Python Learning Materials

Release 0.0.1

Simon Kerr

Nov 28, 2021

CONTENTS:

1 Data Structures: Sets 1

2 Data Structures: Lists 13

3 Data Structures: Dictionaries 17

4 Data Structures: Linked List 19

5 Descriptor Protocol 23

6 Virtual Subclassing 35

7 Iterator Protocol 39

8 Shallow & Deep Cloning 43

9 Context Managers 45

10 Collections: namedtuple 51

11 Positional and Keyword arguments 57

12 Regular Expressions 59

13 Concurrency: Threading 65

14 Pass By Assignment 67

15 String Methods 69

16 Indices and tables 73

CHAPTER

ONE

DATA STRUCTURES: SETS

Python sets are:

- Unordered, non indexable, distinct immutable (hashable) elements.
- Come in two flavours `set` (mutable) & `frozenset` (immutable).
- Sets cannot contain other sets as they are not hashable, they can contain `frozenset` instances.
- Sets offer quick membership testing *in* and removing duplicates from other collections.
- Sets support a whole host of mathematical operations (set theory) such as *union* & *intersection* etc.

1.1 Sets: Instantiation

Python sets can be created in a number of different ways:

```
# simple set() constructor:
empty_set = set()
# set from any iterable:
set_from_iter = set(range(1, 10))
# set using the braces syntax:
set_braces = {"one", "two", "three"}
# set using a set comprehension:
set_comp = {n for n in range(20) if n % 2 == 0}
```

Care is advised when using the curly braces, often when trying to create an empty set, subtle bugs can be introduced as python treats *{}* as a `dict`.

```
type({}) # dict
```

Sets themselves are not immutable and thus, not hashable so this means that sets cannot store sets within themselves, another build in data structure is the `frozenset` which can be used as elements inside sets themselves:

```
s = {{1,2}, {3,4}}
# TypeError: un-hashable type: `set`
frozen = frozenset({1,2})
>>> frozenset({1,2})
```

More can be found about frozenset later in the documentation.

We touched briefly on sets being unable to add non hashable elements, in python both `list` and `dict` are also *mutable* and thus, neither can be added to a normal `set`:

```
s = {[1,2,3]}
# TypeError: un-hashable type: `list`
s = {dict(a=1)}
# TypeError: un-hashable type: `dict`
```

However, because the *set()* class permits building a set from an iterable and both list and dictionary are iterable (dict over keys by default), then populating a set from both of the collections is possible:

```
s = set([1,2,3,4,5])
# {1, 2, 3, 4, 5}
s = set(dict(a=1, b=2, c=3))
# {'a', 'b', 'c'}
```

1.2 Sets: Distinction

We mentioned previously that sets must contain hashable elements only, this is because similarly to dictionary keys, sets use the hash value of the object it is attempting to store internally. This is why *in* checks are extremely fast in sets, they are backed by a hash table. In order to be able to store your custom objects in a *set* (or alternatively use them for dictionary *keys*) you can implement two magic methods, *__hash__* and *__eq__* respectively.

By default, user defined objects have the following in python:

- an implementation of *__hash__*.
- an implementation of *__eq__* which results in no two instances being equal.

```
class Example:
    def __init__(self, x: int) -> None:
        self.x = x

e = Example(100)
e2 = Example(100)
hash(e)  # 108032011057
hash(e2)  # 108032014237 (different)
e == e2  # False
{e, e2}  # {<__main__.Example at 0x192735ab310>, <__main__.Example at␣
↪0x192735b79d0>}
```

By default this permits us to store instances of *Example* in a set by default as highlighted above. In order to use our own user defined objects in sets effectively, we should implement both the dunder *__hash__* and *__eq__* methods to consider two instances of `Example` equal.

```
from __future__ import annotations # __eq__ `other` type hint of the class␣
↪itself

class ImprovedExample:
    def __init__(self, x: int) -> None:
        self.x = x

    def __hash__(self) -> int:
        return hash(self.x)

    def __eq__(self, other: ImprovedExample) -> bool:
```

(continues on next page)

(continued from previous page)

```
        # note: returning `NotImplemented` here tells python to try the␣
↪reflected operation on `other`.
        if not isinstance(other, type(self)): return NotImplemented
        return self.x == other.x
```

Now we are able to store instances of *ImprovedExample* in both sets and in dictionaries as keys:

```
one, two, three = ImprovedExample(100), ImprovedExample(200),␣
↪ImprovedExample(100)
{one, two, three}  # one == three & hash(one) == hash(three) thus only 2 are␣
↪stored (distinct)
"""
{<__main__.ImprovedExample at 0x1927465c490>,
<__main__.ImprovedExample at 0x1927465c880>}
"""
```

** If a class does not implement dunder __eq__, it should never implement dunder __hash__. **

1.3 Sets: Method resolution order

Pythons `collections.abc.Set` MRO is described below:

```
from collections.abc import Set

Set.mro()
"""
(collections.abc.Set,
 collections.abc.Collection,
 collections.abc.Sized,
 collections.abc.Iterable,
 collections.abc.Container,
 object)

Set inherits from `Collection`
`Collection` inherits from `Sized` which provides len(set).
`Collection` inherits from `Iterable` which allows sets to be iterated over.
`Collection inherits from `Container` which allows sets to perform `in` checks␣
↪via `__contains__`.
and lastly, everything inherits from `object`.

`Set` inherits a lot of additional capabilities through its mixin methods:
    * __le__
    * __lt__
    * __eq__
    * __new__
    * __gt__
    * __ge__
    * __and__
    * __or__
    * __sub__
    * __xor__
```

(continues on next page)

(continued from previous page)

```
    * isdisjoint()

A lot of these mixin methods will be discussed later in depth and how objects
can slot right into pythons data model and be considered pythonic.
"""
```

1.4 Sets: Operations I - Basics

Many operations supported on other data structures do not make logical sense for sets, however sets themselves offer a very robust set of operations to align them nicely with sets in mathematics. Some functionality not supported by sets are (that of sequences) like slicing a set, or finding the *index* of a given *element* within the set.

```
s = {1,2,3,4,5,6}
s[1:3]
# TypeError: set object is not subscriptable

s = {5,4,3,2,1}
s.index(4)
# AttributeError: set object has no attribute: index
```

In order to fully understand the power of sets, we need to understand the distinct differences between three things:

- object methods
- object operations
- augmented operations (we will touch on this later on).

Almost all the functionality of python sets can be performed in two main ways. Via set instance methods, for example:

```
s = {1,2,3}
s.union({3,4,5})  # Method invocation -> {1,2,3,4,5}
```

Alternatively, as we touched on earlier, through various mixin methods implemented on `Set`, the following is also supported:

```
one = {1,2,3}
two = {3,4,5}
one | two  # Operation invocation -> {1,2,3,4,5}
```

Notice how the duplicate *3* entry in both cases is deduped, a simple trait of sets (to remove duplicates). Both examples above result in (almost) the same thing happening, functionally it is the same, however operations tend to be slightly faster, this is outlined below:

```
import dis
one = {1,2,3}
two = {3,4,5}
dis.dis("one.update(two)")
"""
1      0 LOAD_NAME                0 (one)
       2 LOAD_METHOD              1 (update)
       4 LOAD_NAME                2 (two)
       6 CALL_METHOD              1
```

(continues on next page)

(continued from previous page)

```
      8 RETURN_VALUE
"""

dis.dis("one | two")
"""
  1   0 LOAD_NAME                0 (one)
      2 LOAD_NAME                1 (two)
      4 BINARY_OR
      6 RETURN_VALUE
"""
```

In the above example we can see two additional *bytecode instructions*: *LOAD_METHOD* and *CALL_METHOD*. For a real world bench mark, lets perform the same task (getting the union of the above two sets) to see the difference (20 million times).

```
import timeit
timeit.timeit("one.union(two)", setup="one={1,2,3}; two={3,4,5}", number=20_
↪000_000)
# 4.2465937000000005 (4.2 seconds)
timeit.timeit("one | two", setup="one={1,2,3}; two={3,4,5}", number=20_000_000)
# 3.168324699999971 (3.1 seconds)
```

While negligible it is important to understand that operator approaches are often faster. There are however a few subtle differences / caveats to be aware of.

- when using the method based approach, e.g *union()* any *iterable* can be provided and python will handle it
- when using the operator based approach, e.g | all objects must be of type: *set*.

```
s = {1,2,3}
s.union([2,4,6,8])
# {1, 2, 3, 4, 6, 8}
s | [2,4,6,8]
# unsupported operand type(s) for |: `set` and `list`.
```

By default, both the methods and basic operators return a new *set* instance. We briefly spoke about *augmented operators*, these can be used to modify set *s* in-place, more on that later.

1.5 Sets: Operations II - Intermediate

We touched briefly on the *union()* method of sets, now we will outline all the available functionality including appropriate *venn* diagrams for various operations.

```
methods = tuple(attr for attr in dir(set()) if "__" not in attr)
"""
('add',
 'clear',
 'copy',
 'difference',
 'difference_update',
 'discard',
```

(continues on next page)

(continued from previous page)

```
 'intersection',
 'intersection_update',
 'isdisjoint',
 'issubset',
 'issuperset',
 'pop',
 'remove',
 'symmetric_difference',
 'symmetric_difference_update',
 'union',
 'update')
"""
```

Method: `add(elem)`:

- **Description**: adds a single element (`elem`) into the set, if `elem` is already a member, this does nothing.
- **Operator equivalent**: Not Applicable

```
s = set()
s.add(100)
# {100}
```

Method: `clear()`:

- **Description**: Removes all elements from the set
- **Operator equivalent**: Not Applicable

```
s = set(range(10))
# {1,2,3,4,5,6,7,8,9}
s.clear()
# set()
```

Method: `copy()`:

- **Description**: Creates a `shallow` copy of the set
- **Operator equivalent**: Not Applicable

```
s = {1,2,3}
s2 = s.copy()
s == s2  # True
s is s2  # False
s.add(4)
# s {1,2,3,4}
# s2 {1,2,3}
```

Method: `difference(*other_sets)`:

- **Description**: Return a `new` set of the difference of this set and `*other_sets`.
- **Operator Equivalent**: -
- **Notes**: Difference is calculated left <- to right -> when multiple `*other_sets` are provided.
- **Notes**: Difference is basically, items in `x` but not in `y` or `z` -> x.difference(y,z) : x | y | z
- **Notes**: As always, operator invocations must be of type: `Set`, `difference()` will work with iterables.

```
x = {1,2,3}
y = {3,4,5}
x.difference(y)
# {1,2}
```

When we compute the difference between one or multiple sets, we are working from left to right and basically subtracting any elements from the next to be checked set from the set that we previously built, here is a documented example using 3 sets:

```
one = {1,2,3}
two = {3,4,5}
three = {2,3}

# Generate three sets, two contains 1 number also in one, three contains two␣
↪numbers in one

# Check one against two using method and operator, both are equivalent except␣
↪for speed.
one.difference(two)
# {1,2}
one - two
# {1,2}

# Why? because `3` is in one and two, so we discard it, left to right is␣
↪important here:

two.difference(one)
# {4,5}
two - one
# {4,5}

# Now when we also check the difference when `three` gets involved:
one.difference(two, three)
# {1}
one - two - three
# {1}
```

Python implements this behaviour at the operator level by implementing __sub__:

```
def __sub__(self, other):
    if not isinstance(other, Set):
        if not isinstance(other, Iterable):
            return NotImplemented
        other = self._from_iterable(other)
    return self._from_iterable(value for value in self if value not in other)
# from_iterable is just a class method to build a set instance from any␣
↪iterable.
```

As we touched on previously, remember when using operator syntax, sets **must** be passed:

```
s = {1,3,5}
s.difference([3], [5])
# {1}
```

(continues on next page)

(continued from previous page)

```
s - [3] - [5]
# TypeError: unsupported operand type(s) for -: 'set' and 'list'
```

Lastly, we can observe when multiple sets are compared for difference, python operates from left <- to right -> performing a `BINARY_SUBTRACT` bytecode instruction at each step:

```
import dis
x = {1,2,3}
y = {3,4}
z = {2}
dis.dis("x - y")
"""
  1   0 LOAD_NAME                0 (x)
      2 LOAD_NAME                1 (y)
      4 BINARY_SUBTRACT
      6 RETURN_VALUE
"""

dis.dis("x - y - z")
"""
  1   0 LOAD_NAME                0 (x)
      2 LOAD_NAME                1 (y)
      4 BINARY_SUBTRACT
      6 LOAD_NAME                2 (z)
      8 BINARY_SUBTRACT
     10 RETURN_VALUE
"""
```

Method: `difference_update(*other_sets)`:

- **Description**: Removes all elements from `other_sets` from this one
- **Operator Equivalent**: `-=`
- **Notes**: Is an `augmented` assignment, modifies the set `in-place`.

`difference_update()` is pretty much the same as `difference` with one core difference, this is an equlvalent `augmented operator`. Below is the bytecode instructions to demonstrate `difference()` vs `difference_update`:

```
import dis
x = {1,2,3}
y = {2}
dis.dis("x.difference(y)")
"""
  1   0 LOAD_NAME                0 (x)
      2 LOAD_METHOD              1 (difference)
      4 LOAD_NAME                2 (y)
      6 CALL_METHOD              1
      8 RETURN_VALUE
"""

dis.dis("x.difference_update(y)")
"""
  1   0 LOAD_NAME                0 (x)
```

(continues on next page)

(continued from previous page)

```
      2 LOAD_METHOD              1 (difference_update)
      4 LOAD_NAME                2 (y)
      6 CALL_METHOD              1
      8 RETURN_VALUE
"""
```

As shown above, the subtle difference only outlines the `difference_update` *LOAD_METHOD* in the latter, however if we inspect the byte code when using the augmented operator:

```
import dis
x = {1,2,3}
y = {2}

dis.dis("x - y")
"""
  1   0 LOAD_NAME                0 (x)
      2 LOAD_NAME                1 (y)
      4 BINARY_SUBTRACT
      6 RETURN_VALUE
"""

dis.dis("x -= y")
"""
1 0 LOAD_NAME                0 (x)
  2 LOAD_NAME                1 (y)
  4 INPLACE_SUBTRACT
  6 STORE_NAME               0 (x)
  8 LOAD_CONST               0 (None)
 10 RETURN_VALUE
"""
```

We observe the `INPLACE_SUBTRACT` instruction. Similarly to `difference()` any number of `iterables` can be passed into the method as arguments, however when using the augmented operator equivalent, only types of `set` may be provided. Another very important limitation is that `augmented operators` can **NOT** be chained together like `x - y - z` can.

```
x = {1,2,3,4,5}
y = {4,5}
z = {3}

x.difference_update(y,z)
print(x)  # {1,2}

"""
Because this is all in-place, here is roughly what happens:

x starts life as a new set of: {1,2,3,4,5}
x.difference_update(y) occurs, resulting in x modified in place to remove {4,5}
x.difference_update(z) occurs, resulting in x modified in place to remove {3}
x is now the same reference, with it's values modified: {1,2}
"""
```

(continues on next page)

(continued from previous page)

```
# Augmented operators are not allowed to be used on multiple targets
x -= y -= z
# SyntaxError: invalid syntax

# Augmented operators like normal operators, must be of type: Set
x = {1,2,3}
y = [3,4,5]
x -= y
# TypeError: unsupported operand type(s) for -=: 'set' and 'list'
```

Method: `discard(elem)`: Description: Attempt to remove `elem` from the set, if `elem` is not in the set, do nothing **Operator Equivalent**: Not Applicable **Notes**: Similar to `remove()` however does **not** raise a `KeyError` **Notes**: Returns `None`.

```
x = {1,2,3,4,5}
x.remove(6)
type(x)
# `NoneType`
```

Method: `intersection(*other_sets)`: Description: Computes the items all sets have `in` `common`. **Operator Equivalent**: & **Notes**: Is not an *augmented* in place operation, creates a new `set()` of the results

Like all the other set methods and operations, `intersection()` has accept an assortment of iterables when using the method format and when using the & operator, types must be `Set`. Creating the intersection of multiple sets moves from left <- to right -> evaluating each one against the next and retaining elements which are common in both:

```
x = {1,2,3}
y = {4,5,6}
x.intersection(y)
# x = {}
# There are no comment elements in X that also are in Y

# Let's find some common elements
x = {1,2,3}
y = {3,6,5}
z = {3,6,7}
x.intersection(y,z)
# {3} - Why? x & y results in: {3}, y & z results in: {3}
# Notice how `6` is not considered common here, because `x & y` creates only {3}␣
↪before & z is compared.

# The same example, using operators:
x = {1,2,3}
y = {3,6,5}
z = {3,6,7}
new = x & y & z
print(new)
# {3}

# This is explained easily by inspecting the bytecode, you can see X & Y is␣
↪compared, then the new set & z
import dis
dis.dis("x & y & z")
```

(continues on next page)

(continued from previous page)

```
"""
dis.dis("x & y & z")
  1           0 LOAD_NAME                0 (x)
              2 LOAD_NAME                1 (y)
              4 BINARY_AND
              6 LOAD_NAME                2 (z)
              8 BINARY_AND
             10 RETURN_VALUE
"""
```

As we previously mentioned for `difference()`, when dealing with an `operator` approach, types of `Set` will be enforced by python, `intersection(*others)` can be any `iterables.`:

```
x = {1,2,3,4,5}
y = [3,4,5]
x & y # TypeError: unsupported operand type(s) for &: 'set' and 'list'
x.intersection(y)  # {3,4,5}
```

Below is a simple venn diagram that demonstrates the `intersection` of the following python code:

```
x = {1,2,3,4}
y = {3,4,5,6}
# 2 items unique to x (1,2)
# 2 items common in x & y (3,4)
# 2 items unique to y (5,6)
```

Method: `intersection_update(*other_sets)`: Description: Computes the items all sets have `in common` and modifies x `in-place`. **Operator Equivalent**: `&=` **Notes**: `x.intersection_update(y,z) updates ``x` in-place, it does **not** create a set.

Another `augmented` operator equivalent method, that updates the set with items in the other iterables (or sets if using the augmented operator approach).

Updating `x` in place:

```
x = {1,2,3}
y = {2,3,4}
x &= y
```

1.6 Sets: Operations III - Advanced

...

1.7 Sets: Frozensets

...

1.8 Sets: Miscellaneous

...

1.9 Sets: Summary

- `frozenset` is immutable, `set` is mutable.
- `set` contain unordered, non indexable distinct hashable immutable elements.
- Using empty *set comprehension* syntax will actually generate a *dictionary*.
- create `set` using *set()*, *{1,2,3}* or *{n for n in range(10) if n % 2 == 0}*.
- create `frozenset` using the *frozenset()* callable.
- user defined objects can be stored in sets by default, but are never considered equal.
- to add user defined objects to sets, implement *__hash__* and *__eq__*.
- `Set` inherits from `collections.abc.Collection` which in turns inherits from *Sized*, *Iterable*, *Container*.
- `Set` permits many of its functionality through both method calls and operators.
- `Set` operator usage tends to be slightly faster due to not having to load & call a method.
- Augmented operators cannot be chained like normal operators: `x -= y -= z` is not permitted like `x - y - z`.
- `x.difference(*other)` removes elements in other, from x creating a new `Set`.
- `x.difference_update(*other)` removes element in other, from x `in-place`.
- `x.discard(y)` removes `y` from the set if it exists, if it does not it quietly does nothing.

CHAPTER

TWO

DATA STRUCTURES: LISTS

2.1 Lists: Introduction

Python lists are mutable `sequences` that store ordered heterogenerous data. There no concept of duplicates in a list and both `hashable` and `non hashable` elements can be stored.

2.2 Lists: Instantiation

There are three main ways to create a list in python:

- the *list()* built in.
- the `[..., ...]` square bracket syntax.
- list comprehensions, `[char.upper() for char in "foobar"]` -> *['F', 'O', 'O', 'B', 'A', 'R']*

When creating a simple list, the fastest way to build one is using the *[]* syntax over the *list()* builtin. This is because there is one less function lookup/call as demonstrated by the following bytecode, a brief benchmark of the *list()* vs *[]* is outlined below:

```
import dis

dis.dis("[]")
"""
1   0 LOAD_NAME                0 (list)
    2 LOAD_NAME                1 (items)
    4 CALL_FUNCTION            1
    6 RETURN_VALUE
"""

dis.dis("list()")
"""
1     0 LOAD_NAME                0 (list)
      2 LOAD_NAME                1 (items)
      4 CALL_FUNCTION            1
      6 RETURN_VALUE
"""

import timeit
timeit.timeit("[]", number=10_000_000)
# 0.15199436400143895
```

(continues on next page)

(continued from previous page)

```
timeit.timeit("list()", number=10_000_000)
# 0.5499999200997263
```

The reality / performance is often a non factor, but to be as python as possible, avoid using *list()* when *[]* is an option. List comprehensions are another beast altogether which we will discuss later, but for now her is a simple example:

```
x = range(1, 11)
odd_nums = [n for n in x if n % 2 == 0]
print(odd_nums)
# [2,4,6,8,10]
```

2.3 Lists: MRO & Collections

As we previously touched on, *list* types in python are *MutableSequences*, what does this actually mean? Firstly let's derive all the subclassing (and virtual subclassing that) is occurring for the python list type. In order to achieve that, we can use this handy function:

```
import collections.abc
import inspect

def build_hierarchy(col):
    abcapi = vars(collections.abc).items()
    return [v for k,v in abcapi if inspect.isclass(v) and issubclass(col, v)]

build_hierarchy(list)
"""
[collections.abc.Iterable,
 collections.abc.Reversible,
 collections.abc.Sized,
 collections.abc.Container,
 collections.abc.Collection,
 collections.abc.Sequence,
 collections.abc.MutableSequence]
"""
```

We can break down the list inheritance hierarchy (explicit and virtual subclasses) outlined above. As we mentioned, python lists are *Mutable* and *Sequences*, let's understand what each of these classes offer the list class:

2.4 Lists: Iterable

Python lists inherit behaviour and are iterable, via the `collections.abc.Iterable` abstract base class. This class requires an abstractmethod implementation for *__iter__*. This is taken care for by the `collections.abc.Iterator` inheritance for python lists, which simply returns *self*.

2.5 Lists: Iterator:

Another part of the `iterator protocol`. `collections.abc.Iterator` implements a default *__iter__* return itself and enforces that subclass have an implementation for `__next__`.

2.6 Lists: Reversible:

The `collections.abc.reversible` abstract base class exposes a `__reversed__` dunder method and itself is an instance of `Iterable`.

2.7 Lists: append():

The `.append()` method of a list takes a single *object* and adds it to the *tail* of the list. If the *object* is iterable, it is **not** unpacked, instead a single object is added, adding a tuple to a list via `append((1,2,3))` will have a list containing the tuple at the tail.

```
items = [1,2]
items.append((3,5,7))
items  # [1,2, (3,5,7)]
```

2.8 Lists: clear():

Removes all elements from the list.

```
items = [1,2,3,4,5]
items.clear()
items # []
```

2.9 Lists: copy():

Creates a `shallow` copy of the list

CHAPTER

THREE

DATA STRUCTURES: DICTIONARIES

3.1 Dictionaries: Introduction

…

3.2 Dictionaries: MRO Hierarchy

…

3.3 Dictionaries: Methods

…

3.4 Dictionaries: View Objects

…

CHAPTER

FOUR

DATA STRUCTURES: LINKED LIST

4.1 Linked Lists: Attempt

```
from __future__ import annotations
from typing import Optional
from typing import Iterable


class Node:
    def __init__(self, value: int, ref: Optional[Node] = None) -> None:
        self.value = value
        self.ref = ref

    def __repr__(self) -> str:
        return f"<Node(value={self.value}, ref={self.ref})>"

    def __str__(self) -> str:
        return str(self.value)


class SingleLinkedList:
    def __init__(self, iterable: Iterable[int] = None) -> None:
        self.head: Optional[Node] = None
        if iterable:
            self.extend(iterable)

    def __str__(self) -> str:
        # By nature a linked list is not aware of all of its nodes, so we must␣
↪iterate
        # all node references.
        start = self.head
        s = ""
        while start:
            s += "-> " + str(start)
            start = start.ref
        return "[" + s + "]"

    def append(self, value: int) -> None:
        """
        Register a new node into the linked list, this node is
```

(continues on next page)

(continued from previous page)

```
        appended to the tail of the linked list.
        :param value: The integer to add
        :return: `None`
        """
        if self.head is None:
            self.head = Node(value)
            return

        current = self.head
        while current.ref:
            current = current.ref
        current.ref = Node(value)

    def insert_left(self, value: int) -> None:
        """
        Insert the new value to the left of the linked list, this
        will push the current head forward once and register a
        new node with `value` at the current head.
        :param value: integer to insert at the head of the linked list.
        :return: None
        """
        if self.head is None:
            self.head = Node(value)
            return
        new = Node(value, ref=self.head)
        self.head = new

    def extend(self, elements: Optional[Iterable[int]] = None):
        """
        Extends the linked list by appending elements from the iterable.
        :param elements:
        :return:
        """
        for element in elements:
            self.append(element)

    def clear(self) -> None:
        """
        Erases all references from head in the linked list.
        :return: None
        """
        self.head = None

    def reverse(self) -> None:
        """
        Reverses the linked list in place.
        :return: None
        """
        previous, current = None, self.head
        while current:
            after = current.ref  # store the next node temporarily
            current.ref = previous  # set the next node to the previous one
```

(continues on next page)

(continued from previous page)

```
            previous = current  #
            current = after  #
        self.head = previous

#  10 -- 5 -- 8 --
#  5 -- 8 -- 10 --
#

# ----------------------------
linked = SingleLinkedList((5, 10, 15))
linked.append(100)
linked.append(250)
linked.append(500)
linked.insert_left(55)
print(linked)
linked.reverse()
print(linked)
linked.clear()
print(linked)
```

CHAPTER

FIVE

DESCRIPTOR PROTOCOL

5.1 Descriptors: Intro

Python descriptors allow objects to customise:

- Attribute lookup
- Attribute storage
- Attribute deletion

If you are new to descriptors, chances are you've already been using them as they are responsible for underpinning core python built in functionality. Some things which are powered by descriptors and we will discuss later are:

- *@property*
- *@staticmethod*
- *@classmethod*
- Creating *bound* methods from *function* types
- Pythons *super()*

5.2 Descriptors: A Trivial Example

To get a feel for descriptors, we will create a rather trivial example, a descriptor that reverses a simple string value, which importantly is computed each time the value is accessed. The important thing to understand at this point are `descriptors` are their own class and they live as `class` attributes in other classes, let's see it action:

```
class UpperAccess:
    """
    This is a simple descriptor, at this point we are only
    dealing with non data descriptors, so we will only
    implement __get__.  More on this later, keep reading!
    but let's keep things simple for now.
    """
    def __init__(self, word: str) -> None:
        self.word = word

    def __get__(self, obj, objtype = None):
        return self.word.upper()
```

(continues on next page)

(continued from previous page)

```
class UsingUpperAccess:
    """
    This is a simple class that uses a ``MyDescriptor`` instance.
    Important note: The descriptor must live as a CLASS attribute
    in another class.
    """
    word = UpperAccess("foo")

clazz = UsingUpperAccess()
print(clazz.word) # 'FOO'
```

The important thing to remember here is, `descriptors` live as class attributes in other classes, when accessing the `clazz.word` python first has a look in the *clazz.__dict__* <instance dict> and then finds the descriptor in the *type(clazz).__dict__* <class dict>. The uppercased value does `NOT` live in either the *instance* or *class* dict, it is computed on demand!

5.3 Descriptors: Compute on demand

To better explain the concept of value(s) being computed on demand, we will build a descriptor instance that based on a working directory, can list the contents. For the sake of this article, we will use a *tree* structure like so:

```
example
├── colors
│   ├── blue.txt
│   └── red.txt
└── numbers
    ├── one.txt
    └── two.txt
```

In a nutshell, we have two subdirectories, *colors* and *numbers*, let's write a descriptor that can read the contents of those files, dynamically:

```
import os

class ContentsOf:
    def __get__(self, obj, objtype=None):
        # it is the obj reference as a way back to the declaring class
        return os.listdir(obj.dirname)

class RootDirectory:
    files = ContentsOf()

    def __init__(self, dirname):
        self.dirname = "/tmp/example/" + dirname

colors = RootDirectory("colors")
numbers = RootDirectory("numbers")
colors.files # ["red.txt", "blue.txt"]
numbers.files  # ["one.txt", "two.txt"]
```

Now that we understand a little better, how descriptors compute value(s) on demand, this example also exposes us to a slightly deeper look into part of the ``descriptor protocol`.

5.4 Descriptors: __get__

Part of the `descriptor protocol`, dunder `__get__` is responsible for handling the _lookup_ part of the descriptor outlined in our first paragraph. The secret to understanding how `__get__` works is to understand this is `class level access`.

```
class Descriptor:
    def __get__(self, obj, objtype = None):
        """
        :param self:
            This instance of ``Descriptor``.

        :param obj:
            The instance of the class in which the descriptor was instantiated

        :param objtype:
            The (optional) own class `type` e.g `obj.__class__`

        __get__() should return the ``computed`` value, or raise an
↪``AttributeError``
        """
        ...
```

By default pythons *__get_attribute__* will provide both arguments to the *__get__* call, here is an example of the types and value(s) accessible via *__get__()*:

```
class D:

    def __get__(self, obj, objtype=None) -> value
        print(locals())
        # should really return here :)


class Instance:
    d = D()

i = Instance()
i.d
# {'self': <__main__.D object at 0x7f489e6f8340>,
# 'obj': <__main__.Instance object at 0x7f489e8078b0>,
# 'objtype': <class '__main__.Instance'>}
# self -> the instance of `D`
# obj -> the instance of `Instance`
# objtype -> the class of instance `i.__class__`)`
```

5.5 Descriptors: Managed Attributes

As we touched on originally in the form of pythons built in *@property*, a great example use case for descriptors is managing access to instance data. The descriptor is assigned to a public attribute in the `class` dictionary (again not the actual value, it's computed on demand) and the actual data is stored as a private attribute in the `instance` dictionary. descriptors *__get__()* and *__set__()* are called for public access. Up until now we have only covered the *__get__()* part of the protocol, let's dive into what are known as *Data Descriptors* (those which do not **only** implement *__get__()*, the former are known as *Non Data Descriptors*. We will create a guarded variable that when accessed audits its access through python logging:

```
import logging
import random
logging.basicConfig(level=logging.INFO)  # Simple root logger to info

class LoggedAccess:
    def __get__(self, obj, objtype=None):
        private = obj._secure
        logging.info(f"Accessed `secure`, resulted in: {private}")
        return private

    def __set__(self, obj, value) -> None:
        # This is new to us, more on that after!
        logging.info(f"Setting `secure` to: {value}")
        obj._secure = value

class Klazz:
    secure = LoggedAccess()  # Class dictionary, public attribute

    def __init__(self, secure):
        self.secure = secure

    def shuffle_secure(self):
        # shuffles the letters in our secure word!
        # Importantly, calls both __get__ & __set__ of our descriptor.
        new = list(self.secure)
        random.shuffle(new)
        self.secure = "".join(new)

k = Klazz("nice")
# INFO:root:Setting `secure` to: nice
k.shuffle_secure()
# INFO:root:Accessed `secure`, resulted in: nice
# INFO:root:Setting `secure` to: inec
```

Looking closer at our example, we have derive a few things:

- All access to the managed access *secure* is logged
- *k* instance dictionary only holds the *_secure* attribute: *vars(k) -> {'_secure': 'inec'}*
- *Klazz* class dictionary holds a instance of *LoggedAccess*: *vars(Klazz) -> `..., 'secure', ...*

One glaring problem with this is that our *_secure* attribute is hardwired and tightly coupled into the *LoggedAccess* descriptor, this creates a bottleneck where each instance can only have a single logged / managed attribute and the name is completely unchangable. We will discuss a solution to that later but for now, let's understand the second piece

of the descriptor procotol, *__set__*.

5.6 Descriptors: __set__

Part of the `descriptor protocol`, dunder `__set__` is responsible for handling the *storage*. Descriptors implementing *__set__()* are automatically considered *Data Descriptors* and that implicitly changes some of the attribute access flow, we will discuss that later. Even if a __set__ implementation has an exception raising place holder, it is enough to qualify as a `Data Descriptor`.

```
class Descriptor:
    def __set__(self, obj, value) -> None:
        """
        Called to update an attribute on the instance of the owner class
        Note: Adding a __set__() to a descriptor transforms it into a data␣
↪descriptor
        which has impacts in terms of the call flow, more on that later.

        In typical setter fashion, __set__ should return `None`.
        """
        ...
```

Part of the `descriptor protocol`, dunder `__get__` is responsible for handling the _lookup_ part of the descriptor outlined in our first paragraph. The secret to understanding how `__get__` works is to understand this is `class level access`.

5.7 Descriptors: Customising names

When a class uses `descriptors`, it can inform the descriptor of which variable name was used, this can help us circumvent the issue we exposed during our managed attribute example. This is achieved through the dunder *__set_name__* method, below is an example where multiple variables can become managed attributes without lots of coupling in the Descriptor implementation itself:

```
import logging
logging.basicConfig(level=logging.INFO)  # root logger configured to info

class LoggedAttr:
    def __set_name__(self, owner, name):
        # This is new! it holds the key to decoupling multiple managed␣
↪attributes
        # Let's store a public/private names on the actual Descriptor instance
        logging.info("__set_name__ called!", locals())
        self.public = name
        self.private = "_" + name

    def __get__(self, obj, objtype = None):
        private = getattr(obj, self.private)
        logging.info(f"Retrieving: {self.public} with value: {private}")
        return private

    def __set__(self, obj, value) -> None:
```

(continues on next page)

(continued from previous page)

```
        logging.info(f"Updating: {self.public} to: {value}")
        setattr(obj, self.private, value)

class Car:
    wheels = LoggedAttr()
    color = LoggedAttr()

    def __init__(self, wheels, color):
        self.wheels = wheels
        self.color = color

    def remodel(self):
        self.wheels = 3
        self.color = "blue"

c = Car(4, "red")
# INFO:root:Updating: wheels to: 4
# INFO:root:Updating: color to: red
c.remodel()
# INFO:root:Updating: wheels to: 3
# INFO:root:Updating: color to: blue
```

As you can see, the same `LoggedAttr` class is now capable of supporting multiple attributes, all handled by the magic of *__set_name__* which aids in setting up attribute name specific values for public and private in the *LoggedAttr* instance namespace. The important thing to understand here is that *LoggedAttr* instances are invoked at the class level, during interpretation of the `Car` class, before a `Car` instance has been created in memory, the `__set_name__` was already invoked, twice by python. Let's now understand `__set_name__` a little better.

5.8 Descriptors: __set_name__

Dunder *__set_name__* is called when the descriptors owning class is **created**. Note: This is **not** to be confused with instantiating an instance of the owner class, remember classes themselves are objects in python.

A very important fact of the `__set_name__` dunder is that it is only called as part of the `type` constructor. (to understand more about `type`, refer to my article on `metaclassess` in python3). This means that if a descriptor is dynamically bolted on after the fact, ``__set_name__` would need to be explicitly called. This is outlined below:

```
class Klazz:
    ...

descriptor = MyDescriptor()
Klazz.d = descriptor  # This is not sufficient.
descriptor.__set_name__(Klazz)  # Retrospectively, explicitly call __set_name__
↪.
```

5.9 Descriptors: __delete__

The final piece of the descriptor protocol, *__delete__()* is called to delete an attribute on an instance of the owner class. Implementing a `__delete__()` is enough to qualify the descriptor as a `Data Descriptor`. This is outlined below:

```
class D:

    def __delete__(self, obj):
        # self -> the instance of D
        # obj -> the instance of the owner class (where D() was instantiated␣
↪at the class level)
        print("deleting x")

class S:
    d = D()

s = S()
del s.d
# deleting x
```

5.10 Descriptors: Summary

- **A `descriptor` is any object that implements:**
  - *__get__*, *__set__*, *__delete__*
- **Optionally, descriptors can have a *__set_name__* if they need to know:**
  - The `class` they where created.
  - The name of the variable they where assigned too.
- `__set_name__` is invoked even for classes which are **not** descriptors.
- Descriptors get invoked by the *dot* operator, during attribute lookup.
- **Accessing a descriptor indirectly, the descriptor instance is not invoked but returned:**
  - vars(Klazz)['descriptor'] # returns the descriptor instance, but does not invoke __get__() etc.
  - `Klazz().__class__.x != Klazz().__class__.__dict__['x']`.
- Descriptors only work as `class` variables, stored in an instance has no effect.
- The main motivation for descriptors is to allow `class level` attributes to have a hook into attribute access.
- In a normal setup, the `calling` class controls what happens during lookup.
- Descriptors invert the control and allow the data being accessed to have a say in the matter.

5.11 Descriptors: A Real use case

So far, we have developed relatively trivial uses for python descriptors. Now we will put together all we have learned to implement a real use case. In this example we will build a *Field* descriptor that can validate data inputs, we will create a `BoundedInteger` to validate integers in a reusable, strict manner:

```
from abc import ABC
from abc import abstractmethod


class Field:
    def __set_name__(self, owner, name):
        self.private_name = "_" + name

    def __get__(self, obj, objtype = None):
        return getattr(obj, self.private_name)

    def __set__(self, obj, value):
        self.validate(value)
        setattr(obj, self.private_name, value)  # noqa

    @abstractmethod
    def validate(self, value):
        ...


class BoundedInteger(Field):

    def __init__(self, min: int = 0, max: int = 256):
        self.min = min
        self.max = max

    def validate(self, value):
        # For the sake of this demo, we want fine grained error messages!
        if not isinstance(value, int):
            raise TypeError(f"Expected {value!r} to be an integer")
        if not isinstance(self.min, int):
            raise TypeError(f"Expected {self.min} to be an integer")
        if not isinstance(self.max, int):
            raise TypeError(f"Expected {self.max} to be an integer")
        if not self.min <= value <= self.max:
            raise ValueError(f"{value} was not between: {self.min}, {self.max}␣
↪[inclusive]")


class RequiresValidation:
    value = BoundedInteger(min=0, max=10)

    def __init__(self, value):
        self.value = value

# Let's try it out!
RequiresValidation("foo")
```

(continues on next page)

(continued from previous page)

```
"""
Traceback (most recent call last):
  File "validation.py", line 47, in <module>
    RequiresValidation("foo")
  File "validation.py", line 43, in __init__
    self.value = value
  File "validation.py", line 13, in __set__
    self.validate(value)
  File "validation.py", line 30, in validate
    raise TypeError(f"Expected {value!r} to be an integer")
TypeError: Expected 'foo' to be an integer
"""

RequiresValidation(7.5)
"""
Traceback (most recent call last):
  File "validation.py", line 47, in <module>
    RequiresValidation(7.5)
  File "validation.py", line 43, in __init__
    self.value = value
  File "validation.py", line 13, in __set__
    self.validate(value)
  File "validation.py", line 30, in validate
    raise TypeError(f"Expected {value!r} to be an integer")
TypeError: Expected 7.5 to be an integer
"""

RequiresValidation(-1)
"""
Traceback (most recent call last):
  File "validation.py", line 47, in <module>
    RequiresValidation(-1)
  File "validation.py", line 43, in __init__
    self.value = value
  File "validation.py", line 13, in __set__
    self.validate(value)
  File "validation.py", line 36, in validate
    raise ValueError(f"{value} was not between: {self.min}, {self.max}␣
↪[inclusive]")
ValueError: -1 was not between: 0, 10 [inclusive]
"""

RequiresValidation(11)
"""
Traceback (most recent call last):
  File "validation.py", line 47, in <module>
    RequiresValidation(11)
  File "validation.py", line 43, in __init__
    self.value = value
  File "validation.py", line 13, in __set__
    self.validate(value)
  File "validation.py", line 36, in validate
```

(continues on next page)

(continued from previous page)

```
    raise ValueError(f"{value} was not between: {self.min}, {self.max}␣
↪[inclusive]")
ValueError: 11 was not between: 0, 10 [inclusive]
    """
```

5.12 Descriptors: Advanced

Up until now we have skimmed the technical internals of descriptors. It is important to grasp the previous concepts well before looking any deeper into the attribute lookup call flow etc.

We briefly touched on `data` and `non data` descriptors and mentioned how depending on which one the descriptor implementation is 'classified' as, has impacts on the attribute lookup call flow. To recap:

- Descriptor protocol consists of *__get__*, *__set__* and *__delete__*.
- Implementing any of the above qualifies.
- If only `__get__` is implemented, it is known as a `Non Data` descriptor
- If `__get__` + `__set__` || `__delete__` are implemented, it is known as a `Data` descriptor.

The default behaviour for attribute access is to get, set or delete an attribute from an object dictionary. for example:

- Firstly object_instance.attribute firstly looks for *attribute* in *object_instance.__dict__*
- Secondly, `type(object_instance).__dict__`
- Thirdly, resolving the `mro` of `type(object_instance)`.
- If the looked up object is a descriptor, python may invoke the descriptor instead
- note: Depending on which descriptor protocols are implemented, mileage varies.

5.13 Descriptors: The Protocol

```
class MyDescriptor:

    def __get__(self, obj, objtype = None):
        ...

    def __set__(self, obj, value):
        ...

    def __delete__(self, obj):
        ...

    # optional
    def __set_name__(self, owner, name):
        ...
```

The above is really all there is too it. `Data` and `Non Data` descriptors vary slightly in how the overrides are calculated in an instance dictionary. For example if a an instance dictionary has an attribute with the same name as the descriptor the `non data` descriptor will take precedence, however if an instance dictionary has an attribute with the same name

as a `data` descriptor, the dictionary attribute will take precedence. Let's understand what this means with an example below:

```
class DataDescriptor:
    def __get__(self, obj, objtype = None):
        print("Inside Data Descriptor Getter")

    def __set__(self, obj, value):
        print("Inside Data Descriptor Setter")


class NonDataDescriptor:
    def __get__(self, obj, objtype = None):
        # Never called.
        print("Inside Non Data Descriptor Getter")


class DataDescriptorOwner:
    x = DataDescriptor()

    def __init__(self, x):
        self.x = x


class NonDataDescriptorOwner:
    x = NonDataDescriptor()

    def __init__(self, x):
        self.x = x


d = DataDescriptorOwner(100)
#  Inside Data Descriptor Setter
d.x
# Inside Data Descriptor Getter

# -----

n = NonDataDescriptorOwner(13)
n.x
# 13 <no __get__ or __set__ is called because a `non data` descriptor instance␣
↪`x` takes priority.
```

In order to make a `read-only` descriptor, implement `__set__` and raise an `AttributeError`. As we briefly touched on earlier, defining a `__set__` with an exception raising placeholder is sufficient to have the descriptor instance be considered a `data descriptor`.

5.14 Descriptors: Invocation

CHAPTER

SIX

VIRTUAL SUBCLASSING

6.1 Python Virtual subclassing

Unlike some other languages, python supports the concept of `virtual subclassing`. What exactly is virtual subclassing? To understand the concept let's go back to pythons roots. I am sure you are familiar with the concept of *duck typing* and perhaps if you have came from another language such as `java`, you will be familiar with the concept of an `interface`. In python we are not overly caught up on the *type* of *X*, but more so how *X* behaves, in the simplest form if it walks like a duck and quacks like a duck, chances are its a duck. Python does not support an official `interface` argument, however more recently `Protocols` can be `@runtime_checkable` to aid with catching issues early.

To understand why we may need virtual subclassing, let's take a simple example. Firstly we are tasked with developing an airplane tycoon game. Our game consists of multiple things that can fly, to use the age old cliche:

- A Bird
- An Aeroplane

Turns out, you cannot load people into A Bird and fly them to a new destination, so we develop a few abstract base classes to easily distinguish them

```
from abc import ABC
from abc import abstractmethod

class Plane(ABC):
    @abstractmethod
    def fly(self):
        ...

# We make a simple plane

class MyPlane(Plane):
    def fly(self):
        print("Plane preparing to fly!")


# Now we can check in our library function
# Note: isinstance, isinstance checks against an ABC are even questionable..
def generate_flight(plane: Plane):
    if not isinstance(plane, Plane):
        raise TypeError("Not a Plane!")
    plane.fly()
```

(continues on next page)

(continued from previous page)

```
# Now, if we accidentally receive a Bird, we will handle the case
# This will allow us to handle it gracefully rather than potentially
# Blow up with some other weird errors later on in the program
class Bird:
    def fly():
        print("bird flying!")

generate_flight(Bird())
"""
    TypeError                                 Traceback (most recent call last)
    <ipython-input-16-be9dc02ec41e> in <module>
    ----> 1 generate_flight(Bird())

    <ipython-input-15-d59f577c3fde> in generate_flight(plane)
         18 def generate_flight(plane: MyPlane):
         19     if not isinstance(plane, MyPlane):
    ---> 20         raise TypeError("Not a Plane!")
         21     plane.fly()
         22

    TypeError: Not a Plane!
"""
generate_flight(MyPlane())
# Plane preparing to fly!
```

Excellent, but whats the point in virtual subclassing still? Let's say in the near future some other fantastic library comes along with a dozen of cool new planes, the problem is our `generate_flight()` function here is strictly prohibiting non explicit subclasses of `Plane`. Let's take a look at the library code

```
# snippet from another library, not developed by you
# boeing.py
class Boeing747:
    def fly(self):
        print("Boeing 747 preparing for travel!")

    def repair(self):
        ...
```

Pretty simple huh, a nice shiny new Boeing that we could use in our system, except of course it does not adhere to our explicit abstract base class:

```
from boeing import Boeing747

generate_flight(Boeing747())
"""
TypeError                                 Traceback (most recent call last)
<ipython-input-23-24437f4c4918> in <module>
----> 1 generate_flight(Boeing747())

<ipython-input-20-8038f0273ec8> in generate_flight(plane)
     18 def generate_flight(plane: MyPlane):
     19     if not isinstance(plane, MyPlane):
```

(continues on next page)

(continued from previous page)

```
---> 20         raise TypeError("Not a Plane!")
     21     plane.fly()
     22

TypeError: Not a Plane!
"""
```

Damn, we have coupled our library a little too tight and we don't own the library code, what gives? Rather than monkey patching and hacking around the inheritance of `Boeing747`, enter `virtual subclassing`. We can simple register the third party code as a virtual subclass of our _interface_ (abstract base class) and the python interpreter will treat it like it has actually subclassed it.

```
from boeing import Boeing747

Plane.register(Boeing747)
isinstance(Boeing747(), Plane)  # True!
issubclass(Boeing747, Plane)  # Also True!
generate_flight(Boeing747())
# Boeing 747 preparing for travel!
```

Voila, we have successfully used third party code and ackknowledged explicitly that we accept it is an adaquate implementation of our interface.

Note: virtual subclassing should be used extremely sparingly, in reality the need for it is often miniscule. It is also possible to automatically consider objects as instance/subclasses based on their interface and python does this internally a lot in its collections.abc module, more on that in a separate post alter.

CHAPTER

SEVEN

ITERATOR PROTOCOL

7.1 Python Iterator Protocol

Here we out cover the iterator protocol in depth, both the newer version via `__iter__` and `__next__` as well as the older protocol piggy backing off ``__getitem__` for sequence types.

In python if you have found yourself wondering, how does the *for each* loop work? This is where the iterator protocol comes into play and making your own iterable user defined types is surprisingly straight forward. Personally the terminology is more complex than the actual logic involved. Let's try and break it down:

- `collections.abc.Iterator` extends `collections.abc.Iterable`
- All `iterators` are `iterable`
- Not all `iterables` are `iterators`

7.2 Iterator Protocol: Iterable ABC

`collections.abc.Iterable` is an abstract base class built into python that offers up the abstract `__iter__` method that should be implemented. Rule of thumb is that anything that is `iterable` when asked for an `iterator` will return one.

```
@abstractmethod
def __iter__():
    while False:
        yield None
```

That's it, iterables are really that simple, if something is iterable, it can return an `iterator`. It is an `iterator` that python actually uses to perform iteration. The built in ``iter()` function calls an objects dunder `__iter__`.

7.3 Iterator Protocol: Iterator ABC

Once you have grasped that iterables are _responsible_ for returning `iterators`, things start to make a lot more sense. Next up is the `collections.abc.Iterator` abstract base class provided by python and there are three main core things of note

- Iterator extends Iterable.
- Iterator implements a simple `__iter__` to make Iterable happy, that returns itself.

- Iterator exposes a new `__next__` abstract method.

As we can see from inspecting the mro of `collections.abc.Iterator`:

```
from collections.abc import Iterator

Iterator.__mro__
#  collections.abc.Iterator, collections.abc.Iterable, object)
```

In order to satisfy the interface from `collections.abc.Iterable`, it implements a very basic `__iter__`, which returns self:

```
def __iter__():
    return self
```

However, iterators themselves offer up a little extra, the interface exposes a new abstract method, known as `__next__` and it is this implementation that when iterated over (for loops, *map*, list comps etc) that is used to exhaust the iterator, one element at a time:

```
def __next__(self):
    raise StopIteration
```

Something of note here, is unlike other languages, occasionally python uses exceptions to handle code logic / flow, when an Iterator is exhausted it should raise a `StopIteration` Exception, this is how python knows internally that there are no more values.

7.4 Iterator Protocol: __getitem__

There is another caveat, an object does not have to define the modern iterable/iterator interfaces to qualify as being iterable, using dunder `__getitem__` if an object can take an integer starting from *0*, python will happily iterate over that object as well, this is known as the older iterator protocol, however due to the symantics, when iterating using this approach, an `IndexError` should be raised instead of the traditional `StopIteration`. Let's demonstrated an example:

```
class ReversedEvenNumbers:
    def __init__(self, max):
        self.nums = [n for n in range(1, max+1)[::-1] if n % 2 == 0]

    def __getitem__(self, index):
        return self.nums[index]

for n in ReversedEvenNumbers(15):
    print(n)
# 14, 12, 10, 8, 6, 4, 2
```

As you can see, we have created something we can iterate over, without actually implementing any of the iterator (modern) protocol. Accessing an index out of range by default raises an `IndexError` so python gracefully handles that in this scenario.

7.5 Iterator Protocol: Modern Example

We have learned a little bit about the older iterator protocol with an example, however let's implement something a little more modern. Now we will use the abstract base classes and create our own custom iterator and explain some of the magic behind pythons virtual subclassing via `abc.register` and the `__subclasshook__`.

In this example, we will be creating a word iterator from a user provided sentence. Continue reading after this topic to understand why our `Sentence` class does not have to explicitly inherit from `collections.abc.Iterator` (a little sprinkle of python magic!):

```
# A typical first approach (albeit naive)
class Sentence:
    def __init__(self, sentence: str) -> None:
        self.word_list = sentence.split()
        self.index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.index >= len(self.word_list):
            raise StopIteration
        value = self.word_list[self.index]
        self.index += 1
        return value
```

When starting out, you might think something like this is, pretty good. However there are a couple of caveats you should be aware of, each time iter(iterable) is called, it should return a fresh `iterator`. What happens in this scenario with the above implementation:

We need to create a fresh iterator, each time python calls `__iter__` on our object. Let's patch that up first:

Better! Each time we ask for an iterator from our custom `RevisedSentence` class, we can access all the values, but can it be improved any more? We'll, python supports a ton of built iterators / iterables, we can much easier piggy back off those in this kind of scenario:

```
class SuperSentence:
    def __init__(self, sentence: str):
        self.word_list = sentence.split()

    def __iter__(self):
        return iter(self.word_list)
```

7.6 Iterator Protocol: Virtual & Subclasshook

...

CHAPTER

EIGHT

SHALLOW & DEEP CLONING

8.1 Copying: Shallow Copy

Shallow copying a container in python builds a new instance of the container type, but the elements inside the container are just references to the same objects in the container prior to copying. This means that if you shallow copy a list for example, and update the index[0], both lists will be updated as they are the same object in memory, let's try it out:

```
items = [1,2,3,4,5]
new_items = items.copy()
items[0] = 10
items [10, 2, ...]
new_items  # [10, 2, ...]
```

8.2 Copying: Deep Copy

...

CHAPTER

NINE

CONTEXT MANAGERS

9.1 Context Managers: Introduction

Python context managers are what underpin the `with` statement in python and they are used for managing resources which need to be closed and or cleaned up. The typical example with any introduction to context manager in python is the open statement, to avoid leaving files open unnecessarily, python builtins support the following:

```
with open("myfile.txt", mode="w+") as file:
    file.write("Hi there\n")
```

In order to create our own user defined context managers, there are typically two simple approaches, we will discuss both options in depth throughout this post:

- `contextlib.contextmanager` (as a decorator around a generator function (yield).
- `__enter__` and `__exit__` dunder implementations in our own user defined classes.#

9.2 Context Managers: User Defined Classes

We touched briefly on the dunder `__enter__` and `__exit__` methods that can be used to turn a class into a context manager.

- `__enter__(self)`

Enters the `runtime context` and returns either `self` or another object _related_ to the `runtime context`. The `with` statement will bind this return value to the target specified in the `as` statement, a simple example:

```
from __future__ import annotations
from typing import Tuple


class Klazz:
    def __enter__(self) -> Klazz:
        print("Entering the runtime context...")
        return self

class Klazz2:
    def __enter__(self) -> Tuple[int, int, int]:
        print("Entering the runtime context & returning something else")
        return 100, 200, 300
```

(continues on next page)

(continued from previous page)

```
with Klazz() as k:
    ...

with Klazz2() as a, b, c:
    ...
```

- `__exit__(self, exc_type, exc_value, traceback)`

Dunder exit as you could have guessed, is responsible for `closing` a resource, clean up after the core with block has been executed, called implicitly by python. There are a few things to know about dunder `__exit__` and we will discuss those here as well as some of the caveats of incorrectly implementing it.

Dunder `__exit__` exits the runtime context and in provides exception information for any exceptions unhandled during the runtime context, if no exceptions where raised during the runtime context, then all parameters passed to __exit__ will be `None`. This is outlined below:

The parameters passed in to handle the exception information is as follows:

- `exc_type` -> The `type` of the exception class raised.
- `exc_value` -> The exception value, e.g -> `Raise ValueError(10)` -> *10*.
- `traceback` -> The traceback `instance`.
- `Note:` If no unhandled exceptions occurred, all three are *None*, making them *Optional*.

If we had to document those in terms of python types, it would look something like this:

```
from typing import Type
from typing import Optional
from types import TracebackType
from contextlib import AbstractContextManager

class K(AbstractContextManager):  # This implements a self returning __enter__␣
↪mixin.
    def __exit__(self,
        exc_type: Optional[Type[BaseException]],
        exc_value: Optional[BaseException],
        traceback: Optional[TracebackType]
    ):
        print(exc_type, exc_value, traceback)

with K(): as k:
    ...
# None, None, None -> No exception was raised and unhandled!

with K() as k:
    try:
        raise ValueError(10)
    except ValueError:
        ...
# None, None, None -> Raised exception was handled.

with K() as k:
    raise ValueError(100)
```

(continues on next page)

(continued from previous page)

```
# <class `ValueError`>, 100, <traceback object at 0x7f052c53d200> (unhandled␣
↪exception).
```

A word of warning about `__exit__`, the return type of dunder exit is evaluated in a boolean context where `truthy` values result in suppressing unhandled exceptions. Dunder `__exit__` should also avoid re raising the exception which is passed in by python when unhandled exceptions occur in the runtime context, this is the responsibility of the caller.

```
from contextlib import AbstractContextManager

class SuppressedExc(AbstractContextManager):
    def __exit__(self, exc_type, exc_value, traceback):
        return True  # Truthy -> True, suppresses exceptions...!


with SuppressedExc() as s:
    raise ValueError(100)

# No exception raised here!

class NotSuppressedExc(AbstractContextManager):
    def __exit__(self, exc_type, exc_value, traceback):
        return False

with NotSuppressedExc() as ns:
    raise ValueError(200)

"""
ValueError                                Traceback (most recent call last)
<ipython-input-7-55fb72d3f55a> in <module>
      1 with NotSuppressedExc() as ns:
----> 2     raise ValueError(200)
      3

ValueError: 200
"""
```

9.3 Context Managers: contextlib

Python ships out of the box with the `contextlib` module, which is a utility module for using various python context managers as well as some context managers that can make using other non context managers easier.

9.4 Context Managers: closing

The `contextlib.closing` context manager can be used to automatically close another object that itself is maybe not necessarily a context manager. It simply takes the object instance and calls a *.close()* method on it, in a nutshell it would be like this:

```
from contextlib import contextmanager

@contextmanager
def close_it(obj):
    try:
        yield obj
    finally:
        obj.close()
```

this allows us to write code like this for any object that has a *.close()* method but itself is not a context manager.

```
from urllib.request import urlopen

with close_it(urlopen("https://www.google.com")) as page:
    for line in page:
        print(line)
```

Even if an exception is raised here, the *page* will always have *.close()* invoked on it.

9.5 Context Managers: nullcontext

`contextlib.nullcontext` can be used to return a no-op, it is intended for use as a stand in for an optional context manager. Based on some logic, e.g some `if` clause, you may use a `nullcontext`, a good example of such a use case is:

```

```

from contextlib import nullcontext from contextlib import suppress

def function(ignore_exceptions: bool = False): mgr = suppress(Exception) if ignore_exceptions else nullcontext() with mgr:

... # Do something, depending on the function arg, exceptions are suppressed!

Basically if you may want to run some sort of context manager or not based on some branched logic in your code, `nullcontext` can be used as a standard in to fill the gap in some alternative case.

9.6 Context Managers: suppress

Often it is necessary to run some piece of code while ignoring an assortment of exceptions, simplifying a `try: except: pass` kind of setup.

```
from contextlib import suppress

# Approach 1
```

(continues on next page)

(continued from previous page)

```
def try_something():
    try:
        do_some_operation()
    except ValueError:
        pass


def with_suppress():
    with suppress(ValueError):
        do_some_operation()
```

CHAPTER

TEN

COLLECTIONS: NAMEDTUPLE

10.1 Namedtuple: Introduction

`collections.namedtuple` is a simple factory function for building more advanced tuples. Just like tuples they permit indexing, are iterable and the main benefit they offer over a standard tuple is that attributes can be accessed by name. Here is a basic introduction to named tuples:

```
from collections import namedtuple

Foo = namedtuple("Foo", "bar, baz", defaults=(100, 200))
print(Foo())
#  Foo(bar=100, baz=200)
```

As you can see from this small snippet, namedtuple also offer a nice dunder *__repr__* implementation right off the bat. We will discuss more throughout this article, especially the factory function arguments in great detail, but for now just know that namedtuples create tuple subclasses with attribute access.

10.2 Namedtuple: Factory Function

The `namedtuple(...)` factory function offers a lot of additional arguments, often overlooked and to be honest, rarely used, however for a full overview, we will discuss each argument and what it does with an example.

```
collections.namedtuple(
    typename: str,
    field_names: Iterable[str],
    *,
    rename: bool = False,
    defaults: Optional[Any] = None,
    module: Optional[Any] = None
)
```

10.3 Namedtuple: typename

typename is the new tuple subclass name. In order for pickling to be natively supported the typename should match the name of the variable assigned to the tuple subclass. This is briefly shown below:

```
import pickle
from collections import namedtuple

Foo = namedtuple("Bar", "a,b,c", defaults=(200,300))
f = Foo(a=25)
print(f)
#  Bar(a=25, b=200, c=300)
pickle.dumps(f)  #  PicklingError: Can't pickle <class '__main__.Bar'>
Foo2 = namedtuple("Foo2", "a,b,c", defaults=(200, 300))
f = Foo2(25)
fbytes = pickle.dumps(f)
# b'\x80\x04\x95 \x00\x00\x00\x00\x00\x00\x00\x8c\x08__main__\x94\x8c\x04Foo2\
↪x94\x93\x94K\x19K\xc8M,\x01\x87\x94\x81\x94...."
```

10.4 Namedtuple: field_names

`field_names` is a sequence of strings or an individual string of the attribute names to be assigned to the underlying tuple subclass. In the latter, attribute names are automatically resolved by splitting on either a comma, or whitespace, named tuples do not have an underlying `__dict__` instance (think `__slots__`) which is what allows them to compete with standard tuples on memory.

```
from collections import namedtuple

f = namedtuple("f", ["one", "two", "Three"])
f2 = namedtuple("f2", "one two three")
f3 = namedtuple("f3", "one, two, three")
```

field names can be any valid python identifier except for anything starting with an underscore. named tuple has an extra param we will discuss later *rename*= which is used for rewriting illegal field names automatically with a prefixed underscore.

10.5 Namedtuple: rename

As previously outlined, *rename* works in tandem with the *field_names* argument in order to automatically rewrite name violations with a prefixed _ positional names, where each violation is incremented += 1. For example:

```
from collections import namedtuple

One = namedtuple("One", "one, def, two, class, three, return", rename=True)
one = One(10, 20, 30, 40, 50, 60)
#  One(one=10, _1=20, two=30, _3=40, three=50, _5=60)
```

As you can see in the example, field names *def*, *class* and *return* are also python core builtin reserved keywords, these have automatically been rewritten with _<*n*> for each violation in the sequence passed to *field_names*.

10.6 Namedtuple: defaults

namedtuple defaults is an `iterable` of names to unpack into the fields when a value is omitted. By default, the values are unpacked from <- right to left, so if there are three field names defined *a,b,c* and two defaults *defaults=(100, 200)* then *b == 100* and *c == 200*, *a* is a required field in this instance. `defaults=` can also be `None` in which case, all field_name attributes are *required*.

```
from collections import namedtuple

Foo = namedtuple("Foo", "a,b,c", defaults=(10, 20))
f = Foo()
# __new__() missing 1 required positional argument: 'a'
f = Foo(2000)
print(f) #  Foo(a=2000, b=10, c=20)
```

10.7 namedtuple: module

namedtuple allows you to customise the *module* of the tuple subclass, if *module=* is assigned then the dunder `__module__` of the namedtuple will be set to that. `__module__` is a writable field defining the name of the module the function was defined in. This is shown below (using an interactive *ipython* shell where by default the module would be `__main__`.

```
from collections import namedtuple

T1 = namedtuple("T1", "a,b,c", defaults=(1,2,3), module="foomod")
T2 = namedtuple("T2", "a,b,c", defaults=(3,2,1))

# T1 has a custom module name assigned; let's inspect its instances:
T1().__module__  # foomod
# T2 omits the module attribute from the sig
T2().__module__  # __main__
```

10.8 Namedtuple: misc

In order for namedtuple instances to be a core part of the python language, they need to retain some of the benefits of standard tuple types. Namedtuples do not have a per instance dictionary (only a class one) this is how they are able to retain the same memory footprint as normal tuples. They are of course also immutable and in order to support pickling by default, the variable named assigned to the namedtuple instance should match that of the defined *typename*. These are outlined below:

```
# -- Memory Footprint
from sys import getsizeof
t = (100, 200, 300)
nt = namedtuple("Foo", "a b c", defaults=(100,200,300))()  # Create the␣
↪instance!
getsizeof(t)  # 64 bytes
getsizeof(nt)  # 64 bytes

# -- Immutability
```

(continues on next page)

(continued from previous page)

```
Immutability = namedtuple("Immutability", "one, two", defaults=(500, 600))
immutable = Immutability()
immutable.__dict__  # `Immutability object has no attribute __dict__`
immutable.one, immutable.two  # (500, 600)
immutable.one = 2  # AttributeError: Cannot set attribute

# -- Pickle capabilities
import pickle
Works = namedtuple("Works", "a")
w = Works(10)
pickle.dumps(w)  # bytes no problem.

DoesntWork = namedtuple("Different", "a")
d = DoesntWork(20)
pickle.dumps(d)  # PicklingError: Can't pickle <class '__main__.Different'>:␣
↪attribute lookup Different on __main__ failed
```

10.9 Namedtuple: _make

The first of the three main methods that are bolted onto namedtuple instances. *_make* is a *@classmethod*. that uses *tuple.__new__* under the hood to create a new namedtuple instance from an iterable.

```
from collections import namedtuple

T = namedtuple("T", "a b c", defaults=(100,150, 200))
t = T()  # T(a=100, b=150, c=200)
t2 = t._make((5,15,25)) # T(a=5, b=15, c=25)
```

10.10 Namedtuple: _asdict

The second of the three main methods that are bolted onto namedtuple instances. *_asdict* returns a dictionary of the namedtuple instance attributes and corresponding values. As of python *3.8* the *_asdict* function returns a normal dictionary, if you need the benefits of an *OrderedDict* consider instantiating one directly using this *_asdict* function:

```
from collections import namedtuple
from collections import OrderedDict

T = namedtuple("T", "a,b", defaults=(50, 100))
t1 = T()
mapping = t1._asdict()
#  {"a": 50, "b": 100}
order = OrderDict(t1._asdict())
#  OrderedDict([('a', 50), ('b', 100)])
```

10.11 Namedtuple: _replace

The third of the three main methods bolted onto namedtuple instances is *_replace*. This allows you to create a new instance of the tuple subclass, replacing fields of the existing instance with keys and respective values from the ***kwargs* mapping:

```
from collections import namedtuple
T = namedtuple("T", "a,b,c", defaults=(12, 24, 36))
t = T()  # T(a=12, b=24, c=36)
mapping = {"c": 4000}
t2 = t._replace(**mapping)
t2  # T(a=12, b=24, c=4000)
```

10.12 Namedtuple: _fields

The *_fields* instance attribute is used for a simple tuple of the namedtuple instance field names. This is useful for introspection and new namedtuple instances containing a subset of an existing instances fields, this recipe is outlined below:

```
from collections import namedtuple

T = namedtuple("T", "a")
t = T(100)  # T(a=100)
T2 = namedtuple("T2", t._fields + ("b", "c"), defaults=(50, 60,70))
t2 = T2()
t2 # T2(a=50, b=60, c=70)
```

10.13 Namedtuple: _field_defaults

Namedtuple *_field_defaults* returns a mapping of fields to their respective default values:

```
from collections import namedtuple
T = namedtuple("T", "one, two, three", defaults=("three", "two"))
t = T(500)
t._field_defaults
#  {"two": "three", "three": "two"}
```

CHAPTER

ELEVEN

POSITIONAL AND KEYWORD ARGUMENTS

11.1 Function arguments: Introduction

Python offers three main ways to control function arguments, these are:

- positional only
- positional or keyword
- keyword only

Here is a short overview of what we will cover here:

```
def example(pos_only /, pos_or_kwd, *, kwd_only)
    # pos_only is positional only, order matters and cannot be passed via pos_
↪only=1
    # pos_or_kwd like traditional args (without /, *) can be specified by␣
↪order (pos) or keyword explicitly
    # kwd_only must be specified as kwd_only=10
    ...

example(1, 2, kwd_only=3)  # Valid
example(1, pos_or_kwd=2, kwd_only=3)  # Valid
example(1,2,3) # Invalid (kwd_only must be keyword only)
example(pos_only=1, 2, kwd_only=3)  # Invalid (pos_only cannot be specified␣
↪via keyword)
```

11.2 Positional Only Arguments

Arguments specified before a / in a function signature, indicate the argument should only be passed as a positional argument (e.g without a keyword and order matters):

```
def one(a, b, c, /):
    ...
```

In the above example, a,b,c must be passed explicitly without a keyword argument.

11.3 Positional OR Keyword Arguments

By default, when either / or * is omitted from a function signature, arguments are considered *positional_or_keyword* args, which means passing them positionally in order is completely valid and passing them via keyword explicitly is also valid:

```
def one(a,b,c):
    ...

one(1,2,3)  # Valid
one(a=10, b=20, c=30)  # Valid
```

11.4 Keyword only Arguments

Arguments which follow a trailing * are considered keyword only arguments. These must be explicitly passed via a keyword= call:

```
def one(a, b, /, * c):
    ...

one(10, 20, c=100)  # Valid, c= must be explicit
one(10, 20, 30)  # Invalid, TypeError: one takes two positional arguments by 3␣
↪were given
```

CHAPTER

TWELVE

REGULAR EXPRESSIONS

A regular expression is a pattern that is matched against a subject string, from left to right. Some common uses of regular expressions (but not exhaustive) are:

- Replacing text within a string
- Capturing groups of information from a string
- Validating data, like a user name with multiple constraints
- much more…

12.1 A Trivial Example

As we previously just mentioned, a common use case for regex is validating user input against an assortment of different constraints. To take the user name validation example, let's look at how we might validate the following constraints on user input:

- Must begin with a capital letter
- Must be at least 10 characters in length
- Must end with a number
- Can only be alphanumeric

This is a simple example, and the point is just to serve as an introduction to regular expressions. Let's have a look at how we would implement such a scenario in python and explain step by step what each part is doing. Don't worry too much about understanding to following as in this article we will be breaking down the core fundamentals of regular expressions into digestable chunks. The aim by the end of it, is that you should be able to piece together complex expressions for matching an assortment of scenarios:

```
import re  # Import pythons regular expressions module

# For demonstration purposes; we will build the string over multiple steps
# for ease of understanding
pattern = r"" # starting point; empty raw string
pattern += "[A-Z]{1}"  # First character MUST be an uppercased A -> Z character
pattern += "[a-zA-Z0-9]{8,}  # Must then contain AT LEAST 8 additional␣
↪characters /1[8+]1/
# We have implicitly guaranteed so far that we have an uppercase char[0] and 8␣
↪alpha numeric chars ending in a digit.
pattern += "[0-9]{1}"  # Must end with a number
```

(continues on next page)

(continued from previous page)

```
# putting it altogether then, pattern is:
pattern = r'[A-Z]{1}[a-zA-Z0-9]{8,}[0-9]{1}'
re.match(pattern, "ValidPassword2")  # <reMatch object; spam=(0, 14), match=
↪'ValidPassword2'>
re.match(pattern, "invalidPassword5") # None (no match due to missing initial␣
↪capital)
re.match(pattern, "InvalidPassword")  # None (no match due to missing ending␣
↪digit
re.match(pattern, "Invalid5")  # None, too short!
```

If you are experienced in regular expressions; you may be screaming that there are *other* or *better* ways to do exactly this; often with regular expressions there are many ways to skin a cat, but for simplicity and to serve as an introduction, this is a decent enough example. Again, if this is all new to you, focus on trying to understand it but not remember it, we will be going in-depth shortly.

Note: Here we are reusing the `pattern` string, it is advisable when reusing a pattern to compile it into a `re.Pattern` object using `re.compile(pattern)`.

12.2 Regular Expr: Simple Matchers

In it's simplest form, a regular expression is just a bunch of characters that we use to perform a search in a string, for each snippet in this article we will be sharing an example of the syntax in action as well as an interactive link to dabble and view it yourself.

Table 1: Simple Matcher

Pattern	Subject String	Expected Match
example	This is a trivial example	This is a trivial **example**
bar	Foo bar	Foo **bar**

Try Simple Matcher: https://regex101.com/r/tTZsZN/1

Typically regular expressions are case **insensitive**, (outside of using the *i* flag - more on that towards the end of the article under the *flags* section).

```
import re
re.match("foo", "Foo will not match")
```

12.3 Regular Expr: Meta Characters

Meta characters are the bread and butter of regular expressions, and understanding them can make staring at a daunting regular expression become somewhat demystified. Here is a brief summary of the core meta characters:

Table 2: Regex Meta Characters

Meta Characters	Description
`.`	Period matches any single character, except a line break character e.g *n*
`[]`	Character classes. Match any character contained within the brackets.
`[^]`	Negated Character classes. Match any character **NOT** contained within the brackets.
`?`	Makes the preceding symbol *optional*.
`+`	Matches **one** or more of the preceding symbol.
`*`	Matches **zero** or more of the preceding symbol.
`{i, j}`	Braces. Matches **at least** *i* but no more than *j* repetitions of the preceding symbol.
`(foo)`	Character group. Matches the characters *foo* in exactly that order.
`|`	Alternation. Matches characters either before **or** after the symbol.
`\`	Escapes the next character, This allows using meta characters (and others) in their literal sense.
`^`	Carat. Matches the beginning of the input (also has use in negative character classes).
`$`	Dollar sign. Matches the end of the input. *^foo$*.

12.4 Regular Expr: Meta -> .

The meta character `.` is used to indicate any single character. This has some exclusions for things like line breaks and it is also worth noting that certain language re implementations can permit flags which also allow this character to match even line breaks as well, we will discuss that here using pythons `DOTALL` flag.

Table 3: Meta Full Stop

Pattern	Subject String	Expected Match
`.at`	I put a hat on my cat	I put a **hat** on my **cat**
`foo.`	foo1 with foo2	**foo1** with **foo2**

Try Full Stop: https://regex101.com/r/Ii7Bj9/1

```
import re
pattern = r"foo."
re.findall(pattern, "foo1 with foo2")
# ["foo1", "foo2"]
```

Line breaks and pythons *DOTALL* flag example:

```
import re
foo = "foo\n"
re.match("foo.", foo)
#  No Match as `.` does not match on the new line
re.match("foo.", foo, flags=re.DOTALL)  # Capture line breaks too!
# < re.Match object; span=(0,4), match='foo\n'>
```

12.5 Regular Expr: Character Classes -> [...]

Character classes in regex are used to denote literal values, so using meta characters inside them do not need escaped. Hyphens can be used inside character classes to signify a range, just like we used in the initial example (username validation). Character classes are denoted by the `[ <–> ]` square brackets. Order inside character classes does **not** matter:

Table 4: Meta Character Classes

Pattern	Subject String	Expected Match
`[Tt]he .at`	The cat	**The cat**
`[sMc]at`	The cat, sat on the Mat	The Foobar, was **foobar**

Try Character Classes: https://regex101.com/r/8iSKB8/1

```
import re
pattern = re.compile(r"[sMc]at")
re.findall(pattern, "The cat sat on the Mat")
# ['cat', 'sat', 'Mat']
```

12.6 Regular Expr: Negated Character Classes -> [^...]

Similar to the `Character Classes` outlined previously, the negated character class matches anything **except** what is defined inside the square brackets. We mentioned previously how the carat `^` symbol can denote the start of the string, however it's additional use case is here (as well as in *lookarounds* more on that one later..). Here we will find any words that do **NOT** start with a letter:

Table 5: Meta Negated Character Classes

Pattern	Subject String	Expected Match
`[^a-zA-Z]*`	NoMatch	<no match>
`[^a-zA-Z]*`	5Matched	**5Matched**

Try Negated Character Classes: https://regex101.com/r/meqZgw/1

```
import re

pattern = re.compile(r"[^a-zA-Z].*")
re.match(pattern, "failed")
re.match(pattern, "5Passed")
```

Note: There are some short hand tricks with regex, which we will discuss later, things like *d* and *w* but for simplicity, bear with me for now. You will also notice various methods of the python `re` module here, the difference between `re.search`, `re.match` and `re.findall` will be outlined later on as well.

12.7 Regular Expr: Question Mark -> ?

The meta character ? indicates an **optional** preceding character (or group). This matches **zero** or more of the preceding character.

Table 6: Meta Optional Repetition (?)

Pattern	Subject String	Expected Match
`[T|t]?he`	he	**he**
`[T|t]?he`	The	**The**

Try Optional Repetition (?): https://regex101.com/r/KQSs7f/1

```
import re
pattern = re.compile(r"[T|S]?he")
re.match(pattern, "The")  # <re.Match object; span=(0, 3), match='The'>
re.match(pattern, "She")  # <re.Match object; span=(0, 3), match='She'>
re.match(pattern, "he")  # <re.Match object; span=(0, 2), match='he'>
```

12.8 Regular Expr: Plus -> +

The meta character + indicates **one** or more repetitions of the preceding character. Unlike the * there should be at least one character. If used after a character class or capture group it finds the repetitions of the character set also. So for example:

Table 7: Meta Optional Repetition (+)

Pattern	Subject String	Expected Match
a+bc	aaaaaaaaaaaaaaaaaaaaaaaaabc	**aaaaaaaaaaaaaaaaaaaaaaaaabc**
a+bc	bc	<No Match>

Try Required Repetition (+): https://regex101.com/r/sH0Bmf/1

```
import re

pattern = re.compile(r"a+bc.*")
re.match(pattern, "abcdef")  # <re.Match object; span=(0,6), match='abcdef'>
re.match(pattern, "abc")  # <re.Match object; span=(0,3), match='abc'>
re.match(pattern, "bc")  # None
```

12.9 Regular Expr: Plus -> *

In a similar sense to the + repetition meta character, * indicates that the preceding character can be either **optional** or infinite amount of the previous character. If used after a character class or capture group it finds the repetitions of the character set also.

Table 8: Meta Optional Repetition (+)

Pattern	Subject String	Expected Match
a*bc	aaaaaaaaaaaaaaaaaaaaaaaaabc	**aaaaaaaaaaaaaaaaaaaaaaaaabc**
a*bc	bc	**bc**

As you an see above, the core difference from + and * here is that the pattern *a*bc* will match if a exists or not, a simple demonstration of that is outlined below:

Try Optional Repetition (*): https://regex101.com/r/sH0Bmf/1

```
import re

star = r"a*bc"
plus = r"a+bc"
text = "bc"
re.search(plus, text)  # NoneType (no match!)
re.search(star, text) # "bc" <re.Match object; span=(0, 2), match='bc'>
```

12.10 Regular Expr: Braces -> { }

Braces (also known as *quantifiers*) are used to apply constraints to the number of repetitions of the previous character or group of characters, Let's say we wanted to write some

CHAPTER

THIRTEEN

CONCURRENCY: THREADING

13.1 Threading: Introduction

Before we start a deep dive into python threading, a core concept to understand is that python in the context of (CPython) has a **Global Interpreter Lock** (GIL). Certain performance centric libraries are able to overcome this limitation but it often involves moving into *C* extensions etc. When wondering if threading is the right answer, consider:

- Moving to *Processing* if you find yourself CPU bound and wish to benefit from multiple cores.
- Use threading if you find yourself IO blocked and want to open up some simultaneous execution of those tasks.

13.2 Threading: Overview

todo

13.3 Threading: Thread Local

CHAPTER

FOURTEEN

PASS BY ASSIGNMENT

14.1 Pass By Assignment

In python, arguments are *passed by assignment*. Not to be confused with *pass by reference* or *pass by value*. The rationale behind this is two fold:

- The parameter passed in by the caller is actually a *reference* to an object.
- Some data types are immutable (e.g *int*, *string*, *bytes*, *tuple* etc..)

What this means in practice is:

- If the caller passes a *mutable* object to a method, the method itself gets a reference

to that same object and it can be mutated as desired, however rebinding the reference inside the method will **NOT** reflect to the outer scope (callers) reference. Rebinding and mutating will not reflect on the callers object.

- If the caller passes a *mutable* object, rebinding the reference will also not impact the

callers object and you will not be able to update the state.

14.2 Pass By Assignment: Mutable

```
mutable = [1,2,3]

def demo_mutable(data):
    mutable.append(4)  # This will modify the outer scope, the callers␣
↪`mutable` will be updated after.

print(mutable)
# [1,2,3,4]

def demo_rebinding(data):
    # remember data[-1] == 4 now.
    data = [1,2,3,4,5,6,7]

print(mutable)
# [1, 2, 3, 4] - outer scope `mutable` is not impacted due to rebinding inside␣
↪`demo_rebinding(...)`
```

14.3 Pass By Assignment: Immutable

```
immutable = "string"

def demo_mutable(data):
    data += "foo"

print(immutable)
# 'string'

def demo_rebinding(data):
    data = "foo"

print(immutable)
# 'string'
```

CHAPTER

FIFTEEN

STRING METHODS

15.1 capitalize

Returns a copy of the string with s[0] capitalized and the rest (s[1:]) lowercased. Capitalize(...) accepts no arguments.

```
s = "hello world."
print(s.capitalize()) # `Hello world.`
```

15.2 casefold

Returns a copy of the string with a stricter lower case enforcing; by default *lower()* does not account for various code points, such as *ß*, using *casefold()* is more aggressive and will convert such characters to *ss*. *casefold()* takes no arguments.

```
s = "foo ß"
print(s.lower()) # `foo ß`
print(s.casefold()) # `foo ss`
```

15.3 center

Returns a copy of the string centered with a length of *width*, padded by a *fillchar*. The argument width is required, fillchar is optional and by default is the ASCII space (code point: *U+0020*). By default no keyword args are supported, width and fillchar are positional only. In the event that *width* is less than the *len(s)* then the original *s* string is returned.

```
s = "center me"
print(s.center(20, "#"))  # `#####center me######`
x = "example"
print(s.center(30))  # `           example            `

# short width
s = "too short"
print(s.center(3)) # `too short`
```

15.4 count

Returns the number of non overlapping occurrences of *sub* in the string. *count(sub, [,start[, end]])* can accept an optional *start* and *end* argument to perform the check on a particular range, interpreted in *slice* notation e.g *s[start:end]*. *count(...)* accepts no keyword and only positional arguments.

```
s = "example of examples"
target = "example"
print(s.count(target))  # 2
print(s.count(target, 0, len(target)) # 1

# Overlapping example
s = "abababab"
print(s.count("ab")) # 3 ( non overlapping)
print(s.count("aba")) # 1 (You may think, `2` but infact it's `1`)
```

15.5 encode

Returns a new *bytes* version of the encoded string. As of recently *encode(encoding=..., errors=...)* via keyword arguments is supported. By default encoding is *utf-8* being the most prevalent in present times and errors is *strict* however additional options are: (*ignore*, *replace*, *xmlcharrefreplace* and *backslashreplace*). User defined additional ones can be provided if they are registered via the codec module using *codecs.register_error(...)*. By default encoding errors raise a *UnicodeError*. Note: Python as a language is unicode aware, this is demonstrated in the examples below:

```
s = "foo  bar"
print(type(s.encode()) # `bytes`
print(s.encode()) # b'foo \xf0\x9f\x98\x8a bar'
print(len(s.encode())) # 12 bytes
# `foo` as normal (3 bytes) + suffix whitespace (1 byte) (4)
# Emoji  is 4 byte UTF-8 (U+1F60A) (4 bytes)
# `bar as normal` (3 bytes) + prefix whitespace (1 byte) (4)
# 3 + 1 + 4 + 1 + 3 (12 bytes).

# Keyword args are supported.
s = "foobar"
print(s.encode(encoding="utf-8", errors="strict")) # b'foobar' (6 bytes).
```

15.6 endswith

Check if a string is suffixed with a particular substring. Optional *start* and *end* arguments can be provided which again are interpreted in *slice* notation. The suffix parameter can also be a tuple of various suffixes to look for. *endswith(...)* does not support keyword arguments, positional only.

```
s = "language:html"
print(s.endswith(("html", "php"))  # True
print(s.endswith("guage", 0, 8))  # True
```

15.7 expandtabs

CHAPTER

SIXTEEN

INDICES AND TABLES

- genindex
- modindex
- search