11**]: list(millionsquares)[-10:]

Out[**11**]: []

Note that in code 9, no squares have been created yet. We can force evaluation of a generator by converting to generator to a list.

Note that generator does not take any memory, but when we convert it to list it consumes a significant amount of memory.

* **Imp:** Just like Iterators, Generators are single use objects. Once exhausted, it cannot yield more items.

Notice that 2nd time we try to fetch the last 10 element, we get empty list.

* Each time we call a generator function, we create a new generator object.   
  To recreate a generator from a generator expression, we must execute the expression itself once more.
* Memory Usage: To compute sum of squares of 1st 10 million number will take lof of space if we 1st create a list of 10 million numbers. However, if we make use of generators we will get the same result, but the amount of memory consumed will be very less.

In [**13**]: sum(i\*i for i in range(1000010))

Out[**13**]: 333342833423500285

In [**14**]: sum([i\*i for i in range(1000010)])

Out[**14**]: 333342833423500285

Note that we didn't supply separate enclosing parentheses for the generator expression in addition to those needed for the sum function call. This elegant ability to have the parentheses used for the function call also serve for the generator expression aids readability. You can include the second set of parentheses if you wish, but it's not required.

* As with comprehensions, we can include an if clause at the end of the generator expression.
* **Additional Iteration Functionality:**
* Python provides several built-in functions for performing common iterator operations. These functions form the call of a sort of vocabulary for working with iterators, and they can be combined to produce powerful statements in very concise, readable code. Examples are **enumerate** for producing integer indices and **sum** for computing summation of numbers. We also have max,min etc
* The **itertools** module contains a wealth of useful functions and generators for processing iterable streams of data.
* The itertools **islice** allows us to perform lazy slicing like the built-in list slicing functionality.
* The itertools **count** allows us to get open-ended version of range. (Note that range is not open ended. It needs to know how many items to create)

Following shows how to generate list of 1st 1000 prime numbers:

for i in itertools.islice((x for x in itertools.count() if is\_prime(x)),1000):

print(i)

* The itertools **chain** allows us to lazily concatenate iterables without having to create a new list. Thus without the memory impact of data duplication.
* **Iteration Built-ins:**
* **Boolean Aggregation:**
  + The **any()** determines if any of the element in Series are True.
  + The **all()** determines if all of the elements in Series are True.  
    Using comprehension with any and all makes it easy to check for a test over entire iterable and give a collective result if True and False.

In [**15**]: any([True,False,False])

Out[**15**]: True

In [**16**]: all([True,False,False])

Out[**16**]: False

In [**17**]: all([True,True])

Out[**17**]: True  
In [**18**]: any([x%2==0 for x in range(1,100)])  
Out[**18**]: True  
In [**20**]: names=['London','Tokya','Paros','Sydney']  
    ...: all([name == name.title() for name in names])  
Out[**20**]: True

* **Zip:** Synchronize iterations over two or more Iterables.

That zip yields tuples when iterated. This in turn means we can use it with tuple unpacking in the for loop.

Zip can accept any number of iterable arguments.

* **Summary:**
* Comprehensions are a concise and readable syntax for describing lists, sets, and dictionaries in a declarative way. These comprehensions iterate on an iterable source object and apply an optional predicate filter and a mandatory expression. Both filter and expression are usually in terms of the current item.
* Iterable objects are objects over which we can iterate item-by-item.
* We retrieve an iterate all from an iterable using the built-in iter() function.
* Iterators produce items one-by-one from the underlying iterable series each time they are passed to the built-in next() function.
* When the series is exhausted, iterators raise a StopIteration exception.
* Generator functions look just like regular functions and have all the same facilities, but they must contain at least one instance of the yield keyword.
* Generators are iterators.
* When the iterator is advanced with next(), the generator starts or resumes execution up to and including the next yield statement.
* Each call to a generator function creates a new generator object.
* Generators can maintain state between calls in local variables and because they are lazy can model infinite series of data.
* Generator expressions are a sort of hybrid of generator functions and list comprehensions. These allow for a more declarative and concise way of creating generator objects.
* Python includes a rich set of tools for dealing with iterable series both in the form of built-in functions such as sum(), any(), and zip(), but also in the itertools module.

**Classes**

* **Classes:**
  + Pythons tool to create new types.
  + All objects in Python have a type, and we report that type using the type built-in function ,the result is couched in terms of class.
  + Classes are a means of defining the structure and behavior of objects at the time we create the object.
  + Classes act as a sort of template or pattern according to which new objects are constructed. The class of an object controls its initialization in which attributes and methods are available through the object.
  + Python is highly object oriented without forcing us to deal with classes until we really need them. This sets the language starkly apart from Java and C#. This means unlike Java, C#, not every code needs to be a part of some class.
  + Class definitions are introduced by the class keyword followed by the class name. By convention, the new class name in Python uses CamelCase, sometimes known as PascalCase with a capital letter for each component word.
  + The class statement introduces a new block, so we indent on the next line. Empty blocks aren't allowed, So the simplest class that is syntactically admissible is as follows:

class abc:  
 pass

* + Just as with def for defining functions, class is a statement that can occur anywhere in a program and which binds a class definition to a class name. The def binds function name to the function definition.
  + For class Flight in module airtravel.py, we use “from airtravel import Flight” to import it in our code/REPL. The thing we've just imported is the class object. Everything is an object in Python, and classes are no exception
  + To create a class object we call its constructor, which is done by calling the class as we would a function. Ex: f=Flight()
  + What are methods: Methods are just functions defined within the class
  + What are instance methods: Instance methods are functions which can be called on objects or instances of our class.
  + Instance methods must accept a reference to the instance on which the method was called as the first argument, and by convention this argument is always called self.
  + When we call the method, we do not provide the instance as the actual argument itself in the argument list. That's because the standard method invocation form with the dot is simply syntactic sugar for the class name followed by a dot followed by the method name with the instance passed as the first argument. Ex: f.mymethod() is same as classname.method(f). Both give the same result.
  + The initializer method must be called \_\_init\_\_ . If provided, the initialization method is called as part of the process of creating a new object when we call the constructor. Like all other instance methods, the first argument to \_\_init\_\_ must be self. Note that having a \_\_init\_\_ method is not mandatory. However if its present its used to initialize the instance.
  + The initializer should not return anything. It simply modifies the object referred to by self.
  + Although it's tempting to think of \_\_init\_\_ as being the constructor, this isn't quite accurate. In Python, the purpose of \_\_init\_\_ is to configure an object that already exists by the time it's called. In Python, the actual constructor is provided by the Python runtime system, and one of the things it does is check for the existence of an instance initializer and call it when present. Any arguments passed to the flight constructor will be forwarded to the initializer. The self argument is analogous to this in Java, C#, or C++.
  + Assigning to an object attribute that doesn't yet exist is enough to bring it into being. Just as we don't need to declare variables until we create them, neither do we need to declare object attributes before we create them.
  + Many a times instance attributes are preceded by \_ Ex: \_number.   
    First because it avoids a name clash with the method of the same name.   
    Second, there is a widely followed convention that the implementation details of objects which are not intended for consumption or manipulation by clients of the object should be prefixed with an underscore.
  + **Convention of leading underscores:** In language like Java or C# we have public, private, and protected access modifiers. But in Python's everything is pubic. The leading underscore convention has proven enough protection even in large and complex Python systems we have worked with. People know not to use these attributes directly, and in fact they tend not to.
  + **Performing Checks in \_\_init\_\_:** It's good practice for the initializer of an object to establish so-called class invariants. The invariants are truths(checks) about the objects of that class that should endure for the lifetime of the object. In Python we establish class invariants in the \_\_init\_\_ method and raise exceptions if they can't be attained.  
      
    **class** Flight:  
     *"""A flight with a particular passenger aircraft. """* **def** \_\_init\_\_(self, number, aircraft):  
     **if not** number[:2].isalpha():  
     **raise** ValueError(**"No airline code in '{}'"**.format(number))  
      
     **if not** number[:2].isupper():  
     **raise** ValueError(**"Invalid airline code '{}'"**.format(number))  
      
     **if not** (number[2:].isdigit() **and** int(number[2:]) <= 9999):  
     **raise** ValueError(**"Invalid route number '{}'"**.format(number))
  + We can also add a docstring to the class. These work just like function and module docstrings.
  + Note that method calls within the same object also require explicit qualification with the self prefix.

**def** allocate\_seat(self, seat, passenger):  
 *"""Allocate a seat to a passenger.  
 Args:  
 seat: A seat designator such as '12C' or '21F'.  
 passenger: The passenger name.  
 Raises:  
 ValueError: If the seat is unavailable.  
 """* row, letter = self.\_parse\_seat(seat)  
  
 **if** self.\_seating[row][letter] **is not** None:  
 **raise** ValueError(**"Seat {} already occupied"**.format(seat))  
  
 self.\_seating[row][letter] = passenger

* + Note that a module could have many class definitions and module level functions defined using def keyword. Check below code.
  + Use of line continuation backslash character allows us to split a long statement over several lines. This can be used together with implicit string concatenation of adjacent strings to produce one long string with no line breaks.
  + To join multiple strings on separate lines, we can use the join method on the “\n” string.
  + Polymorphism is a programing language feature which allows us to use objects of different types through a uniform interface. The concept of polymorphism applies to functions and more complex objects.
  + Polymorphism in Python is achieved through duck typing. Duck typing is in turn named after the duck test attributed to James William Riley the American poet. "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck. "

Duck typing where an object's fitness for a particular use is only determined at runtime is the cornerstone of Python's object system. This is in contrast to statically typed languages where a compiler determines if an object can be used. And in particular, it means that an object's suitability is not based on inheritance hierarchies, base classes, or anything except the attributes an object has at the time of use.

Duck typing is the basis for the collection protocols such as iterator, iterable, and sequence.

* + Inheritance is a mechanism whereby one class can be derived from a base-class allowing us to make behavior more specific in the sub-class. In typed languages such as Java, class-based inheritance is how runtime polymorphism is achieved. This is not the case in Python as we saw with the use of duck typing.
  + The fact that no Python method calls or attribute lookups are bound to actual objects until the point at which they are called known as **late binding** means we can attempt polymorphism with any object, and we'll succeed if the object fits.
  + Although inheritance in Python can be used to facilitate polymorphism, after all derived classes will have the same interfaces as the base classes, inheritance in Python is most useful for sharing implementation between classes.
  + We specify inheritance in Python using parentheses containing the base class name immediately after the class name in the class statement. Ex: class Boeing(AirCraft)
  + Due to duck typing, inheritance is less used in Python than in other languages. This is generally seen as a good thing because inheritance is a very tight coupling between classes.

|  |
| --- |
| *"""Model for aircraft flights"""* **class** Flight:  *"""A flight with a particular passenger aircraft. """* **def** \_\_init\_\_(self, number, aircraft):  **if not** number[:2].isalpha():  **raise** ValueError(**"No airline code in '{}'"**.format(number))   **if not** number[:2].isupper():  **raise** ValueError(**"Invalid airline code '{}'"**.format(number))   **if not** (number[2:].isdigit() **and** int(number[2:]) <= 9999):  **raise** ValueError(**"Invalid route number '{}'"**.format(number))   self.\_number = number  self.\_aircraft = aircraft   rows, seats = self.\_aircraft.seating\_plan()  self.\_seating = [None] + [{letter: None **for** letter **in** seats} **for** \_ **in** rows]   **def** number(self):  **return** self.\_number   **def** airline(self):  **return** self.\_number[:2]   **def** aircraft\_model(self):  **return** self.\_aircraft.model()   **def** \_parse\_seat(self, seat):  *"""Parse a seat designator into a valid row and letter.   Args:  seat: A seat designator such as 12F   Returns:  A tuple containing an integer and a string for row and seat.  """* row\_numbers, seat\_letters = self.\_aircraft.seating\_plan()   letter = seat[-1]  **if** letter **not in** seat\_letters:  **raise** ValueError(**"Invalid seat letter {}"**.format(letter))   row\_text = seat[:-1]  **try**:  row = int(row\_text)  **except** ValueError:  **raise** ValueError(**"Invalid seat row {}"**.format(row\_text))   **if** row **not in** row\_numbers:  **raise** ValueError(**"Invalid row number {}"**.format(row))   **return** row, letter   **def** allocate\_seat(self, seat, passenger):  *"""Allocate a seat to a passenger.   Args:  seat: A seat designator such as '12C' or '21F'.  passenger: The passenger name.   Raises:  ValueError: If the seat is unavailable.  """* row, letter = self.\_parse\_seat(seat)   **if** self.\_seating[row][letter] **is not** None:  **raise** ValueError(**"Seat {} already occupied"**.format(seat))   self.\_seating[row][letter] = passenger   **def** relocate\_passenger(self, from\_seat, to\_seat):  *"""Relocate a passenger to a different seat.   Args:  from\_seat: The existing seat designator for the  passenger to be moved.   to\_seat: The new seat designator.  """* from\_row, from\_letter = self.\_parse\_seat(from\_seat)  **if** self.\_seating[from\_row][from\_letter] **is** None:  **raise** ValueError(**"No passenger to relocate in seat {}"**.format(from\_seat))   to\_row, to\_letter = self.\_parse\_seat(to\_seat)  **if** self.\_seating[to\_row][to\_letter] **is not** None:  **raise** ValueError(**"Seat {} already occupied"**.format(to\_seat))   self.\_seating[to\_row][to\_letter] = self.\_seating[from\_row][from\_letter]  self.\_seating[from\_row][from\_letter] = None   **def** num\_available\_seats(self):  **return** sum(sum(1 **for** s **in** row.values() **if** s **is** None)  **for** row **in** self.\_seating  **if** row **is not** None)   **def** make\_boarding\_cards(self, card\_printer):  **for** passenger, seat **in** sorted(self.\_passenger\_seats()):  card\_printer(passenger, seat, self.number(), self.aircraft\_model())   **def** \_passenger\_seats(self):  *"""An iterable series of passenger seating allocations."""* row\_numbers, seat\_letters = self.\_aircraft.seating\_plan()  **for** row **in** row\_numbers:  **for** letter **in** seat\_letters:  passenger = self.\_seating[row][letter]  **if** passenger **is not** None:  **yield** (passenger, **"{}{}"**.format(row, letter))   **class** Aircraft:   **def** \_\_init\_\_(self, registration):  self.\_registration = registration   **def** registration(self):  **return** self.\_registration   **def** num\_seats(self):  rows, row\_seats = self.seating\_plan()  **return** len(rows) \* len(row\_seats)   **class** AirbusA319(Aircraft):   **def** model(self):  **return "Airbus A319"   def** seating\_plan(self):  **return** range(1, 23), **"ABCDEF"   class** Boeing777(Aircraft):   **def** model(self):  **return "Boeing 777"   def** seating\_plan(self):  *# For simplicity's sake, we ignore complex  # seating arrangement for first-class* **return** range(1, 56), **"ABCDEGHJK"   def** make\_flights():  f = Flight(**"BA758"**, AirbusA319(**"G-EUPT"**))  f.allocate\_seat(**'12A'**, **'Guido van Rossum'**)  f.allocate\_seat(**'15F'**, **'Bjarne Stroustrup'**)  f.allocate\_seat(**'15E'**, **'Anders Hejlsberg'**)  f.allocate\_seat(**'1C'**, **'John McCarthy'**)  f.allocate\_seat(**'1D'**, **'Richard Hickey'**)   g = Flight(**"AF72"**, Boeing777(**"F-GSPS"**))  g.allocate\_seat(**'55K'**, **'Larry Wall'**)  g.allocate\_seat(**'33G'**, **'Yukihiro Matsumoto'**)  g.allocate\_seat(**'4B'**, **'Brian Kernighan'**)  g.allocate\_seat(**'4A'**, **'Dennis Ritchie'**)   **return** f, g   **def** console\_card\_printer(passenger, seat, flight\_number, aircraft):  output = **"| Name: {0}"** \  **" Flight: {1}"** \  **" Seat: {2}"** \  **" Aircraft: {3}"** \  **" |"**.format(passenger, flight\_number, seat, aircraft)  banner = **'+'** + **'-'** \* (len(output) - 2) + **'+'** border = **'|'** + **' '** \* (len(output) - 2) + **'|'** lines = [banner, border, output, border, banner]  card = **'\n'**.join(lines)  **print**(card)  **print**() |

* **Summary:**
  + All types in Python have a class. Classes define the structure and behavior of an object. Classes are the key support for object-oriented programming in Python.
  + Classes are defined using the class keyword followed by the class name, which is in CamelCase.
  + Instances of a class are created by calling the class as if it were a function.
  + Instance methods are functions defined inside the class, which should accept an object instance called self as the first parameter.
  + Methods are called using the instance.method() syntax, which is syntactic sugar for passing the instance as the formal self argument to the method.
  + An optional special initializer method called \_\_init\_\_() can be provided, which is used to configure the self object at creation time. If present, the constructor calls the \_\_init\_\_() method. Note that Double underscore init is not the constructor. The object has been constructed by the time the initializer is called. Arguments passed to the constructor are forwarded to the initializer.
  + Instance attributes are brought into existence simply by assigning to them.
  + Attributes and methods which are implementation details are by convention prefixed with an underscore.
  + There are no public, protected, or private access modifiers in Python.
  + Class invariants should be established in the initializer. If the invariants can't be established, raise exceptions to signal failure.
  + Methods can have docstrings, just like regular functions. Classes can have docstrings.
  + Even within an object method, calls must be preceded with self.
  + You can have as many classes and functions in a module as you wish. Related classes and global functions are generally grouped together this way.
  + Polymorphism in Python is achieved through duck typing where attributes and methods are only resolved at point of use. This is called late binding.
  + Polymorphism in Python does not require shared base classes or named interfaces. Class inheritance in Python is primarily useful for sharing implementation rather than being necessary for polymorphism.
  + We can nest comprehensions. It can sometimes be useful to discard the current item in a comprehension using a dummy reference, conventionally the underscore character.  
      
    [‘1’ for \_ in list1]
  + Functions are also objects.
  + Complex comprehensions or generator expressions can be split over multiple lines to aid readability. Statements can be split over multiple lines using the backslash line continuation character. Use this feature sparingly and only when it improves readability.

**Files and resource Management**

* To open a file in Python, we call the built-in open() function.

Common arguments:

1) **File**, the path to the file(required)

2) **Mode**, which specifies read/write/append, and binary or text mode. This is optional, but we always recommend specifying it for clarity. Explicit is better than implicit.

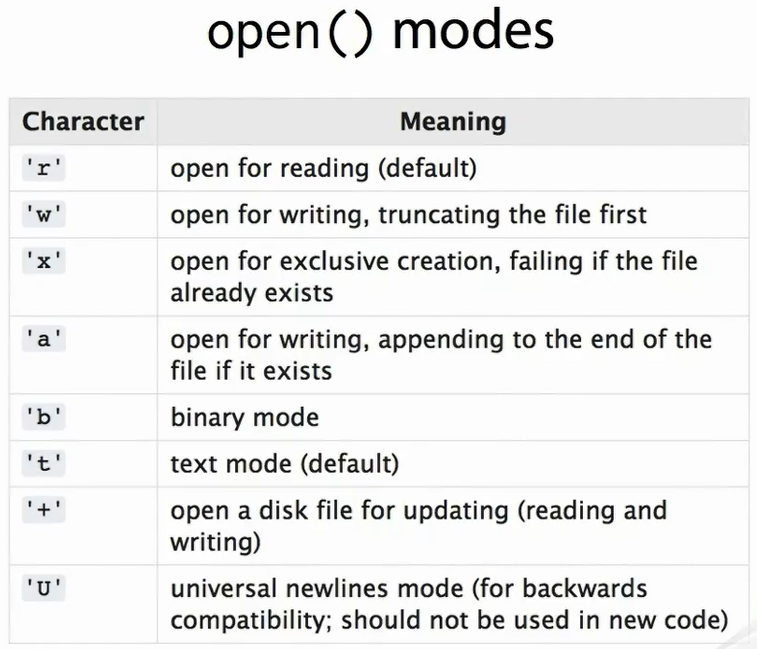
3) **Encoding**. If the file contains encoded text data, this is the text encoding to use. It's often a good idea to specify this. If you don't specify it, Python will choose a default encoding for you.

The exact type of the object returned by open depends on how the file was opened, dynamic typing in action. However, know that the object returned is a file-like object.

* At the file system level, files contain only a series of bytes. Python distinguishes between files opened in binary and text modes even when the underlying operating system doesn't.
  + Files opened in binary mode return and manipulate their contents as bytes objects without any decoding. Binary mode files reflect the raw data in the file.
  + A file opened in text mode treats its contents as if it contains text strings of the str type, the raw bytes having first been decoded using a platform dependent encoding or using the specified encoding if given.   
    By default, text mode also engages support for Python's universal newlines. This causes translation between a single portable newline character in our program strings, /n, and a platform-dependent newline representation in the raw bytes stored in the file system, for example carriage return newline /r/n on Windows.
* **Default Encoding:** Getting the encoding right is crucial for correctly interpreting the contents of a text file. If you don't specify an encoding, Python will use the default from sys.getdefaultencoding.
* **File open Modes:** The mode argument in open builtin function All mode strings should consist of a read, write, or append mode. One of R, W, or A with the optional plus modifier should be combined with a selective text or binary mode T or B. is a string containing letters with different meanings.

f=open(‘wasteland.txt’,mode=’wt’,encoding=’utf-8’)

Both parts of the mode code support defaults, its recommended being explicit for the sake of readability.



* **The write method:** used to write to a file. The write call returns the number of codepoints or characters written to the file. It is the caller's responsibility to provide newline characters where they are needed. There is no writeline method.

When we finish writing, we should remember to close the file by calling the close method.

* The size of the files written on windows and linux may be different. The difference is because Python's universal newline behavior for files has translated the line endings to your platform's native endings. (on windows \n will be translated by python to \r\n).

The number returned by the write method is the number of codepoints or characters in the string passed to write, not the number of bytes written to the file after encoding a universal newline translation. This means when working with text files, you cannot sum the quantities returned by write to determine the length of the file in bytes.

In [**22**]: f1=open('wasteland.txt',mode='wt',encoding='utf-8')

In [**28**]: type(f1)

Out[**28**]: \_io.TextIOWrapper

In [**23**]: f1.write("This is a crazy world\n")

Out[**23**]: 22

In [**24**]: f1.write("filled with stupid ppl")

Out[**24**]: 22

In [**25**]: f1.close()

* **The Read Function:**
  + If we know how many bytes to read or if we want to read the whole file, we can use the read function. In text mode the read method accepts the number of characters to read from the file, not the number of bytes.
  + The call returns the text and advances the file pointer to the end of what was read. Subsequent read call will read next piece of data.
  + In text Mode, the return type is str. In Binary mode, the return type is bytes.(.i.e no encoding)
  + To read all the remaining data in the file, we can call read without an argument. This gives us multiple lines in one string with newline characters embedded in middle.
  + At the end of the file, further calls to read return an empty string.

In [**45**]: f2=open('wasteland.txt',mode='rt',encoding='utf-8')

In [**46**]: type(f2)

Out[**46**]: \_io.TextIOWrapper

In [**47**]: f3=open('wasteland.txt',mode='rb')

In [**48**]: s1=f2.read(5)

In [**49**]: print(s1)

This

In [**50**]: type(s1)

Out[**50**]: str

In [**51**]: b1=f3.read(5)

In [**52**]: print(b1)

b'This '

In [**53**]: type(b1)

Out[**53**]: bytes

In [**54**]: s2=f2.read()

In [**55**]: print(s2)

is a crazy world

filled with stupid ppl

In [**56**]: print(f2.read())

* The **seek method** can be used to move the file pointer to any location. Use 0 offset to move it to start of the file. We can use this to go over the file repeatedly without having to closing and reopening.
* Use **readline() function** to read file line by line. The returned lines are terminated by a single newline character if there is one present in the file. The last line does not terminate with a newline because there is no newline sequence at the end of the file.

Again, the universal newline support will have translated to \n from whatever the platform native newline sequence is. This means on windows \r\n will be translated by python to \n.

Once we reach the end of the file, further calls to readline return an empty string.(Similar to read() method)

* Use **readlines() method** to read all lines into a list. Note that memory may be an issue. This is particularly useful if pausing the file involves hopping backwards and forwards between lines.

In [**57**]: f2.seek(0)

Out[**57**]: 0

In [**58**]: f2.readline()

Out[**58**]: 'This is a crazy world\n'

In [**59**]: f2.readline()

Out[**59**]: 'filled with stupid ppl'

In [**60**]: f2.readline()

Out[**60**]: ''

In [**61**]: f2.seek(0)

Out[**61**]: 0

In [**62**]: f2.readlines()

Out[**62**]: ['This is a crazy world\n', 'filled with stupid ppl']

* To **append** to an existing file, we can open the file with mode a, which opens the file for writing, appending to the end of the file if it already exists.

There is no writeline method in Python, there is a **writelines** method, which writes an iterable series of strings to a stream. If you want line endings on your strings, you must provide them yourself.

In [**66**]: f2=open('wasteland.txt',mode='rt',encoding='utf-8')

    ...: f2.readlines()

    ...: f2.close()

In [**67**]: f2=open('wasteland.txt',mode='rt',encoding='utf-8')

    ...: print(f2.readlines())

    ...: f2.close()

['This is a crazy world\n', 'filled with stupid ppl']

In [**68**]: f3=open('wasteland.txt',mode='at',encoding='utf-8')

In [**69**]: f3.writelines(['most of which want to\n','watch world burn'])

In [**70**]: f3.close()

In [**71**]: f2=open('wasteland.txt',mode='rt',encoding='utf-8')

    ...: print(f2.readlines())

    ...: f2.close()

['This is a crazy world\n', 'filled with stupid pplmost of which want to\n', 'watch world burn']

* File objects support the iterator protocol with each iteration yielding the next line in the file. This means they can be used in for loops and any other place where an iterator can be used.

In [**74**]: f2=open('wasteland.txt',mode='rt',encoding='utf-8')

In [**75**]: for i in f2:

    ...: print(i)

This is a crazy world

filled with stupid pplmost of which want to

watch world burn

The double line spacing occurs because each line of the file is terminated by a newline, and then print adds its own. To fix that we could use the strip method to remove the whitespace from the end of each line prior to printing.

Instead we can use the write method of the standard out stream. Files and streams are closely related and can be used because the stream is a file-like object. We can get hold of a reference to the standard out stream from the sys module.

In [**76**]: import sys

    ...: f2=open('wasteland.txt',mode='rt',encoding='utf-8')

    ...: for i in f2:

    ...: sys.stdout.write(i)

This is a crazy world

filled with stupid pplmost of which want to

watch world burn

* **Context Managers:** When working with files, the **close** method call is important. It informs the underlying OS that we are done working with a file. If we don't close a file, it's possible to lose data. There may be pending rights buffered up, which might not get written completely.

Many a times during exceptions , the close call is never executed.

Furthermore, if you're opening lots of files, your system may run out of resources.

One option to make sure that files are closed no matter what, is to make use of try-finally clause. The finally block will make sure the close call is executed every time (irrespective of how execution exits the try block)

To ease the need for resource cleanup, Python implements a control flow structure called with-block to support it. With-blocks can be used with any object which supports the context-manager protocol, and that includes the file objects returned by open().

We no longer need to call close explicitly because the with construct will call it for us when and by whatever means execution exits the block. This also removes the need for an explicit close.

The with-block syntax is so-called syntactic sugar for a much more complex arrangement of try/except and try/finally blocks.

* **Working with Binary Files:** We open the file for write in binary mode using the 'wb' mode string. With Binary files we don't specify an encoding as that makes no sense for raw binary files. To the write method we should pass bytes object as the file is opened in binary mode. To convert things to bytes, use the bytes constructor and use b’’ for byte literals. Ex: b’\x01’
* **Bitwise operators to work on bytes:**   
  & - bitwise and (Remember than python uses ‘and’ for logical and)  
  | - bitwise OR  
  >> right-shift  
  << left-shift
* **Summary:**
* Files are opened using the built-in open() function, which accepts a file mode. This controls read/write/append behavior and also whether the file is treated as binary or encoded text data.
* For text data, it's good practice to always specify an encoding.
* Text files differ from binary files by dealing with string objects and performing universal newline translation and string encoding. Binary files deal with bytes objects with no newline translation or encoding.
* When you write text files, it's up to us to provide newline characters for line breaks.
* Files should always be closed after use to prevent resource leaks and to ensure that all data has been committed to the file system.
* Files provide various convenient methods for working with lines, but are also iterators, which yield values line-by-line.
* Files are also context mangers and can be used with the with-statement. This ensures that cleanup operations such as closing the files are performed.
* Context managers aren't restricted to file-like objects. We can use the tools in the contextlib standard library module such as the closing() wrapper to create our own context managers.
* Python supports bitwise operators bitwise &, bitwise or, and left- and right-bitwise shifts.

**Shipping Working and maintainable code**

* **The unittest Module:**
* The unittest module can be used to automate testing of our code.
* Python standard library includes the unittest module which is very useful to have a set of tests that you can run which will tell you if you code is acting as you expect it to. This allows us to handle on defects and keep our code quality high.
* This module provides a flexible framework for automating tests of all sorts: acceptance tests, integration tests , unit tests.
* It makes automated and repeatable tests. Meaning that we can cheaply and easily verify our code at any time.
* Key Concepts:
  + **TestCase** groups together a set of related individual test functions. A TestCase is the basic unit of test organization in the unittest framework.
  + **Fixtures** are pieces of code which run before and/or after every test method. Fixtures are used to make sure that the test environment is in an expected state before a test is run. For example, to create a necessary database table or populate a cache. Fixtures are then used to clean up any resources that may have been used in a test method.
  + **Assertions** are how you can tell the unittest framework to make specific checks which determine whether a test passes or fails. Among other things, assertions can make simple Boolean checks, perform object equality tests, or verify that the proper exceptions are thrown. If an assertion fails, then a test function fails. So assertions are really the lowest level of testing you can perform.
* To create test cases with the unittest framework, we create our test case by creating a class which derives from unittest.TestCase.
* To define individual test methods in a test case, you simply create methods to start with test\_. These are automatically discovered by the unittest framework and don't require any sort of explicit registration.

|  |
| --- |
| **import** os **import** unittest   **def** analyze\_text(filename):  *"""Calculate the number of lines and characters in a file.   Args:  filename: The name of the file to analyze.   Raises:  IOError: If ``filename`` does not exist or can't be read.   Returns: A tuple where the first element is the number of lines in  the file and the second element is the number of characters.  """* lines = 0  chars = 0  **with** open(filename, **'r'**) **as** f:  **for** line **in** f:  lines += 1  chars += len(line)  **return** (lines, chars)   **class** TextAnalysisTests(unittest.TestCase):  *"""Tests for the ``analyze\_text()`` function."""* **def** setUp(self):  *"""Fixture that creates a file for the text methods to use."""* self.filename = **'text\_analysis\_test\_file.txt'  with** open(self.filename, **'w'**) **as** f:  f.write(**'Now we are engaged in a great civil war.\n'  'testing whether that nation,\n'  'or any nation so conceived and so dedicated,\n'  'can long endure.'**)   **def** tearDown(self):  *"""Fixture that deletes the files used by the test methods."""* **try**:  os.remove(self.filename)  **except**:  **pass   def** test\_function\_runs(self):  *"""Basic smoke test: does the function run."""* analyze\_text(self.filename)   **def** test\_line\_count(self):  *"""Check that the line count is correct."""* self.assertEqual(analyze\_text(self.filename)[0], 4)   **def** test\_character\_count(self):  *"""Check that the character count is correct."""* self.assertEqual(analyze\_text(self.filename)[1], 131)   **def** test\_no\_such\_file(self):  *"""Check the proper exception is thrown for a missing file."""* **with** self.assertRaises(IOError):  analyze\_text(**'foobar'**)   **def** test\_no\_deletion(self):  *"""Check that the function doesn't delete the input file."""* analyze\_text(self.filename)  self.assertTrue(os.path.exists(self.filename))   **if** \_\_name\_\_ == **'\_\_main\_\_'**:  unittest.main() |

* **Unittest.main:**In the module, we define the idiomatic main block which calls unittest.main. when this module is executed. unittest.main will search for all TestCase subclasses in a module and execute all of their test methods.

The unittest.main produces a simple report telling us how many tests run and how many failed. It also shows us how the test failed.

* **Fixtures:** 
  + **setUp**: This function is run before each test method. Ex: to create a file for us and remember the filename as a member of the TestCase
  + **tearDown**: This function is run after every test method. Ex: to delete the file we created in setUp.

Use of these fixtures allows each test method to start in a stable known state. This is critical to making reproducible tests. The setUp and tearDown function names aren't in line with what PEP 8 prescribes. This is because the unittest module predates those parts of PEP 8 which specify the convention of function names being in lowercase with underscores

* **Assertion Methods:** The testCase class has many assertion methods: assertEqual, assertRaises (To check if the correct type of Exception is thrown), assertTrue(Checks if value passed to it is true), assertFalse. Note that to call these methods we make use of “self.” If the assert checks fail, the test fails with the AssertionError exception.
* **Virtual Environments:**
* A virtual environment is a light-weight, self-contained Python installation that users can create without needing administrator rights on their system.
* With Python 3. 3 or later, we should already have a module called **venv** installed with the standard library.
* If venv is not installed, we can use virtualenv that we can get from the Python Package Index which works very similarly.   
  We can use either venv or virtualenv

python3 -m venv venv3  
here venv is the module name and venv3 is the virtual environment name.

* The tool creates the new directory and populates it with the installation. Once the environment is created, you can activate it by using the activate script in the environment bin directory. On Linux or Mac OS, we have to source the script. On Windows, we simply run it. Once we do this, our prompt will change to remind us that we are in a virtual environment.
* Once activated the Python that will execute when we run Python is from the virtual environment.
* To leave the virtual environment, we use the deactivate command. This will return us to the parent shelf from which the virtual environment was activated.
* **Distributing programs:**
* The **distutils module** allows us to write a simple Python script which knows how to install your Python modules into any Python installation. By convention, this script is called setup.py and it exists at the top of your project structure. This script can then be executed to perform the actual installation.
* Example: Under the palindrome directory, we move our .py code and at the top level we create a new file with following contents

|  |
| --- |
| **from** distutils.core **import** setup  setup(  name=**'palindrome'**,  version=**'1.0'**,  py\_modules=[**'palindrome'**],   *# metadata* author=**'Austin Bingham'**,  author\_email=**'austin@sixty-north.com'**,  description=**'A module for finding palindromic numbers.'**,  license=**'Public domain'**,  keywords=**'example'**, ) |

**Handling Exceptions**

* Exception handling is a mechanism for stopping normal program flow and continuing at some surrounding context or code block.
* The event of interrupting normal flow is called the act of raising an exception.
* In some enclosing context the raised exception must be handled upon which control flow if transferred to the exception handler. If an exception propagates up the call stack to the start of the program, then an unhandled exception will cause the program to terminate.
* And exception object containing information about where and why an exceptional event occurred is transported from the point at which the exception was raised to the exception handler so that the handler can interrogate the exception object and take appropriate action.
* Try- Except construct can be used to handle exception. Both the try and except keywords introduce new blocks. The try block contains code that could raise an exception, and the except block contains the code which performs error handling in the event that an exception is raised.
* Each try block can have multiple corresponding except blocks, which intercept exceptions of different types.
* When multiple exception handlers have same code duplication we can collapsing them into one using the ability of the except statement to accept a tuple of exception types.

def convert(s):

try:

x = int(s)

print("conversion success")

except(ValueError,TypeError):

print("Conversion failed")

* Almost anything that goes wrong with the Python program results in an exception, but some such as IndentationError, SyntaxError, and NameError are the result of programmer errors, which should be identified and corrected during development rather than handled at runtime. The fact that these things are exceptions is mostly useful if you're creating a Python development tool such as a Python IDE, embedding Python itself in a larger system to support application scripting, or designing a plug-in system, which dynamically loads code.
* The pass Statement: It’s a special statement which does precisely nothing. It's a NOOP, and its only purpose is to allow us to construct syntactically permissible blocks which are semantically empty.
* Named Reference to exception Object: To get ahold of the exception object and interrogate it for more details of what went wrong, we can get a named reference to the exception object by tacking an “as” clause onto the end of the except statement.  
  Ex:

In [**108**]: import sys

     ...: def convert(s):

     ...: try:

     ...: x = int(s)

     ...: print("conversion success")

     ...: except(ValueError,TypeError) **as e:**

     ...: print("Conversion failed: {}".format(str(e)),file=sys.stderr)

     ...: return -1

In [**109**]: convert("Sukul")

Conversion failed: invalid literal for int() with base 10: 'Sukul'  
Out[**109**]: -1

Above shows how to print to standard error. First we import sys module and pass sys.stderr as the keyword argument called file to print function.  
Also note that exception objects can be converted to strings using the str constructor.

* Re-raising Exceptions: We can re-raise the exception object we're currently handling simply by using the ‘raise’ statement at the end. Without a parameter, raise simply re-raises the exception that is being currently handled.   
  This can be useful when we want to log some information before raising the exception.
* Exceptions are part of API of the function: Exceptions form an important aspect of the API of a function. Callers of a function need to know which exceptions to expect under various conditions so that they can ensure appropriate exception handlers are in place. In fact we should also modify the docstring to make it plain which exception type will be raised and under what circumstances. The exceptions which are raised are as much a part of a function's specification as the arguments it accepts, and as such must be implemented and documented appropriately.
* Standard Python Exceptions: Python provides us with several standard exception types to signal common errors. If a function parameter is supplied with an illegal value, it is customary to raise a ValueError. We can do this by using the raise keyword with a newly created exception object, which we can create by calling the ValueError constructor. The ValueError constructor accepts an error message.

In [**114**]: import sys

In [**115**]: def cubeme(x):

     ...: if x < 0:

     ...: raise ValueError("Dont want to work with negative numbers")

     ...: return x \* x \* x

In [**116**]: try:

     ...: cubeme(1)

     ...: cubeme(97)

     ...: cubeme(-1)

     ...: except ValueError as v:

     ...: print(v,file=sys.stderr)

Dont want to work with negative numbers

* There are a handful of common exception types in Python, and usually when we need to raise an exception in our own code one of the built-in types is a good choice.
  + IndexError is raised when an integer index is out of range. You can see this when you index pass the end of a list.
  + ValueError is raised when the object is of the right type, but contains an inappropriate value.
  + KeyError is raised when a look-up in a mapping fails
* Do not guard against Type Errors: doing so runs against the grain of dynamic typing in Python and limits the reuse potential of the code that we write.  
  If a function works with a type, even one you couldn't have known about when you designed your function, then that's all to the good. If not, execution will probably result in a TypeError anyway.
* EAFP vs LYBL: Only two approaches to dealing with a program operation that might fail.
  + The first approach is to check that all the preconditions for a failure-prone operation are met in advance of attempting the operation.
  + The second approach is to perform the operation but be prepared to deal with the consequences if it doesn't work out.

In Python culture, these two philosophies are known as

* + Look Before You Leap, LBYL, and
  + It's Easier to Ask Forgiveness than Permission, EAFP

Python is strongly in favor of EAFP because it puts primary logic for the happy path in its most readable form with deviations from the normal flow handled separately rather than interspersed with the main flow.

Problem with LYBL is that we need to think of all the preemptive checks before performing the risky operation.Also, there is a chance of a race condition (atomicity issue). Things might change between the check and the actual risky operation.   
Ex: we may check for file existence with a pre-emptive test, however file may get deleted by another process between the check and actual use of the file in our code.

With Pythonic EAFP approach, we simply attempt the operation without checks in advance, but we have an exception handler in place to deal with any problems. We don't even need to know in a lot of detail exactly what might go wrong.

EAFP is standard in Python, and that philosophy is enabled by exceptions. Without exceptions, that is using error codes instead, you are forced to include error handling directly in the main flow of the logic. Since exceptions interrupt the main flow, they allow you to handle exceptional cases non-locally. Exceptions coupled with EAFP are also superior because unlike error codes exceptions cannot be easily ignored. By default, exceptions have a big effect whereas error codes are silent by default.

* Try-finally: Code in the finally-block is executed whether execution leaves the try-block normally by reaching the end of the block or exceptionally by an exception being raised. So finally block can be used to perform a cleanup action irrespective of whether an operation succeeds.
* **Summary:**
* The raising of an exception interrupts normal program flow and transfers control to an exception handler.
* Exception handlers are defined using the try…except construct. Try blocks define a context in which exceptions can be detected. Corresponding except blocks define handlers for specific types of exceptions.
* Except blocks can capture an exception object, which is often of a standard type such as a ValueError, KeyError, or IndexError.
* Programmer errors such as indentation error and syntax error should not normally be handled.
* Exceptional conditions can be signaled using the raise keyword, which accepts a single parameter of an exception object. Raise without an argument with an except block re-raises the exception which is currently being processed.
* We tend to not to routinely check for TypeErrors. To do so would negate the flexibility afforded to us by Python's dynamic type system.
* Exception objects can be converted to strings using the str() constructor for the purposes of printing message payloads.
* The exceptions thrown by a function form part of its API and should be appropriately documented.
* When raising exceptions, prefer to use the most appropriate built-in exception type.
* Cleanup and restorative actions can be performed using the try…finally construct, which may optionally be used in conjunction with except blocks.
* Output of the print() function can be directed to standard error using the optional file argument

**Modularity**

* Modularity is an important property for any software systems as it gives us the power to make self-contained, reusable pieces, which can be combined in new ways to solve different problems.

In Python the most fine-grained modularization facility is the definition of reusable functions.

* Collections of related functions are typically grouped into source code files called modules.   
  Modules can be used from other modules, so long as we take care not to introduce a circular dependency.

Python 3 runtime expects source code files to be