Python Data Structures and Comprehensions

By Arianne Dee

Survey – Multi-choice

- How much Python do you know?
  - Basically nothing
  - The basics (conditionals, loops)
  - The common built-in data structures (lists, dicts)
  - The less common built-in data structures (tuples, sets)
  - How to write classes
  - How to install external libraries

Survey – multi-choice

- What topics are you interested in?
  - Determine when to use what data structure
  - Built-in structures (lists, dicts, tuples, sets)
  - Built-in modules (collections)
  - NumPy and Pandas for data analysis
  - Basic comprehensions
  - Advanced comprehensions
  - Other – say in chat

About me

- Instructor for O'Reilly:
  - 8 Live Trainings
  - 4 Videos
  - Interactive Lab: Python Foundations

- 10+ years of software development, mostly in Python and Django web development
  - games, startups, consulting, agencies, freelance, education








Course goals

- Give you a broad overview of different data structures available
- Introduce you to simple and more complex comprehensions for creating lists, dicts and sets
- Look at two of the most common data structures for data analysis: Numpy **`ndarrays`** and Pandas **Series** and **`DataFrames`**

Course outline

- 0:00 – Intro and setup
- 0:15 – **Built-in data structures**
  - Break (10 min)
- 1:45 – **Comprehensions**
  - Break (10 min)
- 2:30 – **Built-in module data structures**
  - Break (10 min)
- 3:30 – **NumPy** and **Pandas**
- 4:55 – Course wrap-up

Questions and Breaks

- Use attendee chat throughout class
  - Off-topic questions go in Q&A widget
- 3 x 10 min breaks
  - Step away or work through code
  - I'll answer questions in the Q&A widget
- Email more in-depth questions at arianne.dee.studios@gmail.com

Run the code in your browser

Interactive site:
https://ariannedee.github.io/python-data-structs/lab/index.html

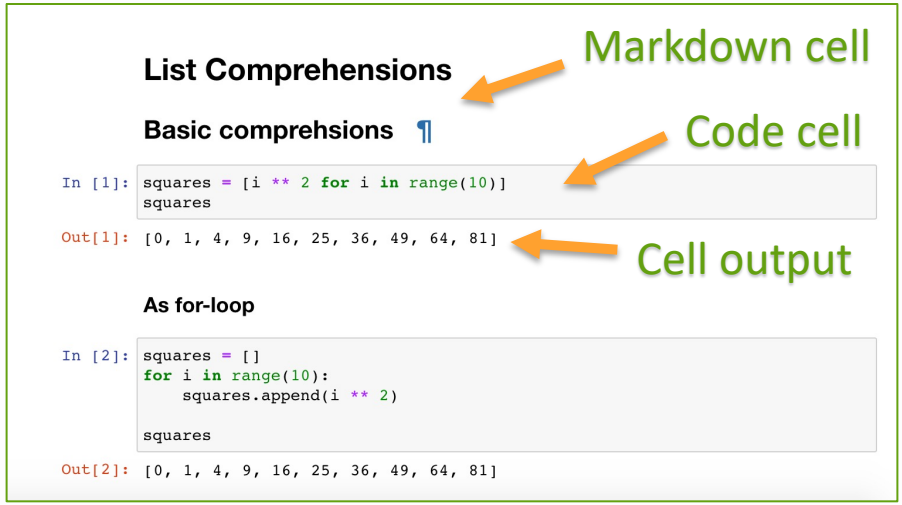
See the source code at:
https://github.com/ariannedee/python-data-structs

Local setup (optional)

Local setup - optional

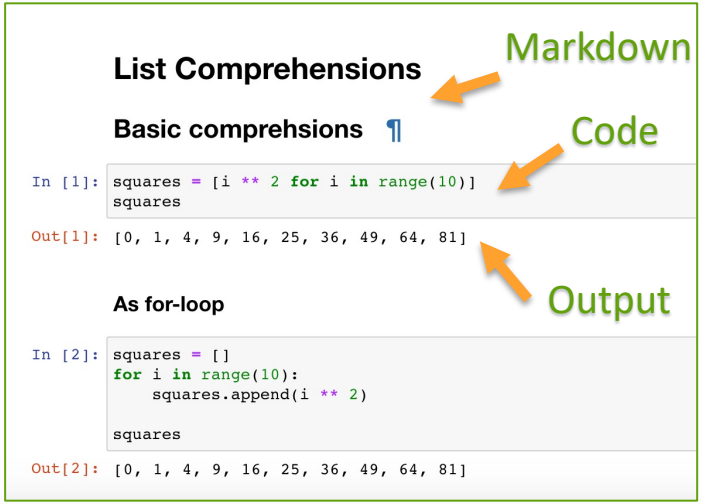
- Use **Python 3.7 or higher**
  - https://www.python.org/downloads/
- Get the **example code**
  - https://github.com/ariannedee/python-data-structures
- Install **external packages**
  - `$ pip install jupyter numpy pandas`
- Run Jupyter
  - `$ jupyter-lab`

About notebooks



Notebooks

- Code in code blocks called **cells**
- Cells can be run, edited and rerun
- Result of a run and print() are displayed below cell
- Formatted text cells use Markdown format
- Popular in data science to map process and display plots



Course project

Data collection

- Fill out a survey of **which programming languages you know**
  - Survey link: https://bit.ly/python-data-structures-survey
  - "**Know**" ≈ you could use loops, conditionals and print to console

In class

- We will analyze the data to determine
  - The full set of languages that attendees know
  - The listed languages that nobody knows
  - Rank languages from most known to least

- Bonus content:
  - Use Pandas to rank languages per age group
  - Compare the results of this training with the 2022 Stack Overflow Developer Survey results

Survey results data

- I will upload a CSV of the survey results here
  - https://github.com/ariannedee/python-data-structures/tree/main/project/data

- You can pull the changes using Git

- Or download the file and paste into the project/data/ folder

Built-in data structures

Lists, dicts, tuples and sets

Sequences

- A container object that supports efficient element access using integer **indices**
- `s[0]` returns the first element

Documentation

Lists

- Ordered
- Can contain duplicate values
- Mutable
- Get values by integer **index**

```
primes = [2, 3, 5, 7, 11]
```

Documentation

List functionality

- You **can**:
  - Store any type of data in values
  - Access 1st, nth and last elements
  - Can get a subset (slice)
  - Sort the objects based on different criteria

- You **cannot**:
  - Use as a **`dict`** key

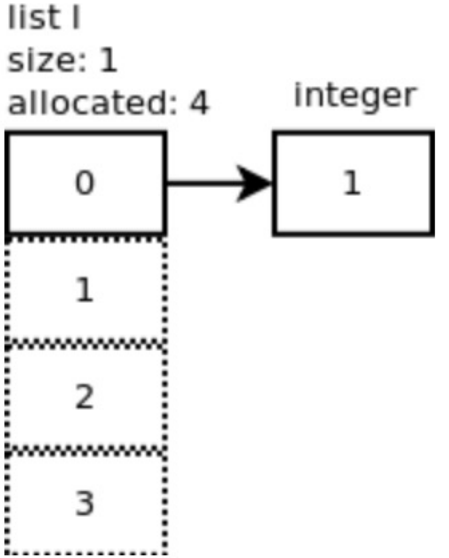
Why use lists?

- Represent a collection of similar objects
- Care about the order
- Need to include duplicates
- Don't need efficient access individual items besides index

- Examples:
  - Countries in the United Nations
  - Students registered for a class
  - Top 10 movies of the year
  - Prime numbers under 100
  - Temperature recordings per hour

How **lists** are implemented

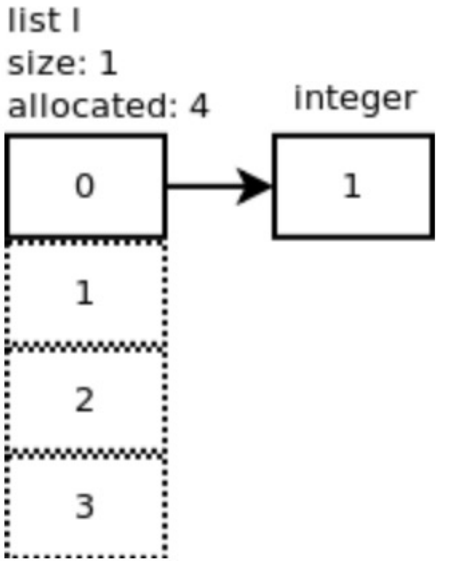
- In C Python:
  - Size of the list
  - Number of slots allocated
  - Array of pointers to objects



How **lists** are implemented cont.

- Appending an item will:
  - Add 1 to the size
  - Store the new value
  - Resize the pointer array to increase allocated slots if there are no more
  - Set the next available slot to point to the new value

- Blog post with more examples
  - https://www.laurentluce.com/posts/python-list-implementation/



List – time complexity

- Fast:
  - Get item by index
  - Update an item
  - Append/pop item
  - Determine length

- Slow:
  - Inserting at an index (especially closer to 0)
  - Finding an item by value
  - Check value containment
  - Deleting an item (especially closer to 0)

Full list of actions

Tuples

- Ordered
- Can contain duplicate values
- **Immutable**
- Get values by integer **index**

```
coords = (16.77, -3.00)
```

Documentation

Tuple features

- You can:
  - Store any type of data in values
  - Access 1st, nth and last elements
  - Can get a subset (slice)
  - Use as a **`dict`** key
  - Pack and unpack
    - `coords = 16.77, -3.00` - pack multiple items into one variable
    - `lat, lon = coords` - unpack an item to multiple variables

- You cannot:
  - Sort the objects based on different criteria
  - Alter the contents, size, or order

Why use **tuple**?

- Represent a set of objects where the order matters
  - e.g. `person = ('Arianne', 'Dee', 35)`
    - `person[0]` is the first name
    - `person[1]` is the last name
    - `person[2]` is the age

- Return multiple values from a function
- Represent items that should be constant (coordinates)
- Use as dictionary keys
- Swap variable names

tuple vs list?

- **`list`** is a bucket of items
  - contains homogeneous data
  - reordering doesn't significantly change meaning

- **`tuple`** is an item with multiple parts
  - contains heterogeneous data
  - reordering does significantly change meaning

```
blue = 0, 0, 255
colours = ['red', 'green', blue]
```

Other built-in **sequence** types

- `range` – sequence of integers represented by start, stop, step
- `str` – immutable array of characters
- `bytes` – immutable array of bytes
- `bytearray` – mutable array of bytes
- `memoryview` – array of pointers to bytes (used in image processing)

Mapping

- A container object that supports arbitrary key lookups

https://docs.python.org/3/glossary.html#term-mapping

Dicts

- Cannot contain duplicate keys
- Mutable
- Get values by **key**
- Ordered (as of Python 3.6)

```
fruit = {'a': 'apple', 'b': 'banana'}
```

Documentation

Dictionary features

- You can:
  - Store any type of data in values
  - Access values by key
  - Add, remove or update items (keys or values)

- You cannot:
  - Use mutable data structures as keys (must be **hashable**)
    - **tuples** and **frozensets** are okay
  - Have duplicate keys
    - **Note**: True, 1 and 1.0 represent the same key
  - Rearrange order of items

Why use **dict**?

- Map keys to values:
  - Map country names to capital cities
  - Lookup definition, synonyms and antonyms of a word
  - Map data of expensive calculations, like integers mapped to their list of factors

- Represent objects:
  - User with keys 'Name', 'Email', 'Username'

Dict – time complexity

- Fast:
  - Get item by key
  - Add, update or delete a value by key
  - Check key containment
  - Determine length

- Slow:
  - Finding an item by value
  - Check value containment

Full list of actions

JSON

- **J**ava**S**cript **O**bject **N**otation
- A common format for storing data
  - **Objects** – key/value data that represent objects
    - {key: value}
  - **Arrays** – for a sequence of objects/attributes
    - [value1, ..., valuen]

- JSON is normally a string that can be easily converted to Python **lists** and **dicts**

Converting from JSON to Python

```
import json

people_json = '[{"id": 1, "name": "Anya"},{"id": 2,
"name": "Bijan"}]'

people = json.loads(people_json)
print(type(people))     # <class 'list'>
print(type(people[0]))  # <class 'dict'>
```

Sets

- Unordered
- **Cannot contain duplicates**
- Mutable
- Cannot get individual values

Documentation

Set features

- You **can**:
  - Compare set contents quickly
  - Perform set operations (unions, difference, etc)
  - Add/remove items

- You **cannot**:
  - Sort items
  - Access individual items by index or key (only looping)
  - Insert unhashable items like lists, dicts or sets
    - **tuples** and **frozensets** are okay

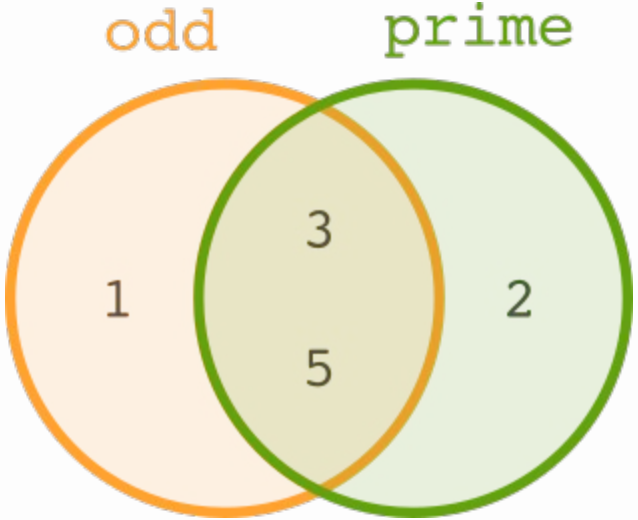
Why use **set**?

- Need to quickly check if an item is in a set
- Your algorithm needs to perform set operations
- Want to find unique values

Set operation overview

Given 2 sets:

```
odd = {1, 3, 5}
prime = {2, 3, 5}
```



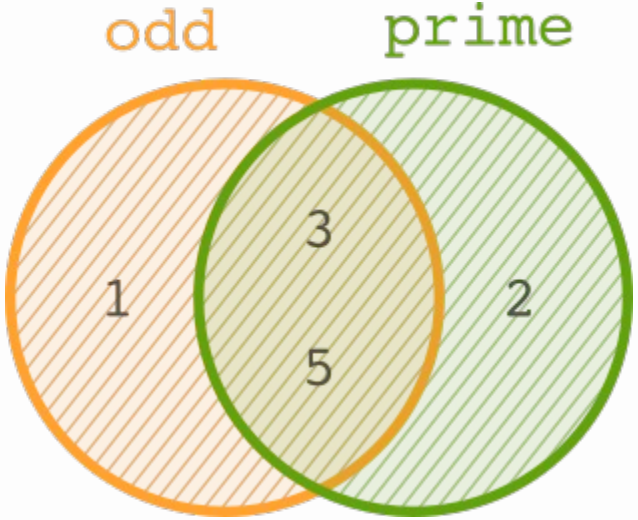
Set operation overview - Union

Union of 2 sets:

```
s1.union(s2)
s1 | s2
```

Result:

```
{1, 2, 3, 5}
```



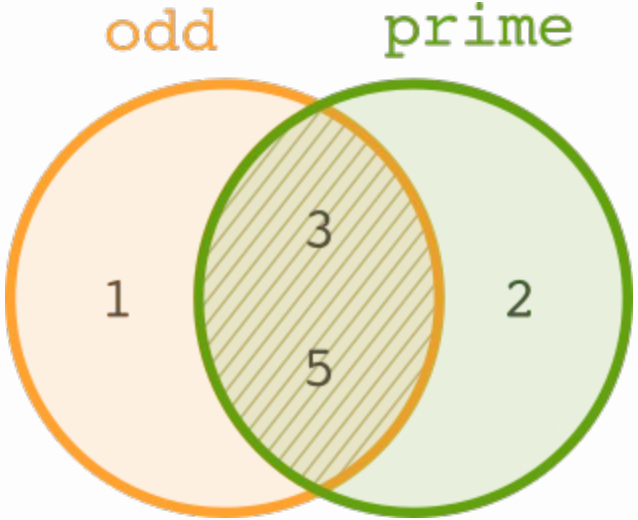
Set operation overview - Intersection

Intersection of 2 sets:

```
s1.intersection(s2)
s1 & s2
```

Result:

```
{3, 5}
```



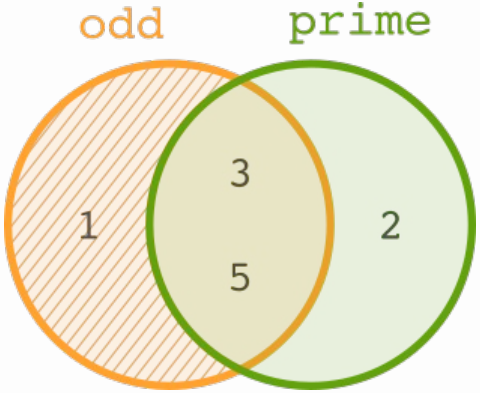
Set operation overview - Difference

Difference of s1 – s2:

```
s1.difference(s2)
s1 - s2
```

Result:

```
{1}
```

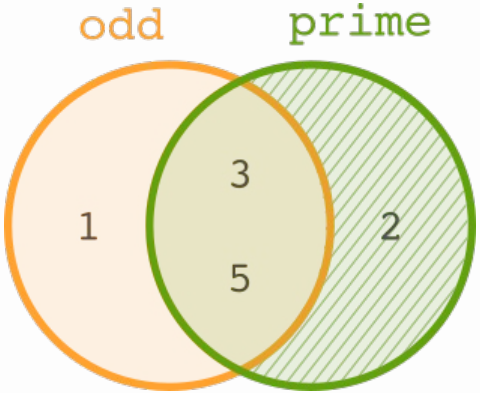


Difference of s2 - s1:

```
s2.difference(s1)
s2 - s1
```

Result:

```
{2}
```



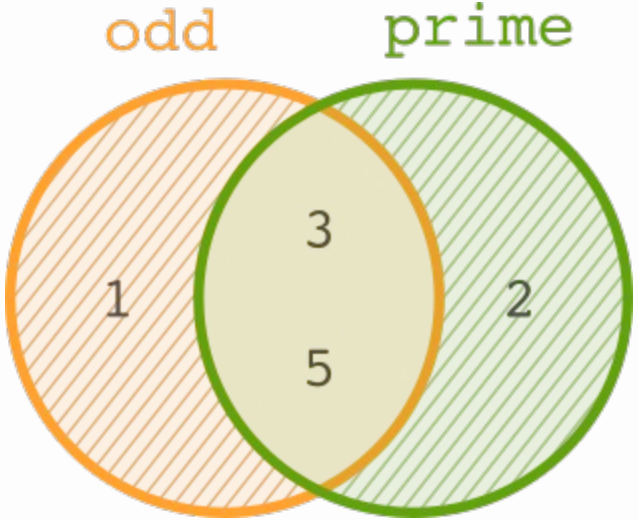
Set operation overview – Symmetric difference

Symmetric difference:

```
s1.symmetric_difference(s2)
s1 ^ s2
```

Result:

```
{1, 2}
```



Set – Time complexity

- Fast:
  - Add and remove a value
  - Check value containment
  - Determine length

- Impossible:
  - Get individual item
  - Update an item

Full list of actions

Other built-in set types

- `frozenset` – immutable set

Resources – Built-in data structures

- [O'Reilly interactive lab] **Data Structures**
  - https://learning.oreilly.com/scenarios/hands-on-python-foundations/9780137904648X004/
- [Real Python] **Lists and Tuples in Python**
  - https://realpython.com/python-lists-tuples/
- [Real Python] **Dictionaries in Python**
  - https://realpython.com/python-dicts/
- [Real Python] **Sets in Python**
  - https://realpython.com/python-sets/

Project – pt. 1

- Update **`survey_analysis.ipynb`**
- Use `list`, `dict`, `tuple` and/or `set`

Comprehensions

One-line data structure creation

List comprehensions

```
squares = [i ** 2 for i in range(5)]
```

Result: `[0, 1, 4, 9, 16]`

Creating lists with `for` loops

List are often created using a **`for`** loop.

1. Create a list
2. Iterate over a sequence
3. Append items to the list

```
squares = []
for i in range(5):
    squares.append(i ** 2)
```

- 3 lines of code
- Requires a lot of mental work

Creating lists with `map()`

You can also create lists in a "*functional*" way using **`map()`** and **`filter()`**.

1. Define a function
2. Call **map**/**filter** with the function and a sequence
3. Convert the resulting map object to a list

```
def square(x):
    return x**2
squares = list(map(square, range(5)))
```

Or using a lambda expression:

```
list(map(lambda x: x**2, range(5)))
```

Creating lists with comprehensions

We can use list comprehensions to create a list in one line

```
squares = [i ** 2 for i in range(5)]
```

It generalizes to:

```
new_list = [expression for member in iterable]
```

1. **expression:** what do you want to add to the list
2. **member:** an element in the sequence
3. **iterable:** an object that can return its elements one at a time

Why list comprehensions?

- Easier to read and understand
- Signals intent of creating a new list
- More succinct
- More *"Pythonic"*

- Compared to **for** loops:
  - Focus on **what** is in the list instead of **how** to create it
  - Less mental work to map actions

- Compared to `map()` and `filter()`:
  - Don't need to remember which function to use or argument order
  - More explicit
  - Don't need to name functions or use lambdas

Using conditionals

```
evens = [i for i in range(5) if i % 2 == 0]
```

Result: `[0, 2, 4]`

With condition

Generalized:

```
new_list = [expression for member in iterable if condition]
```

1. **expression:** what do you want to add to the list
2. **member:** an element in the sequence
3. **iterable:** an object that can return its elements one at a time
4. **condition:** how to filter members to include in new list

Using conditionals

- As a **for** loop:

```
evens = []
for i in range(5):
    if i % 2 == 0:
        evens.append(i)
```

- Using **filter()**:

```
evens = list(filter(lambda x: x % 2 == 0, range(5)))
```

- Using list comprehensions:

```
evens = [i for i in range(5) if i % 2 == 0]
```

Using ternaries

```
is_even = [True if i%2 == 0 else False for i in range(5)]
```

Result: `[True, False, True, False, True]`

Ternary operator

Without ternary operator:

```
if <condition>:
    x = a
else:
    x = b
```

With ternary operator:

```
x = a if <condition> else b
```

Example:

```
if i % 2 == 1:
    even = True
else:
    even = False
```

```
even = True if i % 2 == 0 else False
```

Using ternaries

- As a **for** loop:

```
is_even = []
for i in range(5):
    if i%2 == 0:
        is_even.append(True)
    else:
        is_even.append(False)
```

- Using **map()**:

```
evens = list(map(lambda x: x % 2 == 0, range(5)))
```

- Using list comprehensions:

```
is_even = [True if i%2 == 0 else False for i in range(5)]
```

Nested comprehensions

nested lists

```
coords = [
    [(x, y) for y in range(3)]
    for x in range(3)
]
```

Result:

```
[[(0, 0), (0, 1), (0, 2)],
 [(1, 0), (1, 1), (1, 2)],
 [(2, 0), (2, 1), (2, 2)]]
```

Nesting – nested lists

- As a **for** loop:

```
coords = []
for x in range(3):
    row = []
    for y in range(3):
        row.append((x, y))
    coords.append(row)
```

- Using **map()**:

```
list(map(lambda y: list(map(lambda x: (x, y), range(3))), range(3)))
```

- Using list comprehensions:

```
coords = [[(x, y) for y in range(3)] for x in range(3)]
```

Nested comprehensions

flattened list

```
coords = [
    (x, y) for x in range(3)
    for y in range(3)
]
```

Result:

```
[(0, 0), (0, 1), (0, 2),
 (1, 0), (1, 1), (1, 2),
 (2, 0), (2, 1), (2, 2)]
```

Nesting

- As a **for** loop:

```
coords = []
for x in range(3):
    for y in range(3):
        coords.append((x, y))
```

- Using list comprehensions:

```
coords = [(x, y) for x in range(3) for y in range(3)]
```

- Great for flattening nested sequences

Complex comprehensions

```
num_lists = [[7, 2], [6], [1, 3]]
small_nums = [num
              for nums in num_lists
              for num in nums
              if num < 5]
```

Result: `[2, 1, 3]`

Complex comprehensions

- As a **for** loop:

```
small_nums = []
for nums in num_lists:
    for num in nums:
        if num < 5:
            small_nums.append(num)
```

- Using list comprehensions:

```
small_nums = [num
              for nums in num_lists
              for num in nums
              if num < 5]
```

Complex comprehensions

```
num_lists = [[7, 2], [6], [1, 3]]
a_list = [num if num > 3 else -num
          for nums in num_lists
          if len(nums) > 1
          for num in nums
          if num % 2 == 1]
```

Result: `[7, -1, -3]`

Complex comprehensions

- Can have:
  - a complicated expression
  - nested for loops
  - a filter condition on each level of nesting
- They can be:
  - hard to logic through
  - difficult to understand, especially for novices
- Does it make the code easier to read for most of the programmers who will read it?

Dict comprehensions

```
powers_of_2 = {i: 2**i for i in range(2, 9, 2)}
```

Result: `{2: 4, 4: 16, 6: 64, 8: 256}`

Dict comprehensions

- As a **for** loop:

```
powers_of_2 = {}
for i in range(2, 9, 2):
    powers_of_2[i] = 2 ** i
```

- Using dict comprehensions:

```
powers_of_2 = {exp: 2**exp for exp in range(2, 9, 2)}
```

Set comprehensions

```
num_list = [1, 2, 3, 3, 8]
num_set = {num for num in num_list}
```

Result: `{1, 2, 3, 8}`

It is equivalent to `set(num_list)` but conditionals and nesting to make it more useful

Nested set comprehensions

```
words = ['mom', 'dad', 'ham']
chars = {char for word in words for char in word}
```

Result: `{'d', 'm', 'a', 'o', 'h'}`

Nested set comprehensions

```
words = ['Mom', 'Dad', 'Ham']
consonants = {
    char.lower()
    for word in words
    for char in word.lower()
    if char not in 'aeiou'
}
```

Result: {'d', 'm', 'h'}

Nested set comprehensions

```
words = ['mom', 'dad', 'add']
num_set = {
    frozenset({char for char in word})
    for word in words
}
```

Result: {

```
    frozenset({'m', 'o'}),
    frozenset({'d', 'a'})
}
```

Nested set comprehensions

- Set contents must be hashable
  - hashable ≈ immutable

- To include a set of sets, use `frozenset()` to make the inner sets immutable and hashable

- **Note**: immutable dictionaries are not yet a part of Python, but may be in the future (see PEP 603 – `frozenmap()`)

Are there tuple comprehensions?

- No
- Since tuples are immutable, you cannot create them with comprehensions
- Using parentheses `( )` with the comprehension syntax creates a generator

Generators

- Like lazily-evaluated lists
  - **Can** be looped/iterated over
  - **Cannot** get item by index
  - **Cannot** call `len()` on it

- Doesn't store all values in memory at one time
  - Not technically a data structure

- Created using:
  - A function that uses the **`yield`** keyword
  - A generator comprehension/expression

Generator comprehensions

```
squares = (i ** 2 for i in range(10**10))
```

Result: `<generator object <genexpr> at 0x1...>`

`next(squares)` returns `0`

`next(squares)` returns `1`

`next(squares)` returns `4`

```
for square in squares:
    print(square)
```

will loop 10^{10} times

will print squares up to $(10^{10}-1)^2$

Generator comprehensions

```
has_evens = any(True for num in nums if num % 2 == 0)
```

Result:

`True` if any of the numbers in nums are even

- Returns on the first match

`False` if none of the numbers in nums are even

- Evaluates all nums

Generator comprehensions

- Can use a generator expression instead of list comprehension in any functions that take an iterable:
  - `any()`, `all()`
    - will return once a True (any) or False (all) found
  - `min()`, `max()`, `sum()`
    - No need to create a new list first
  - `tuple()` to create a tuple from a comprehension

- https://docs.python.org/3/tutorial/classes.html#generators

Resources – Comprehensions

- [Real Python] **List Comprehensions**
  - https://realpython.com/list-comprehension-python/
- [Medium] **Examples to master list comprehensions**
  - https://towardsdatascience.com/11-examples-to-master-python-list-comprehensions-33c681b56212
- [Real Python] **Using Generators**
  - https://realpython.com/introduction-to-python-generators

Project – pt. 2

- Update **`survey_analysis.ipynb`**
- Use comprehensions

Module data structures

The collections module

Sequences

- `array.array`
- `collections.namedtuple`

array.array

```
import array
arr = array.array("f", (1.0, 1.5, 2.0, 2.5))
```

- Must declare the type of data it holds
  - `'f'` – float
  - `'i'` – integer
  - `'u'` – Unicode character
- Uses less memory than **`list`**
- Supports most of the same methods as **`list`**

`array.array` type codes

Type code	C Type	Python Type	Minimum size in bytes
`'b'`	signed char	int	1
`'B'`	unsigned char	int	1
`'u'`	wchar_t	Unicode character	2
`'h'`	signed short	int	2
`'H'`	unsigned short	int	2
`'i'`	signed int	int	2
`'I'`	unsigned int	int	2
`'l'`	signed long	int	4
`'L'`	unsigned long	int	4
`'q'`	signed long long	int	8
`'Q'`	unsigned long long	int	8
`'f'`	float	float	4
`'d'`	double	float	8

collections.namedtuple

- Useful for creating lightweight data objects
- Gives tuple values a name, accessed as attribute

```
from collections import namedtuple

coords = namedtuple("Location", "name lat lon")

c = coords('London', 42.990, -81.243)

assert c.name == 'London'
assert c[0] == 'London'
assert c == ('London', 42.990, -81.243)
```

Mappings

- `collections.OrderedDict`
- `collections.defaultdict`
- `collections.ChainMap`

collections.OrderedDict

- Cannot access item by index (only key)
- Can reorder items by moving to end or beginning
- Can pop from beginning or end with `.popitem()`

```
from collections import OrderedDict

d = OrderedDict({1: 'a', 2: 'b'})
d[0] = 'z'
d.move_to_end(2, last=False)
```

Result:

```
{2: 'b', 1: 'a', 0: 'z'}
```

When to use OrderedDict over dict?

- Clearly state that the order is important
- Need functionality of `.move_to_end()`
- Need to test equality based on order
- Need it to be backwards compatible (Python < 3.6)

collections.defaultdict

- **dict** with a "default" behaviour if key is not found
- Pass a callable in the constructor (no params)
  - function – calls function
  - class – Initializes class

```
from collections import defaultdict

letter_countries = defaultdict(list)
letter_countries['B'].append('Bolivia')

word_counts = defaultdict(lambda: 0)
word_counts['the'] += 1
```

collections.ChainMap

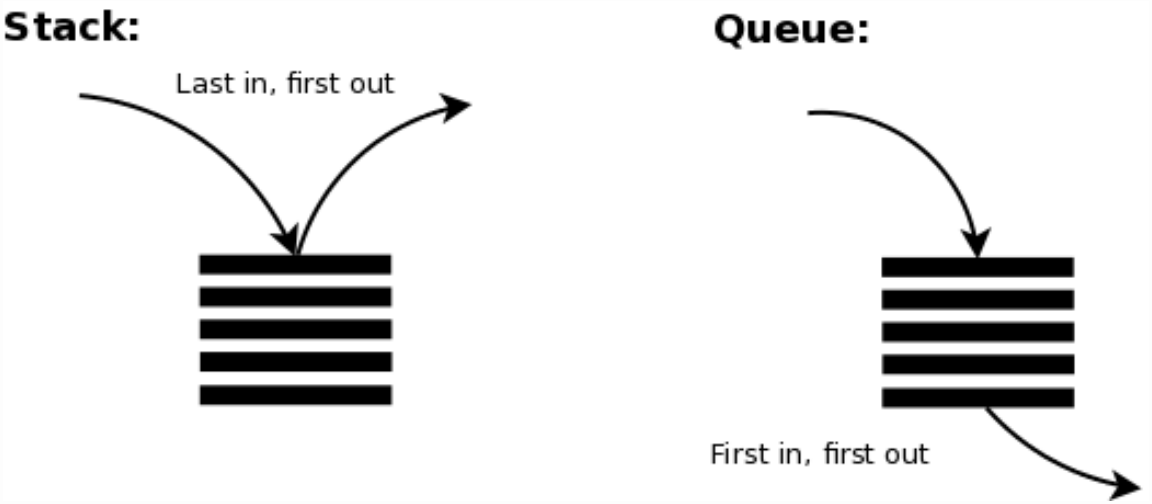
- Search through multiple **dicts** at once
- Updates only affect the first **`dict`**

```
from collections import ChainMap

dict1 = {"one": 1, "two": 2}
dict2 = {"three": 3, "four": 4}
chain = ChainMap(dict1, dict2)

assert chain["three"] == 3
```

Stacks and queues

- **`list`**
- `collections.`**`deque`**

Stack vs Queue



Stack

- Can use builtin **`list`**
  - `.append(x)` – add to end of stack
  - `.pop(x)` – removes and returns last element

- Do not add or remove from the front (index 0)

collections.deque

- Double-ended queue
- Quickly remove or append items from either end
- Implemented as a doubly-linked list

Node Structure of Doubly Linked List

collections.deque

- To use:
  - `.append(x)` and `.pop(x)` to add/remove from end
  - `.appendleft(x)` and `.popleft(x)` to add/remove from front

- **Stack**
  - `.append(x)`, `.pop(x)`

- **Queue**
  - `.append(x)`, `.popleft(x)` or
  - `.appendleft(x)`, `.pop(x)`

Counters

- `collections.Counter`

collections.Counter

- Keeps track of how many times an item is in a set
  - Also known as a multiset or bag
- Can update the set with a sequence (list, tuple, set)
  - Each item's count increases by 1
- Can update with a mapping (**`dict`**) of `{item: n}`
  - Each item's count is increased by n

collections.Counter

```
from collections import Counter

coins = Counter()
coins.update(['silver', 'silver', 'gold'])
coins.update({'copper': 4, 'silver': 2})
```

Result:

```
Counter({'copper': 4, 'silver': 4, 'gold': 1})
```

Custom classes

- Sometimes you need to build it yourself

Custom data structure classes

- Lots of ways to create custom data structures
- Need to know how to:
  - Create classes
  - Implement dunder methods
    - e.g. `.__getitem__()`
  - Add/override methods
    - `.append()`, `.push()`

Useful dunder methods

Dunder method	Description	Example
`__init__`	Initializer	`m = MyList([1, 2, 3])`
`__getitem__`	Retrieve value	`m[1]`
`__setitem__`	Set value	`m[1] = 100`
`__len__`	Get length of list	`len(m)`
`__iter__`	Support **for** loops	`for val in m: ...`
`__contains__`	Check contents with **in**	`if val in m: ...`
`__eq__`	Check equality with **==**	`m == [1, 2, 3]`
`__hash__`	Return integer so obj can be used as a **dict** key or in a **set**	`my_dict[m] = val`

How to create custom classes

1. Build your own from scratch
   - Inherit from **`object`**

2. Use a Data Class
   - Decorate with `dataclasses.dataclass`

3. Inherit from an existing data structure
   - Overwrite/add methods
   - data is accessed through **`self`**

4. Implement an abstract base class
   - E.g. inherit from **`collections.abc.MutableSequence`**

5. Inherit from a User defined class from collections module
   - UserDict, UserList, UserString
   - Underlying data accessed from `.data`

Custom classes

- `dataclasses.dataclass`
- `collections.UserDict`
- `collections.UserList`
- `collections.UserString`

Data Classes

`UserDict`, `UserList`, `UserString`

- Inherit from these to create custom dicts, lists and strings
- Access underlying dict/list/string via `.data`
- Override/implement any methods or dunder methods you want to add

Example **UserList**

```
from collections import UserList

class MyList(UserList):
    def upper(self):
        new_data = []
        for val in self.data:
            if isinstance(val, str):
                new_data.append(val.upper())
            else:
                new_data.append(val)
        return MyList(new_data)

l = MyList([1, 'a', 2, 'b'])
print(l.upper())
```

Prints:

```
[1, 'A', 2, 'B']
```

Resources – Module data structures

- [Real Python] **Common Python Data Structures**
  - https://realpython.com/python-data-structures/

- [Real Python] **Custom Python Lists: Inheriting From list vs UserList**
  - https://realpython.com/inherit-python-list/

Project – pt. 3

- Update **`survey_analysis.ipynb`**
- Use **`collections`**

NumPy and Pandas

Multi-dimensional arrays

NumPy

- **Num**erical **Py**thon

- **`ndarray`**: n-dimensional array

NumPy – `ndarray`

- Multi-dimensional arrays
- All same type of data
  - **`int`**, **`float`**, **`bool`**, **`str`**
  - numbers can be different sizes (e.g. int8, float128)

- Uses 0-indexed axis
- In a 2d matrix, axis 0 is vertical and axis 1 is horizontal

```
[[0, 1, 2],
 [3, 4, 5]]
```

NumPy

```
$ pip install numpy
```

```
import numpy as np

as_list = [1, 2, 3, 4]
as_array = np.array(as_list)
```

Result:

```
[1 2 3 4]
```

NumPy

- Lots of built-in operations for matrices
- Faster than using lists
- Less looping/iterating
- Easier to read
- Large community, used widely, reliable
  - Less buggy than writing your own operations
- Basis of other data science libraries:
  - Matplotlib (plotting), Pandas (data analysis), scikit-learn (machine learning)

NumPy – Creating arrays

- From lists or tuples
  - `np.array(nested_list)`
- Empty array
  - `np.empty(shape)`
- All 1's or 0's
  - `np.zeroes(shape)`, `np.ones(shape)`
- Range of data
  - `np.arange(start, stop, step)`: like Python's `range()`
  - `np.linspace(start, stop, num)`: specify the number of items

Reading arrays

- Prints similar to nested lists, with:
  - the last axis is printed from left to right,
  - the second-to-last is printed from top to bottom,
  - the rest are also printed from top to bottom, with each slice separated from the next by an empty line.

`ndarray` – Arithmetic operations

- Operations are performed on each element
  - Returned as a new array
  - `[1, 2, 3] ** 2` returns `[1, 4, 9]`
- Augmented assignment (e.g. +=) modifies the array in-place
- You can operate on an array if the second value is:
  - An array of the same shape
  - A single number (vector)
  - An array where each axis has the same size or is 1
    - See broadcasting rules

`ndarray` – Methods

- Lots of array methods available
  - e.g. `.round()`, `.dot()`, `.transpose()`, `.clip()`

- Some, like `.sum()`, `.max()`, `.round()`, `.any()` work on:
  - the entire array (if axis is None)
  - one axis (if axis is an integer)
  - a set of axes (if axis is a tuple)

- Some convert the array into another format:
  - `.astype()`, `.tolist()`, `.tofile()`, `.dump()`

`ndarray` – Indexing and slicing

- On 1d arrays, works similar to lists
  - `a[0:5:2]` returns items at indices `[0, 2, 4]`

- On multidimensional arrays, you can pass multiple indices
  - Getting an item: `a[2, 3]` is similar to `a[2][3]`
  - Getting a slice: `a[:2, 3]` returns an array of the 4th items (3) in the first two rows (:2)
  - If you provide fewer indices than the array has, it fills any extra in with `:`

Example

- Curve grades so the average score is 80
- Look ma, no loops!

Numpy example

```
import numpy as np
CURVE_CENTER = 80
grades = np.array([72, 35, 64, 88, 51, 90, 74, 12])

def curve(grades):
    average = grades.mean()
    change = CURVE_CENTER - average
    new_grades = grades + change
    return np.clip(new_grades, grades, 100)

curve(grades)
```

Example from https://realpython.com/numpy-tutorial/

`ndarray` vs lists/loops

- Can be clearer to read
  - Focus on **what** to do not **how** to do it
- Much faster for larger datasets
  - Clear benefits with > 1M data points
  - See example 20 for code comparison

Resources – Numpy

- [Numpy] **NumPy quickstart**
  - https://numpy.org/doc/stable/user/quickstart.html
- [Numpy] **The absolute basics for beginners**
  - https://numpy.org/doc/stable/user/absolute_beginners.html
- [Real Python] **NumPy Tutorial**
  - https://realpython.com/numpy-tutorial/

Pandas

- **Pa**nel **Da**ta *or* Pytho**n** **D**ata **A**nalysi**s**

Pandas

- Lots of tools for data science/data analysis and machine learning tasks
- Built on top of NumPy
- Makes it simple to do many of the time consuming, repetitive tasks associated with working with data

Pandas tasks

- Data cleansing
- Data filling
- Data normalization
- Merges and joins
- Data visualization
- Statistical analysis
- Data inspection
- Loading and saving data

Pandas

```
$ pip install pandas
```

```
import numpy as np
import pandas as pd
```

Series

- 1d array with labels

Pandas – Series

- 1d array of data with labels
- Set of labels is called the **index**

```
import pandas as pd

s = pd.Series({"a": 0.0, "b": 1.0, "c": 2.0})
```

Result:

```
a    0.0
b    1.0
c    2.0
dtype: float64
```

Pandas – Series

- A **Series** is like an **ndarray** and a **dict**

```
s = pd.Series({"a": 0.0, "b": 1.0, "c": 2.0})
```

- `s[:2]` returns a slice, including the index

```
a    0.0
b    1.0
dtype: float64
```

- `s['b']` or `s.get('b')` returns the value 1.0

Creating **Series**

```
s = pd.Series(data, index=index)
```

- **data** can be:
  - a Python **dict**
  - an **ndarray**
  - a scalar value (like 5)

Create `Series` from `ndarray`

From **ndarray**

- **index** must be the same length as the **data**
  - `pd.Series(np.array([1, 2, 3]), index=["c", "a", "b"])`
    - index is `["c", "a", "b"]` and values is `[1, 2, 3]`

- If **index** is not passed, it defaults to integers [0, ..., len - 1]
  - `pd.Series(np.array([1, 2, 3]))`
    - index is `[0, 1, 2]` and values is `[1, 2, 3]`

- An index does not have to be unique

Create `Series` from `dict`

From **dict**

- If **index** is not passed, it is retrieved from the keys
  - `pd.Series({"b": 1, "a": 0, "c": 2})`
    - index is `["b", "a", "c"]` and values is `[1, 0, 2]`

- If **index** is passed, it will pull corresponding values from **data**
  - `pd.Series({"b": 1, "a": 0, "c": 2}, index=["b", "d", "c"])`
    - index is `["b", "d", "c"]` and values is `[1, NaN , 2]`

Create `Series` from scalar

- From scalar
  - an **index** must be provided
  - **data** is the value set for every item in index
    - `pd.Series(2, index=["a", "b", "c"])`
      - index is `["a", "b", "c"]` and values is `[2, 2, 2]`

`Series` is like `ndarray`

- Indexing and slicing is like **`ndarray`**
  - Slicing includes the index

- **`Series`** is a valid input to most NumPy functions that expect an **`ndarray`**

- **`dtype`** is often a NumPy **`dtype`** but can be a Pandas **`ExtensionDtype`**

- To get an **`ndarray`**, use `.to_numpy()`

`Series` is like `dict`

- You can get and set values by index
- Use `.get()` to avoid **`KeyErrors`** when getting a possibly non-existant key value
- Can also use index as attribute to get values
  - e.g. `s['a'] == s.a`

`Series` – Operations

- Work with **`Series`** like **`ndarrays`**

- Data is automatically aligned with their labels

```
s1 = pd.Series({'a': 1, 'b': 2})
s2 = pd.Series({'b': 2, 'c': 3})
s1 + s2
```

```
a    NaN
b    4.0
c    NaN
```

`Series` – Methods

- Can use many **numpy** functions on `Series` data
- Access string methods through `.str`
  - e.g. `s1.str.lower()`
- Access datetime methods through `.dt`
  - e.g. `s1.dt.year`
- Full API reference

DataFrame

- 2d array with labels
- Columns can be different types

Pandas - `DataFrame`

- 2-dimensional labeled data structure with columns of potentially different types

- Similar to:
  - a spreadsheet
  - SQL table, or
  - a **`dict`** of **`Series`** objects

- Often faster and more powerful than tables and spreadsheets

DataFrame example

```
import numpy as np
import pandas as pd

dates = pd.date_range("20130101", periods=6)
df = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=list("ABCD"))

print(df)
```

```
                   A         B         C         D
2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
2013-01-04  0.721555 -0.706771 -1.039575  0.271860
2013-01-05 -0.424972  0.567020  0.276232 -1.087401
2013-01-06 -0.673690  0.113648 -1.478427  0.524988
```

Constructing a `DataFrame`

- Input to a **`DataFrame`** can be:
  - Dict of 1D **ndarrays**, **lists**, **dicts**, or **Series**
  - 2-D **ndarray**
  - Structured or record **ndarray**
  - A **Series**
  - Another **DataFrame**
  - CSV, Excel, SQL or JSON file

- As well as optional **index** (row labels) or **columns** (column labels) arguments

DataFrame – From CSV

From CSV/table:

Name	Percent used
JavaScript	65.36
HTML/CSS	55.08
SQL	49.43
Python	48.07

```
df = pd.read_csv('languages.csv')
```

https://survey.stackoverflow.co/2022/#most-popular-technologies-language

`DataFrame` – From CSV

Resulting **`DataFrame`**:

	Name	Percent used
0	JavaScript	65.36
1	HTML/CSS	55.08
2	SQL	49.43
3	Python	48.07

DataFrame – Getting columns

`df.columns` →

	Name	Percent used
0	JavaScript	65.36
1	HTML/CSS	55.08
2	SQL	49.43
3	Python	48.07

`df.index`

`df['Name']`
or
`df.Name`

`df['Percent used']`

DataFrame – Basic operations

```
df.index = df.index + 101
df['Percent not used'] = 100 - df['Percent used']
```

	Name	Percent used	Percent not used
100	JavaScript	65.36	34.64
101	HTML/CSS	55.08	44.92
102	SQL	49.43	50.57
103	Python	48.07	51.93

`DataFrame` – Slicing and indexing

- Can be done by:
  - label `.loc[]`
  - integer position `.iloc[]`

- For each dimension, you can input:
  - A single integer/label (`0` or `100`)
  - A list of integers/labels (`[1, 3]` or `[101, 103]`)
  - A slice object with integers/labels (`1:3` or `101:103`)
    - **Note**: for labels, the stop value is inclusive
  - A Boolean array
  - A callable

DataFrame – Getting rows and data

`df.loc[100]` →

`df.iloc[2]` →

	Name	Percent used	Percent not used
100	JavaScript	65.36	34.64
101	HTML/CSS	55.08	44.92
102	SQL	49.43	50.57
103	Python	48.07	51.93

`df.loc[102, 'Name']`

`df.iloc[3, 2]`

DataFrame – Slicing by position

```
df.iloc[:2]
```

	Name	Percent used	Percent not used
101	JavaScript	65.36	34.64
102	HTML/CSS	55.08	44.92

```
df.iloc[:3, :2]
```

	Name	Percent used
101	JavaScript	65.36
102	HTML/CSS	55.08
103	SQL	49.43

DataFrame – Boolean indexing

```
df['Percent used']> 50
```

- Returns the `Series` `{1: True, 2: True, 3: False, 4: False}`

Can use to filter the `DataFrame`

```
df[df['Percent used']> 50]
```

	Name	Percent used	Percent not used
100	JavaScript	65.36	34.64
101	HTML/CSS	55.08	44.92

`DataFrame` – Methods (just a taste)

- `.fillna()`
- `.dropna()`
- `.sort_values()`
- `.sort_index()`
- `.T` – transpose (swap index and columns)
- `.rename()` – rename index or column labels
- `.groupby()` – e.g. 'Percent used' grouped by respondent type (*professional* vs *learner*)

Most return a new `DataFrame`, unless you pass `inplace=True`

- Full API reference

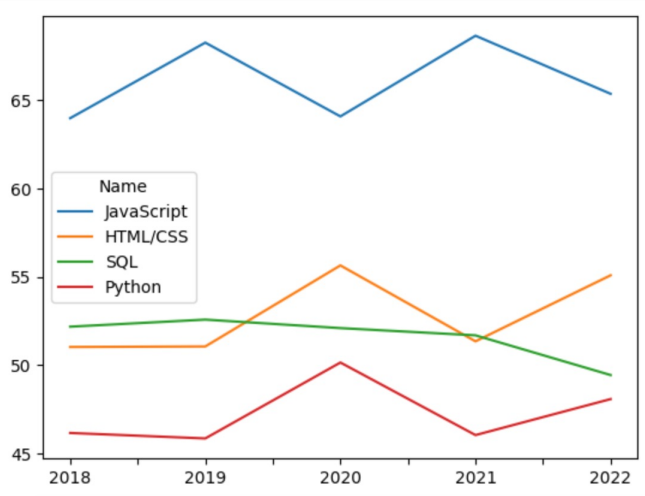
`DataFrame` – Iterating

- It's much faster to apply changes based on the entire DataFrame, rows or columns

- If you need to iterate, you can use:
  - `.items()` – columns as `Series`
  - `.iterrows()` – rows as `Series`
  - `.itertuples()` – rows as `namedtuple`

Plotting

- Supports plotting with `matplotlib`:
  - Line
  - Bar
  - Histogram
  - Box
  - Area
  - Scatter
  - Pie
  - and more



More types of data

- `DatetimeIndex`
  - Like a range series, but with datetimes
  - Create using date_range()

- `Categorical`
  - Represents a limited, fixed number of possible values
  - Categories are ordered
  - Use `dtype="category"` when creating a `Series` **OR**
  - Create one with `pd.Categorical()`

Resources – Pandas

- [Pandas] **10 minutes to pandas**
  - https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html
- [Pandas] **Intro to data structures**
  - https://pandas.pydata.org/pandas-docs/stable/user_guide/dsintro.html
- [Real Python] **The Pandas DataFrame**
  - https://realpython.com/pandas-dataframe
- [Real Python] **Pandas: How to Read and Write Files**
  - https://realpython.com/pandas-read-write-files/

Need more than 2 dimensions?

- Try **Xarray**!
- Provides N-D labeled arrays and datasets in Python

- https://xarray.dev/

Xarray

- Adds labeled indices/dimensions on top of NumPy n-d arrays
- Add coordinates and attributes

- More intuitive, concise and less error prone than using NumPy

- E.g. **`arr.sum(index=1)`** becomes **`arr.sum('time')`**

Xarray data structures

- **DataArray** is like a multi-dimensional **pandas.Series**
- **Dataset** is a dict-like container of **DataArray** objects, with a similar purpose to **pandas.DataFrame** objects
- Overview: Why xarray?

Project – pt. 4

- Update **`survey_analysis.ipynb`**
- Use **`pandas`**

Course wrap-up

For algorithm and interview practice

- Learn about big-O notation
- Create your own custom Linked List implementation
- Find existing Python data structures for graphs, trees, and heaps or create them from scratch
- Do some practice problems or go through a course:
  - Free Code Camp course
  - Hacker Rank

Big-O notation

- **Big-O cheatsheet**
  - https://www.bigocheatsheet.com/

- **Time complexity of Python data structures**
  - https://betterprogramming.pub/a-comprehensive-guide-to-pythons-built-in-data-structures-4d7ca2d242e5

Recommended follow-up

Videos

- Data Structures, Algorithms, and Machine Learning Optimization – link

Live training

- Python Comprehensions and Generator Expressions - link

Course

- Designing Data Structures in Python – link

Books

- Data Structures and Algorithms in Python – link

Resources – Built-in data structures

- [O'Reilly interactive lab] **Data Structures**
  - https://learning.oreilly.com/scenarios/hands-on-python-foundations/9780137904648X004/

- [Real Python] **Lists and Tuples in Python**
  - https://realpython.com/python-lists-tuples/

- [Real Python] **Dictionaries in Python**
  - https://realpython.com/python-dicts/

- [Real Python] **Sets in Python**
  - https://realpython.com/python-sets/

Resources – Custom classes

- [Real Python] **Custom Python Lists: Inheriting From list vs UserList**
  - https://realpython.com/inherit-python-list/

Resources – Comprehensions

- [Real Python] **List Comprehensions**
  - https://realpython.com/list-comprehension-python/
- [Medium] **Examples to master list comprehensions**
  - https://towardsdatascience.com/11-examples-to-master-python-list-comprehensions-33c681b56212
- [Real Python] **Using Generators**
  - https://realpython.com/introduction-to-python-generators

Resources – Module data structures

- [Real Python] **Common Python Data Structures**
  - https://realpython.com/python-data-structures/

Resources – Numpy

- [Numpy] **NumPy quickstart**
  - https://numpy.org/doc/stable/user/quickstart.html
- [Numpy] **The absolute basics for beginners**
  - https://numpy.org/doc/stable/user/absolute_beginners.html
- [Real Python] **NumPy Tutorial**
  - https://realpython.com/numpy-tutorial/

Resources – Pandas

- [Pandas] **10 minutes to pandas**
  - https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html
- [Pandas] **Intro to data structures**
  - https://pandas.pydata.org/pandas-docs/stable/user_guide/dsintro.html
- [Real Python] **The Pandas DataFrame**
  - https://realpython.com/pandas-dataframe
- [Real Python] **Pandas: How to Read and Write Files**
  - https://realpython.com/pandas-read-write-files/

Beginner Live Trainings by Arianne

- **Introduction to Python Programming**
  - Variables, functions, conditionals, lists, loops
  - Skill level – 1/10

- **Programming with Python: Beyond the Basics**
  - Dictionaries, exceptions, files, HTTP requests, web scraping
  - Skill level – 2/10

- **Python Environments and Best Practices**
  - Virtual envs, testing, debugging, PyCharm tips, git, modules
  - Skill level – 3/10

- **Hands-on Python Foundations in 3 Weeks**
  - Multi-week course that covers most of the above material
  - Skill level 1-3

Intermediate Live Trainings by Arianne

- **Object-Oriented Programming in Python**
  - Classes, dunder methods, and decorators
  - Skill level – 3/10

- **Python Data Structures and Comprehensions**
  - Overview of data structures from the standard library, Numpy and Pandas
  - Skill level – 4/10

- **Python Under the Hood**
  - CPython overview, dunder variables and methods, inspecting objects, debugging
  - Skill level – 5/10

- **Learn GraphQL in 4 Hours**
  - Explore GraphQL features
  - Build a GraphQL API in Django and Node.js
  - Skill level – 5/10

Video Trainings by Arianne

- **Introduction to Python LiveLessons - Link**
  - Very beginner content w/ brief intro to data analysis and web development
- **Next Level Python LiveLessons - Link**
  - Material from this class
  - Setting up Python projects with virtual environments and git
  - Testing, debugging, and understanding modules
- **Introduction to Django - Link**
  - Understand basics of creating web applications in Django
  - Start a new project and app
  - Overview of different components and features
- **Rethinking REST: A hands-on guide to GraphQL and Queryable APIs - Link**
  - Explore GraphQL features
  - Build a GraphQL client in JavaScript and a server in Django or Node.js

Interactive labs by Arianne

Hands-On Python Foundations course

1. Getting Started with Python
2. Types, Variables and Strings
3. Functions and Control Flow
4. Data Structures
5. Exceptions and File Handling
6. Requests and APIs
7. Virtual Environments and Pip
8. Intro to Classes
9. Modules and Packages

Thanks!

- Questions?

- Email me at
- arianne.dee.studios@gmail.com