Department of Artificial Intelligence and Data Science

Fundamentals of Data Science (AD45)

Academic Year : 2024-24, Batch 2023

Credits: 3:0:0

Text Books:

Joel Grus, "Data Science from Scratch", 2nd Edition, O'Reilly Publications/Shroff Publishers and Distributors Pvt. Ltd., 2019. ISBN-13: 978- 9352138326

UNIT 1

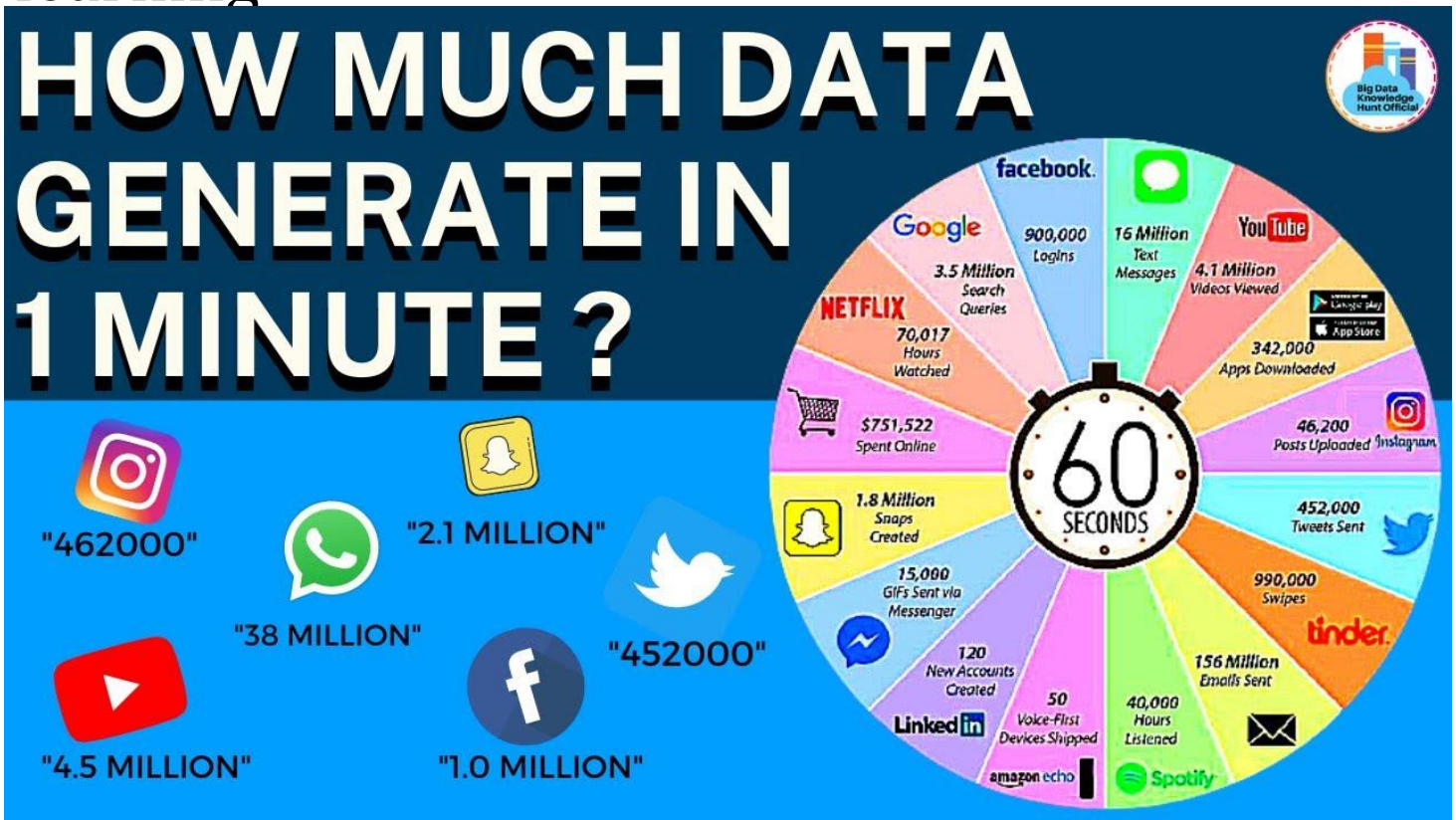
What is Data Science? Types of Data & Data Sources, Visualizing Data, matplotlib, Bar Charts, Line Charts, Scatterplots, Linear Algebra, Vectors, Matrices, Statistics, Describing a Single Set of Data, Correlation, Simpson's Paradox, Some Other Correlational Caveats, Correlation and Causation. Probability, Dependence and Independence, Random Variables, Continuous Distributions, The Normal Distribution.

The Ascendance of Data

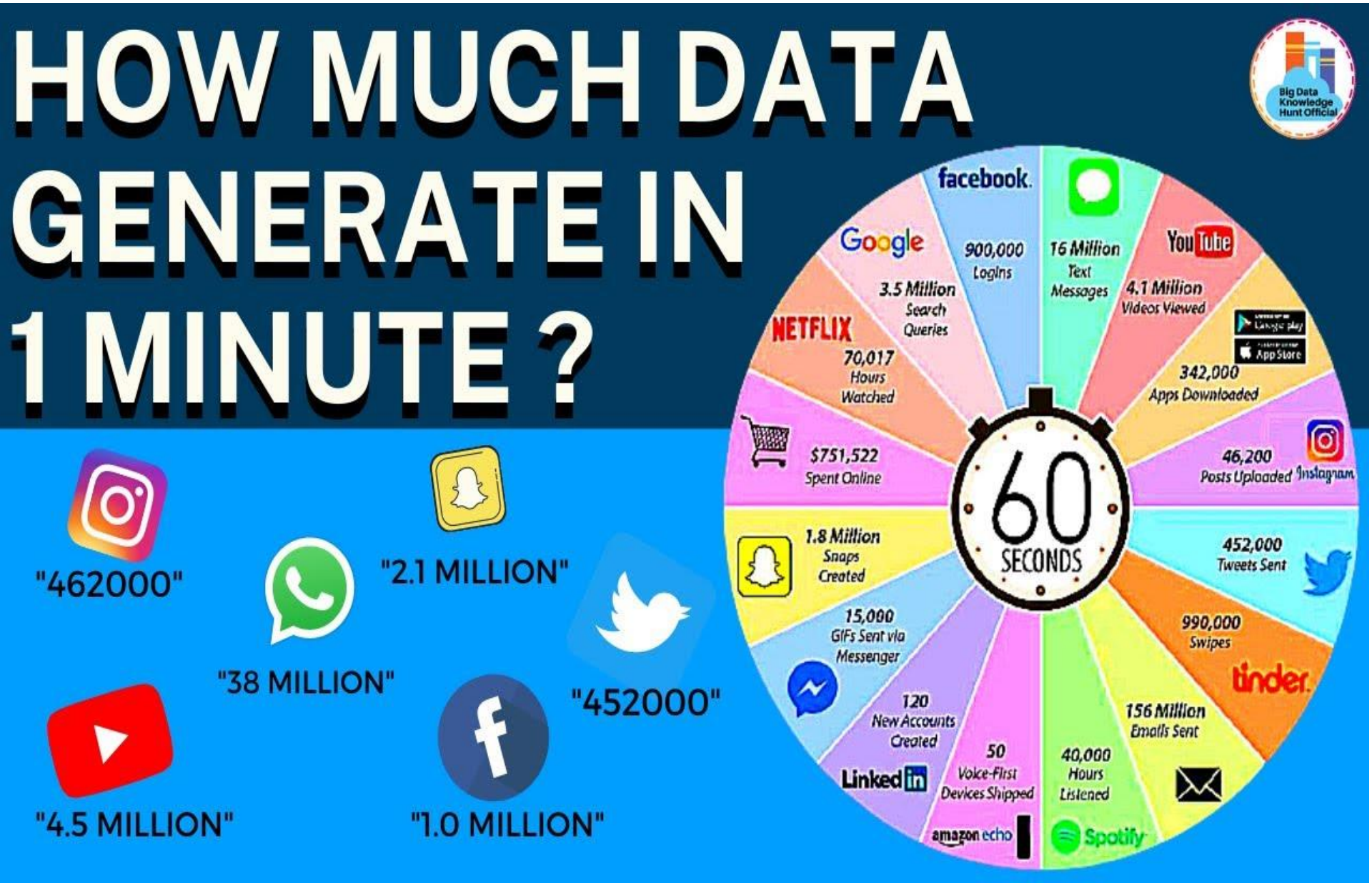
- We live in a world that's drowning in data. Websites track every user's every click.
- Your **smartphone** is building up a record of your location and speed every second of every day. "Quantified selfers" wear pedometers-on-steroids that are always recording their heart rates, movement habits, diet, and sleep patterns.
- **Smart cars** collect driving habits, smart homes collect living habits, and smart marketers collect purchasing habits.
- The **internet** itself represents a huge graph of knowledge that contains (among other things) an enormous cross-referenced encyclopedia; domain-specific databases about movies, music, sports results, pinball machines, memes, and cocktails; and too many government statistics (some of them nearly true!) from too many governments to wrap your head around.

What Is Data Science?

- **What is Data?**
- **Data** is a collection of facts, numbers, measurements, observations. It can be raw or processed and is used to gain insights, make decisions, and drive technologies like AI and machine learning.

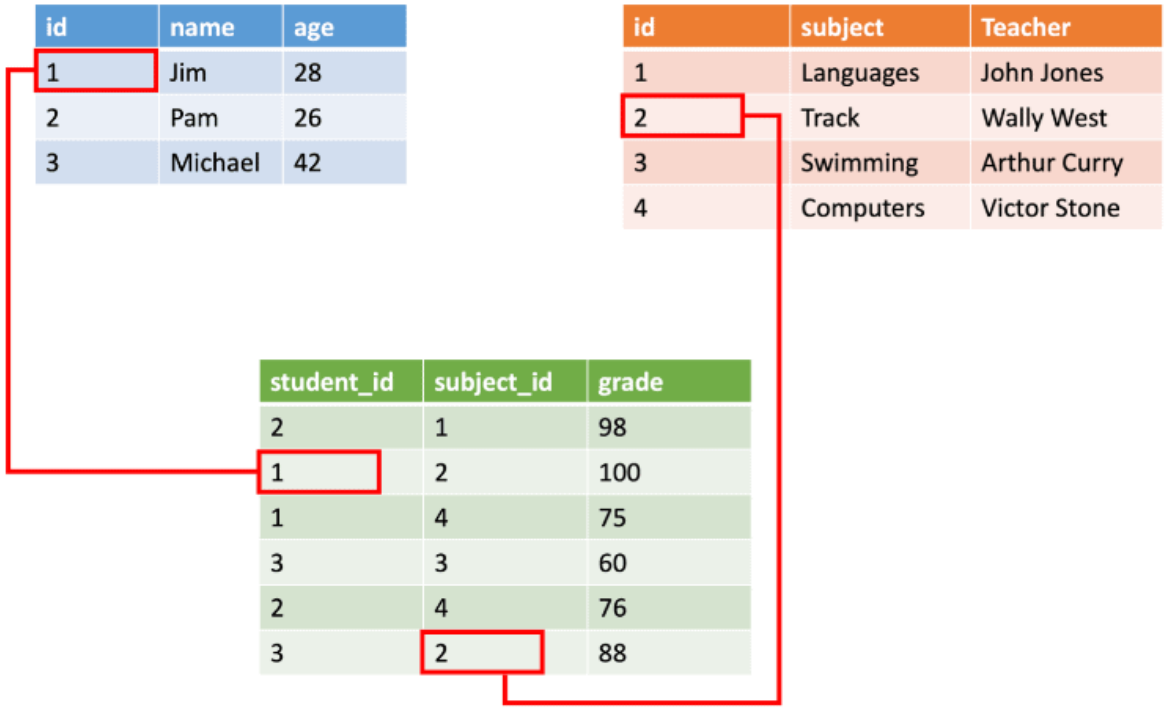


What Is Data Science?



What Is Data Science?: Types of Data

- **Structured Data**
  - Organized and stored in a fixed format (e.g., databases, spreadsheets).
  - Examples:
    - Customer records (Name, Age, Email)
    - Sales data (Price, Quantity, Date)
    - SQL Databases



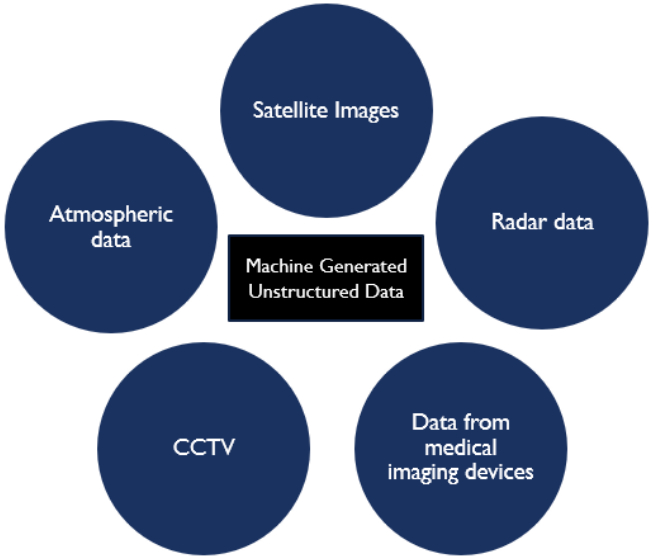
id	name	age
1	Jim	28
2	Pam	26
3	Michael	42

id	subject	Teacher
1	Languages	John Jones
2	Track	Wally West
3	Swimming	Arthur Curry
4	Computers	Victor Stone

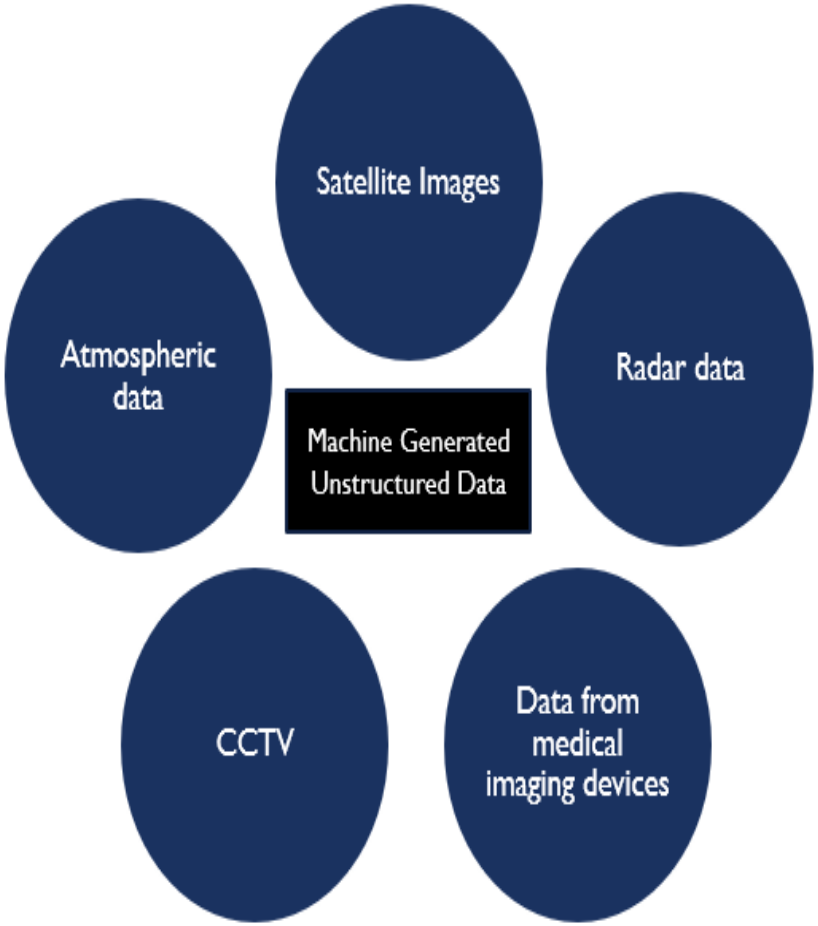
student_id	subject_id	grade
2	1	98
1	2	100
1	4	75
3	3	60
2	4	76
3	2	88

What Is Data Science?: Types of Data

- **Unstructured Data**
- No predefined format, making it harder to process.
- Examples:
  - Text documents, emails, social media posts
  - Images, videos, audio files
  - Sensor data, logs



What Is Data Science?: Types of Data



What Is Data Science?: Types of Data

- **Semi-Structured Data**
- Has some organization but doesn't fit traditional databases.
- Examples:
  - JSON, XML files
  - Emails with metadata (subject, timestamp)

Semi-structured data

```
<University>
 <Student ID="1">
  <Name>John</Name>
  <Age>18</Age>
  <Degree>B.Sc.</Degree>
 </Student>
 <Student ID="2">
  <Name>David</Name>
  <Age>31</Age>
  <Degree>Ph.D. </Degree>
 </Student>
....
</University>
```

```
{
  "name": "Jane Smith",
  "email": "jane.smith@example.com",
  "preferences": {
    "newsletter": true,
    "smsNotifications": false
  }
}
```

What Is Data Science?: Sources of Data

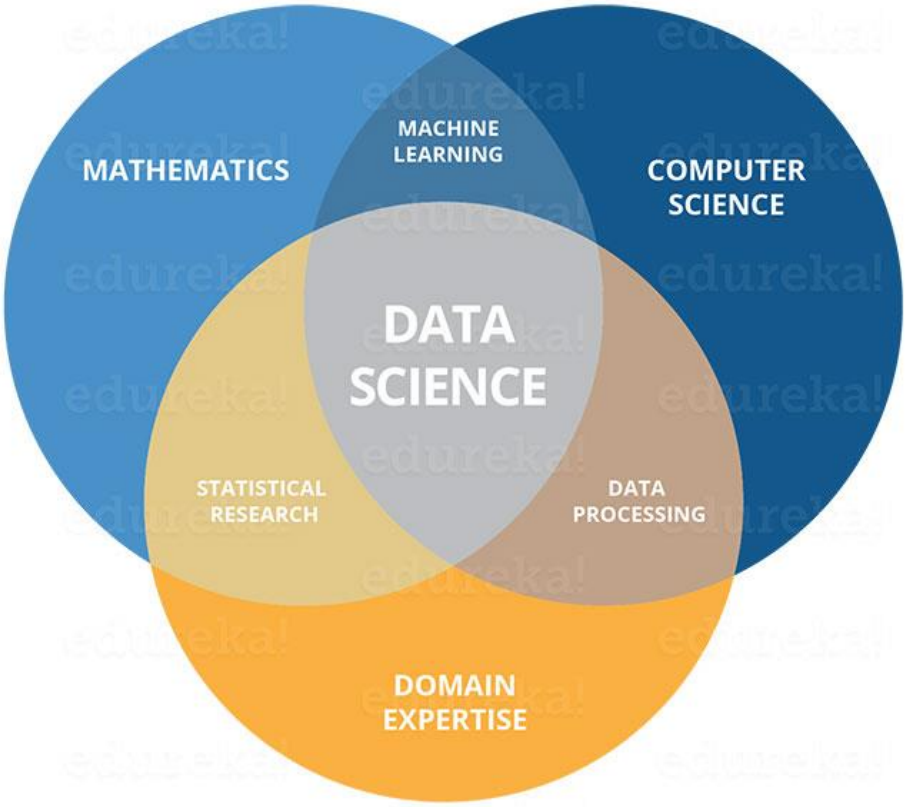
- **Digital Transactions** – Sales records, banking transactions.
- **Social Media** – Posts, likes, shares, and comments.
- **Sensors & IoT Devices** – Temperature sensors, smartwatches.
- **Healthcare Records** – Patient data, medical images.
- **Government & Research** – Census data, scientific experiments.

Why is Data Important?

- **Drives decision-making** – Businesses use data to improve operations.
- **Enables AI & Machine Learning** – Models learn from large datasets.
- **Improves efficiency** – Automates tasks and predicts trends.
- **Personalizes experiences** – Recommender systems (Netflix, Amazon).

What Is Data Science?

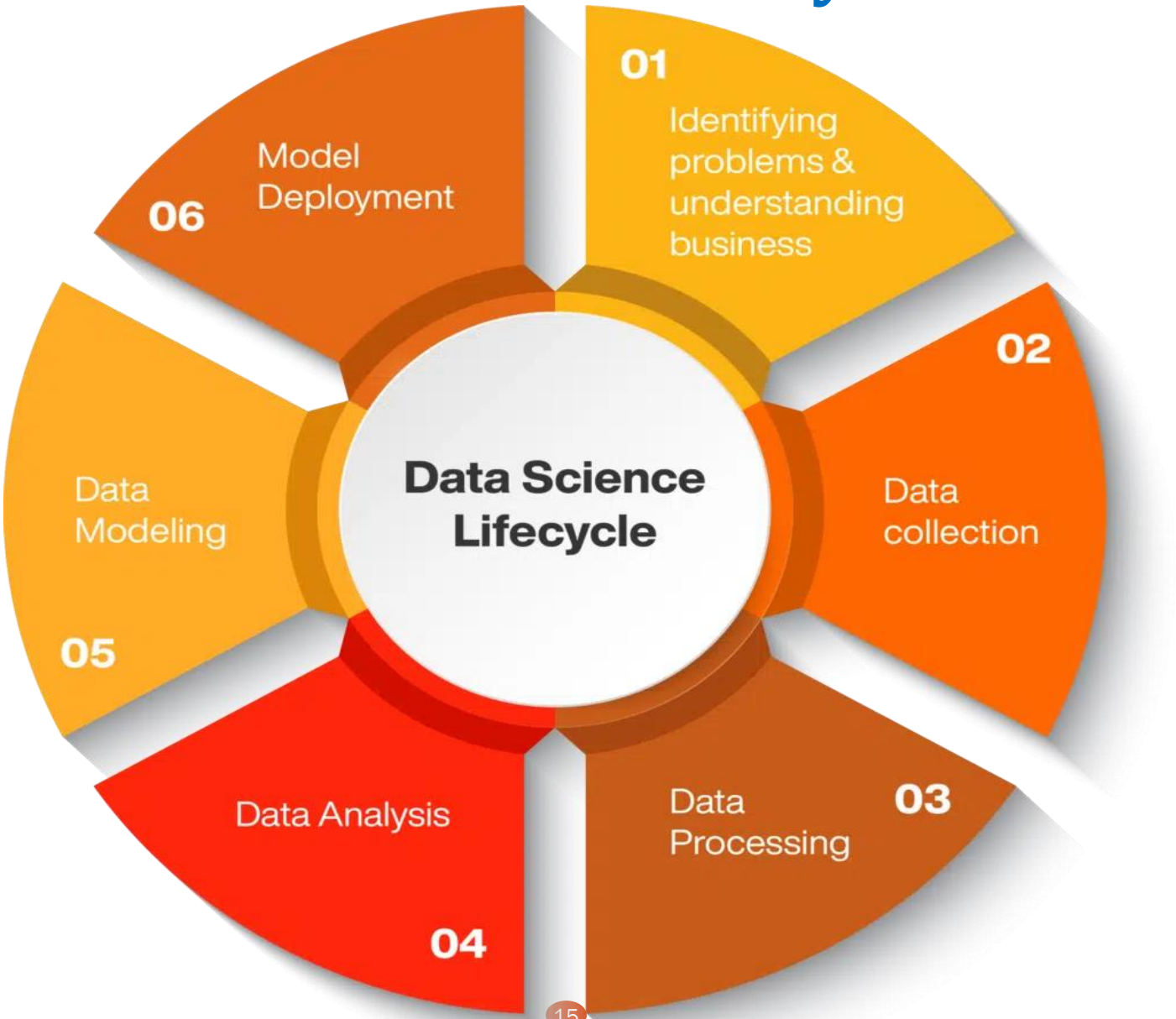
- Data science is the **study** of data to **derive** meaningful insights for businesses. It's a multidisciplinary field that combines **statistics, math, computer engineering, and artificial intelligence** (AI).



What Is Data Science Lifecycle?

- This lifecycle encompasses **Six key stages**, that include—Identify problem, data collection, data pre-processing, data analysis, data modeling, and model deployment and dissemination.
- All of these are crucial in the process of converting raw data into valuable knowledge to support projects and their execution.

What Is Data Science Lifecycle?



What are lambda functions in Python?

- In Python, an anonymous function is a function that is defined without a name.
- While normal functions are defined using the ***def*** keyword in Python, anonymous functions are defined using the ***lambda*** keyword.
- Hence, anonymous functions are also called ***lambda*** functions
- ***How to use lambda Functions in Python?***
- A lambda function in python has the following syntax.

lambda arguments: expression

- Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

Lists

- The most fundamental data structure in Python is the list. A list is simply an ordered collection. It is called as array in other programming languages.

✓ `integer_list = [1, 2, 3]`

✓ `heterogeneous_list = ["string", 0.1, True]`

✓ `list_of_lists = [ integer_list, heterogeneous_list, [] ]`

✓ `list_length = len(integer_list) # equals 3`

✓ `list_sum = sum(integer_list) # equals 6`

Lists

- If you don't want to modify x, you can use list addition:

```
x = [1, 2, 3]
y = x + [4, 5, 6]        # y is [1, 2, 3, 4, 5, 6]; x is unchanged
```

- More frequently we will append to lists one item at a time:

```
x = [1, 2, 3]
x.append(0)        # x is now [1, 2, 3, 0]
y = x[-1]          # equals 0
z = len(x)         # equals 4
```

- It's often convenient to unpack lists when you know how many elements they contain:

```
x, y = [1, 2]     # now x is 1, y is 2
```

although you will get a *ValueError* if you don't have the same number of elements on both sides.

- A common idiom is to use an underscore for a value you're going to throw away:

```
_, y = [1, 2]     # now y == 2, didn't care about the first element
```

Tuples

- Tuples are lists' immutable cousins.
- Pretty much anything you can do to a list that doesn't involve modifying it, you can do to a tuple.
- You specify a tuple by using parentheses (or nothing) instead of square brackets:

```
my_list = [1, 2]
my_tuple = (1, 2)
other_tuple = 3, 4
my_list[1] = 3      # my_list is now [1, 3]
```

```
try:
    my_tuple[1] = 3
except TypeError:
    print("cannot modify a tuple")
```

- Tuples are a convenient way to return multiple values from functions:

```
def sum_and_product(x, y):
    return (x + y), (x * y)

sp = sum_and_product(2, 3)      # sp is (5, 6)
s, p = sum_and_product(5, 10)   # s is 15, p is 50
```

- Tuples (and lists) can also be used for multiple assignment:

```
x, y = 1, 2     # now x is 1, y is 2
x, y = y, x     # Pythonic way to swap variables; now x is 2, y is 1
```

Dictionaries

- Another fundamental data structure is a dictionary, which associates values with keys and allows you to quickly retrieve the value corresponding to a given key:

```
empty_dict = {}                          # Pythonic
empty_dict2 = dict()                     # less Pythonic
grades = {"Joel": 80, "Tim": 95}         # dictionary literal
```

- You can look up the value for a key using square brackets:

```
joels_grade = grades["Joel"]             # equals 80
```

- But you'll get a KeyError if you ask for a key that's not in the dictionary:

```
try:
    kates_grade = grades["Kate"]
except KeyError:
    print("no grade for Kate!")
```

Dictionaries

- You can check for the existence of a key using ***in.*** This membership check is fast even for large dictionaries.

```
joel_has_grade = "Joel" in grades      # True
kate_has_grade = "Kate" in grades      # False
```

- Dictionaries have a get method that returns a default value (instead of raising an exception) when you look up a key that's not in the dictionary:

```
joels_grade = grades.get("Joel", 0)    # equals 80
kates_grade = grades.get("Kate", 0)    # equals 0
no_ones_grade = grades.get("No One")   # default is None
```

- You can assign key/value pairs using the same square brackets:

```
grades["Tim"] = 99                     # replaces the old value
grades["Kate"] = 100                   # adds a third entry
num_students = len(grades)             # equals 3
```

Dictionaries

- you can use dictionaries to represent structured data:

```
tweet = {
    "user" : "joelgrus",
    "text" : "Data Science is Awesome",
    "retweet_count" : 100,
    "hashtags" : ["#data", "#science", "#datascience", "#awesome",
"#yolo"]
}
```

- Besides looking for specific keys, we can look at all of them:

```
tweet_keys   = tweet.keys()     # iterable for the keys
tweet_values = tweet.values()   # iterable for the values
tweet_items  = tweet.items()    # iterable for the (key, value) tuples

"user" in tweet_keys            # True, but not Pythonic

"user" in tweet                  # Pythonic way of checking for keys
"joelgrus" in tweet_values       # True (slow but the only way to check)
```

- Dictionary keys must be “hashable”; in particular, you cannot use lists as keys.
- If you need a multipart key, you should probably use a tuple or figure out a way to turn the key into a string.

defaultdict

- Imagine that you're trying to count the words in a document.
- An obvious approach is to create a dictionary in which the keys are words and the values are counts.
- As you check each word, you can increment its count if it's already in the dictionary and add it to the dictionary if it's not:

```
word_counts = {}
for word in document:
    if word in word_counts:
        word_counts[word] += 1
    else:
        word_counts[word] = 1
```

- You could also use the "forgiveness is better than permission" approach and just handle the exception from trying to look up a missing key:

```
word_counts = {}
for word in document:
    try:
        word_counts[word] += 1
    except KeyError:
        word_counts[word] = 1
```

- A third approach is to use get, which behaves gracefully for missing keys:

```
word_counts = {}
for word in document:
    previous_count = word_counts.get(word, 0)
    word_counts[word] = previous_count + 1
```

defaultdict

- Every one of these is slightly unwieldy, which is why defaultdict is useful.
- A ***defaultdict*** is like a regular dictionary, except that when you try to look up a key it doesn't contain, it first adds a value for it using a zero-argument function you provided when you created it.
- In order to use defaultdicts, you have to import them from collections:

```
from collections import defaultdict

word_counts = defaultdict(int)          # int() produces 0
for word in document:
    word_counts[word] += 1
```

- They can also be useful with list or dict, or even your own functions:

```
dd_list = defaultdict(list)             # list() produces an empty list
dd_list[2].append(1)                    # now dd_list contains {2: [1]}

dd_dict = defaultdict(dict)             # dict() produces an empty dict
dd_dict["Joel"]["City"] = "Seattle"     # {"Joel" : {"City": Seattle"}}

dd_pair = defaultdict(lambda: [0, 0])
dd_pair[2][1] = 1                       # now dd_pair contains {2: [0,
1]}
```

- These will be useful when we're using dictionaries to "collect" results by some key and don't want to have to check every time to see if the key exists yet.

Counters

- A Counter turns a sequence of values into a defaultdict(int)-like object mapping keys to counts:

```
from collections import Counter
c = Counter([0, 1, 2, 0])          # c is (basically) {0: 2, 1: 1, 2: 1}
```

- This gives us a very simple way to solve our word_counts problem:

```
# recall, document is a list of words
word_counts = Counter(document)
```

- A Counter instance has a ***most_common*** method that is frequently useful:

```
# print the 10 most common words and their counts
for word, count in word_counts.most_common(10):
    print(word, count)
```

Sets

- Another useful data structure is set, which represents a collection of distinct elements.
- You can define a set by listing its elements between curly braces:

```
primes_below_10 = {2, 3, 5, 7}
```

- However, that doesn't work for empty sets, as {} already means "empty dict." In that case you'll need to use set() itself:

```
s = set()
s.add(1)       # s is now {1}
s.add(2)       # s is now {1, 2}
s.add(2)       # s is still {1, 2}
x = len(s)     # equals 2
y = 2 in s     # equals True
z = 3 in s     # equals False
```

Sets

- We'll use sets for two main reasons. The first is that in is a very fast operation on sets.
- If we have a large collection of items that we want to use for a membership test, a set is more appropriate than a list:

```
stopwords_list = ["a", "an", "at"] + hundreds_of_other_words + ["yet",
"you"]

"zip" in stopwords_list     # False, but have to check every element

stopwords_set = set(stopwords_list)
"zip" in stopwords_set      # very fast to check
```

- The second reason is to find the distinct items in a collection:

```
item_list = [1, 2, 3, 1, 2, 3]
num_items = len(item_list)                  # 6
item_set = set(item_list)                   # {1, 2, 3}
num_distinct_items = len(item_set)          # 3
distinct_item_list = list(item_set)         # [1, 2, 3]
```

Control Flow

- As in most programming languages, you can perform an action conditionally using if:

```
if 1 > 2:
    message = "if only 1 were greater than two..."
elif 1 > 3:
    message = "elif stands for 'else if'"
else:
    message = "when all else fails use else (if you want to)"
```

- You can also write a ternary if-then-else on one line, which we will do occasionally:

```
parity = "even" if x % 2 == 0 else "odd"
```

- Python has a while loop:

```
x = 0
while x < 10:
    print(f"{x} is less than 10")
    x += 1
```

```
1  x = 3
2  if x % 2 == 0:
3      parity = "even"
4  else:
5      parity = "odd"
6  print(parity)
7
```

✓ 0.4s

odd

```
1  x = 3
2  parity = "even" if x % 2 == 0 else "odd"
3  print(parity)
4
```

✓ 0.4s

odd

Control Flow

- for and in:

```
# range(10) is the numbers 0, 1, ..., 9
for x in range(10):
    print(f"{x} is less than 10")
```

- If you need more complex logic, you can use continue and break:

```
for x in range(10):
    if x == 3:
        continue  # go immediately to the next iteration
    if x == 5:
        break     # quit the loop entirely
    print(x)
```

- This will print 0, 1, 2, and 4.

Truthiness

- Booleans in Python work as in most other languages, except that they're capitalized:

```
one_is_less_than_two = 1 < 2          # equals True
true_equals_false = True == False     # equals False
```

- Python uses the value None to indicate a nonexistent value. It is similar to other languages' null:

```
x = None
assert x == None, "this is the not the Pythonic way to check for None"
assert x is None, "this is the Pythonic way to check for None"
```

- Python lets you use any value where it expects a Boolean. The following are all "falsy":
  - False
  - None
  - [] (an empty list)
  - {} (an empty dict)
  - ""
  - set()
  - 0
  - 0.0

Truthiness

- Pretty much anything else gets treated as True. This allows you to easily use if statements to test for empty lists, empty strings, empty dictionaries, and so on.
- It also sometimes causes tricky bugs if you're not expecting this behavior:

```
s = some_function_that_returns_a_string()
if s:
    first_char = s[0]
else:
    first_char = ""
```

```
1  s = "abc"
2  first_char = s and s[0]
3  print(first_char)
```
✓ 0.4s

```
a
```

- A shorter (but possibly more confusing) way of doing the same is:

```
first_char = s and s[0]
```

- since ***and*** returns its second value when the first is "truthy," and the first value when it's not. Similarly, if x is either a number or possibly None:

```
safe_x = x or 0
```

- is definitely a number, although:

```
safe_x = x if x is not None else 0
```

is possibly more readable.

```
1  x = 5
2  safe_x = x or 0
3  print(safe_x)
```
✓ 0.4s

```
5
```

```
1  x = None
2  safe_x = x or 0
3  print(safe_x)
```
✓ 0.4s

```
0
```

```
1  x = None
2  safe_x = x if x is not None else 0
3  print(safe_x)
4
```
✓ 0.5s

```
0
```

Truthiness

- Python has an all function, which takes an iterable and returns True precisely when every element is truthy, and an any function, which returns True when at least one element is truthy:

```
all([True, 1, {3}])   # True, all are truthy
all([True, 1, {}])    # False, {} is falsy
any([True, 1, {}])    # True, True is truthy
all([])               # True, no falsy elements in the list
any([])               # False, no truthy elements in the list
```

Sorting

- Every Python list has a sort method that sorts it in place.
- You can use the sorted function. which returns a new list:

```
x = [4, 1, 2, 3]
y = sorted(x)      # y is [1, 2, 3, 4], x is unchanged
x.sort()           # now x is [1, 2, 3, 4]
```

- By default, sort (and sorted) sort a list from smallest to largest based on naively comparing the elements to one another.
- If you want elements sorted from largest to smallest, you can specify a ***reverse=True*** parameter. And instead of comparing the elements themselves, you can compare the results of a function that you specify with key:

```
# sort the list by absolute value from largest to smallest
x = sorted([-4, 1, -2, 3], key=abs, reverse=True)  # is [-4, 3, -2, 1]

# sort the words and counts from highest count to lowest
wc = sorted(word_counts.items(),
            key=lambda word_and_count: word_and_count[1],
            reverse=True)
```

List Comprehensions

- List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

The Syntax

newlist = [expression for item in iterable if condition == True]

- The return value is a new list, leaving the old list unchanged.

Condition

- The condition is like a filter that only accepts the items that valuate to True.

Iterable

- The iterable can be any iterable object, like a list, tuple, set etc.

Expression

- The expression is the current item in the iteration, but it is also the outcome, which you can manipulate before it ends up like a list item in the new list.

List Comprehensions

```python
# Based on a list of fruits, you want a new list, containing only the fruits with the letter "a" in the name.

# Without list comprehension you will have to write a for statement with a conditional test inside:

fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

for x in fruits:
    if "a" in x:
        newlist.append(x)

print(newlist)
```

```
['apple', 'banana', 'mango']
```

```python
# With list comprehension you can do all that with only one line of code:

fruits = ["apple", "banana", "cherry", "kiwi", "mango"]

newlist = [x for x in fruits if "a" in x]

print(newlist)
```

```
['apple', 'banana', 'mango']
```

List Comprehensions

- Frequently, you'll want to transform a list into another list by choosing only certain elements, by transforming elements, or both.
- The Pythonic way to do this is with list comprehensions:

```
even_numbers = [x for x in range(5) if x % 2 == 0]  # [0, 2, 4]
squares      = [x * x for x in range(5)]            # [0, 1, 4, 9, 16]

even_squares = [x * x for x in even_numbers]        # [0, 4, 16]
```

- You can similarly turn lists into dictionaries or sets:

```
square_dict = {x: x * x for x in range(5)}  # {0: 0, 1: 1, 2: 4, 3: 9, 4:
16}
square_set  = {x * x for x in [1, -1]}      # {1}
```

- If you don't need the value from the list, it's common to use an underscore as the variable:
-

```
zeros = [0 for _ in even_numbers]       # has the same length as
even_numbers
```

```
1  evenNumbers = [1, 2, 3, 4, 5]
2  zeros = [0 for _ in evenNumbers]
3  print(zeros)
✓ 0.4s
[0, 0, 0, 0, 0]
```

List Comprehensions

- A list comprehension can include multiple ***fors***:

```
pairs = [(x, y)
         for x in range(10)
         for y in range(10)]   # 100 pairs (0,0) (0,1) ... (9,8), (9,9)
```

- and later ***fors*** can use the results of earlier ones:

```
increasing_pairs = [(x, y)                      # only pairs with x < y,
                    for x in range(10)          # range(lo, hi) equals
                    for y in range(x + 1, 10)]  # [lo, lo + 1, ..., hi -
1]
```

Dictionary Comprehension

- Dictionary comprehension is an elegant and concise way to create dictionaries.
- The minimal syntax for dictionary comprehension is:

dictionary = {key: value for vars in iterable}

```
1  # Example 1: Dictionary Comprehension
2  # Consider the following code:
3
4  square_dict = dict()
5  for num in range(1, 11):
6      square_dict[num] = num * num
7  print(square_dict)
8
```

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}
```

```
1  # Now, let's create the dictionary in the above program using dictionary comprehension.
2
3  # dictionary comprehension example
4  square_dict = {num: num * num for num in range(1, 11)}
5  print(square_dict)
6
```

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}
```

Dictionary Comprehension

```
1  #item price in dollars
2  old_price = {'milk': 1.02, 'coffee': 2.5, 'bread': 2.5}
3
4  dollar_to_pound = 0.76
5  new_price = {item: value*dollar_to_pound for (item, value) in old_price.items()}
6  print(new_price)
```

```
{'milk': 0.7752, 'coffee': 1.9, 'bread': 1.9}
```

```
1  original_dict = {'jack': 38, 'michael': 48, 'guido': 57, 'john': 33}
2
3  even_dict = {k: v for (k, v) in original_dict.items() if v % 2 == 0}
4  print(even_dict)
```

```
{'jack': 38, 'michael': 48}
```

Automated Testing and assert

- As data scientists, we'll be writing a lot of code. How can we be confident our code is correct?
- One way is with types (discussed shortly), but another way is with automated tests.
- We will be using assert statements, which will cause your code to raise an AssertionError if your specified condition is not truthy:

```
assert 1 + 1 == 2
assert 1 + 1 == 2, "1 + 1 should equal 2 but didn't"
```

- As you can see in the second case, you can optionally add a message to be printed if the assertion fails.
- It's not particularly interesting to assert that 1 + 1 = 2. What's more interesting is to assert that functions you write are doing what you expect them to:

```
def smallest_item(xs):
    return min(xs)

assert smallest_item([10, 20, 5, 40]) == 5
assert smallest_item([1, 0, -1, 2]) == -1
```

Object Oriented Programming

- Python allows you to define classes that encapsulate data and the functions that operate on them.
- Here we'll construct a class representing a "counting clicker," the sort that is used at the door to track how many people have shown up for the "advanced topics in data science" meetup.
- It maintains a count, can be clicked to increment the count, allows you to read_count, and can be reset back to zero.
- To define a class, you use the class keyword and a PascalCase name.
- By convention, each takes a first parameter, self, that refers to the particular class instance.
- Normally, a class has a constructor, named __init__. It takes whatever parameters you need to construct an instance of your class and does whatever setup you need.
- we construct instances of the clicker using just the class name.

random.seed() and random.randrange()

- The random module actually produces pseudorandom (that is, deterministic) numbers based on an internal state that you can set with ***random.seed*** if you want to get reproducible results.

```
1  random.seed(10) # set the seed to 10
2  print(random.random())
3  random.seed(10) # reset the seed to 10
4  print(random.random())
5
```

✓ 0.5s

```
0.5714025946899135
0.5714025946899135
```

- We'll sometimes use ***random.randrange***, which takes either one or two arguments and returns an element chosen randomly from the corresponding range:

```
random.randrange(10)    # choose randomly from range(10) = [0, 1, ..., 9]
random.randrange(3, 6)  # choose randomly from range(3, 6) = [3, 4, 5]
```

random.shuffle() and random.choice()

- There are a few more methods that we'll sometimes find convenient.
- For example, ***random.shuffle*** randomly reorders the elements of a list:

```
1  up_to_ten = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2  random.shuffle(up_to_ten)
3  print(up_to_ten)
✓ 0.4s
[10, 9, 5, 3, 6, 4, 2, 1, 8, 7]
```

- If you need to randomly pick one element from a list, you can use ***random.choice***

```
1  my_best_friend = random.choice(["Alice", "Bob", "Charlie"])
2  print(my_best_friend)
✓ 0.3s
Alice
```

- And if you need to randomly choose a sample of elements without replacement (i.e., with no duplicates), you can use ***random.sample***:

```
1  lottery_numbers = range(60)
2  winning_numbers = random.sample(lottery_numbers, 6)
3  print(winning_numbers)
✓ 0.4s
[33, 31, 20, 4, 15, 47]
```

- To choose a sample of elements with replacement (i.e., allowing duplicates), you can just make multiple calls to ***random.choice***:

```
1  four_with_replacement = [random.choice(range(10)) for _ in range(4)]
2  print(four_with_replacement)
✓ 0.5s
[5, 0, 6, 2]
```

zip and Argument Unpacking

- Often we will need to zip two or more iterables together.
- The ***zip*** function transforms multiple iterables into a single iterable of tuples of corresponding function:

```
1  list1 = ['a', 'b', 'c']
2  list2 = [1, 2, 3]
3  # zip is lazy, so you have to do something like the following
4  [pair for pair in zip(list1, list2)]
✓ 0.7s
[('a', 1), ('b', 2), ('c', 3)]
```

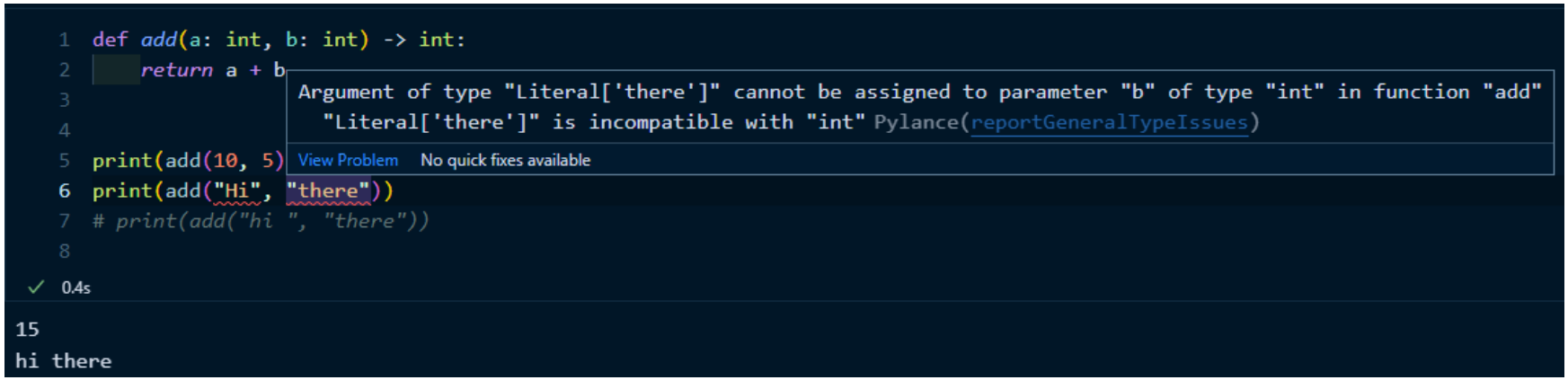
- If the lists are different lengths, zip stops as soon as the first list ends.
- You can also "unzip" a list using a strange trick:

```
pairs = [('a', 1), ('b', 2), ('c', 3)]
letters, numbers = zip(*pairs)
```

- The asterisk (*) performs argument unpacking, which uses the elements of pairs as individual arguments to zip.

Type Annotations

- Python is a dynamically typed language.
- That means that it in general it doesn't care about the types of objects we use, as long as we use them in valid ways.
- Whereas in a statically typed language our functions and objects would have specific types.



- Type annotations allow us to tell Python what we expect to be assigned to names at given points in our application. We can then use these annotations to check the program is in fact doing what we intend.
- However, these type annotations don't actually do anything. You can still use the annotated add function to add strings, and the call to add(10, "five") will still raise the exact same TypeError.

Type Hinting

- In a nutshell: **Type hinting** is literally what the words mean. You hint the type of the object(s) you're using.
- Due to the dynamic nature of Python, inferring or checking the type of an object being used is especially hard.
- This fact makes it hard for developers to understand what exactly is going on in code they haven't written and, most importantly, for type checking tools found in many IDEs that are limited due to the fact that they don't have any indicator of what type the objects are.
- As a result they resort to trying to infer the type with around 50% success rate.

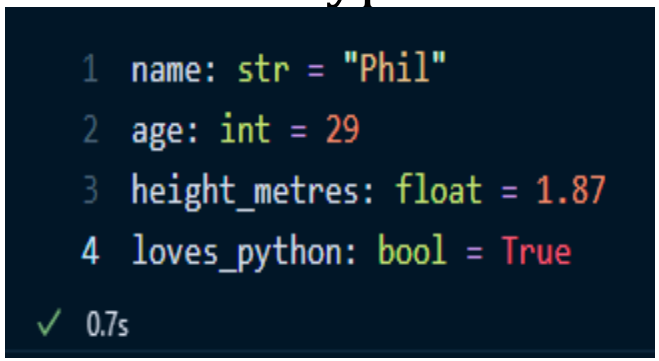
Type Annotations

- There are still (at least) four good reasons to use type annotations in your Python code:

1. Types are an important form of documentation.
2. There are external tools (mypy and pylance) that will read your code, inspect the type annotations, and let you know about type errors before you ever run your code.
3. Having to think about the types in your code forces you to design cleaner functions and interfaces.
4. Using types allows your editor to help you with things like autocomplete and to get angry at type errors.

Type Annotations

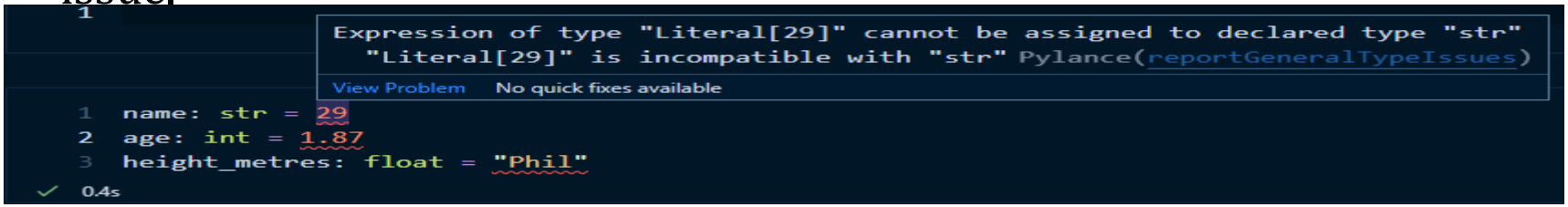
- Let's start by using type hinting to annotate some variables that we expect to take basic types.

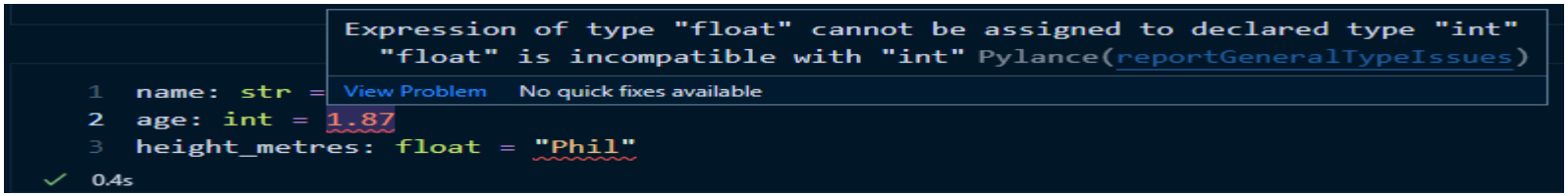


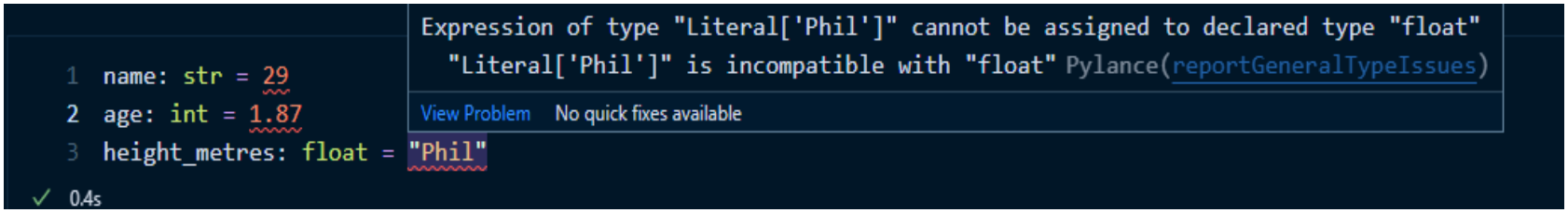
- As you can see from the examples above, we can annotate a variable by adding a colon after the variable name, after which we specify the type.
- Here we've indicated that name is a string, age is an integer, and height_metres should be a float.
- In this case, all of our type annotations align with the values that have been assigned, so we don't have any issues.

Annotating collections

- Now let's look at what happens when things are not what we intended.
- All of our values have gotten jumbled up. Maybe we assigned these values from an iterable that contained the values in the wrong order.
- The errors are actually very helpful. They explain exactly what went wrong, and at the start of each line we get a reference to the file where the error happened, along with a line number.
- Using this information, we can easily track down the source of this issue.

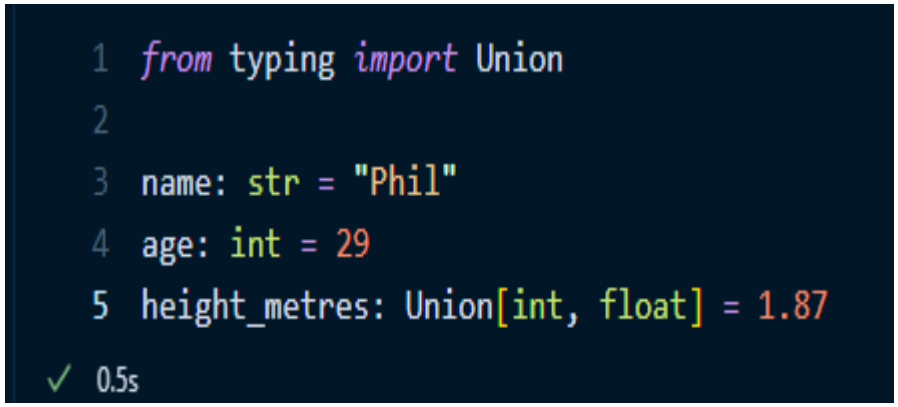






Using *typing* module and *Union*

- Let's say we want to be a little more flexible in how we handle height_metres.
- We only really care that it's a real number, so instead of accepting just floats, I want to also accept integers.
- The way we accomplish this is by using a tool called ***Union*** which we have to import from the ***typing*** module.
- Here we've added ***Union[int, float]*** as a type annotation for ***height_metres***, which means we can accept either integers or floats.
- We can add as many types as we like to this Union by adding more comma separated values between the square brackets.



- The ***typing*** module provides runtime support for type hints.

Using *Any*

- We can also get super flexible and use another tool called ***Any***, which matched any type.
- You should be careful about using ***Any***, because it largely removed the benefits of type hinting.
- It can be useful for indicating to readers that something is entirely generic though.
- We can use ***Any*** like any of the other types:

```
1 from typing import Any
2
3 name: str = "Phil"
4 age: int = 29
5 height_metres: Any = 1.87
✓ 0.4s
```

Annotating collections

- Now that we've looked at annotating basic types, let's talk about how we might annotate that something should be a list, or maybe a tuple containing values of a specific type.
- In order to annotate collections, we have to import special types from the ***typing*** module.
- For lists we need to import ***List*** and for tuples we need to import ***Tuple***.
- As you can see, the names make a lot of sense.

```
1 from typing import List
2
3 names: List = ["Rick", "Morty", "Summer", "Beth", "Jerry"]
✓ 0.5s
```

- Here is an example of a variable using a List annotation:

Annotating collections

- If we wanted to specify which types should in the list, we can add a set of square brackets, much like we did with Union.

```
1  from typing import List
2
3  names: List[str] =  ["Rick", "Morty", "Summer", "Beth", "Jerry"]
✓ 0.6s
```

- If we want, we can allow a variety of types in a list by combining List and Union like this:

```
1  from typing import List, Union
2
3  random_values: List[Union[str, int]] =  ["x", 13, "camel", 0]
✓ 0.5s
```

- When working with tuples, we can specify a type for each item in sequence, since tuples are of fixed length and are immutable.
- For example, we can do something like this:

```
1  from typing import Tuple
2
3  movie: Tuple[str, str, int] = ("Toy Story 3", "Lee Unkrich", 2010)
✓ 0.4s
```

Creating type aliases

- Let's consider a case where we want to store lots of movies. How would we annotate something like that?
- Maybe something like this:

```python
from typing import List, Tuple

movies: List[Tuple[str, str, int]] = [
    ("Finding Nemo", "Andrew Stanton", 2005),
    ("Inside Out", "Pete Docter", 2015),
    ("Toy Story 3", "Lee Unkrich", 2010)
]

```

- This does work, but that type annotation is getting very hard to read.
- That's a problem, because one of the benefits of using type annotations is to help with readability.
- In cases like this where we have complex type annotations, it's often better to define new aliases for certain type combinations. For example, it makes a lot of sense to call each of these tuples a ***Movie***.
- We can do this like so:

```python
from typing import List, Tuple

Movie = Tuple[str, str, int]

movies: List[Movie] = [
    ("Finding Nemo", "Andrew Stanton", 2005),
    ("Inside Out", "Pete Docter", 2015),
    ("Toy Story 3", "Lee Unkrich", 2010)
]
```

Annotating functions

- Now let's get into annotating functions, which is where this kind of tool is most useful.
- The function below is for the subtraction of an integer value, which subtracts two integers and returns the value.
- Here the function above needs to accept two integers 'x' and 'y', but since the rules are not imposed, can take any data type.

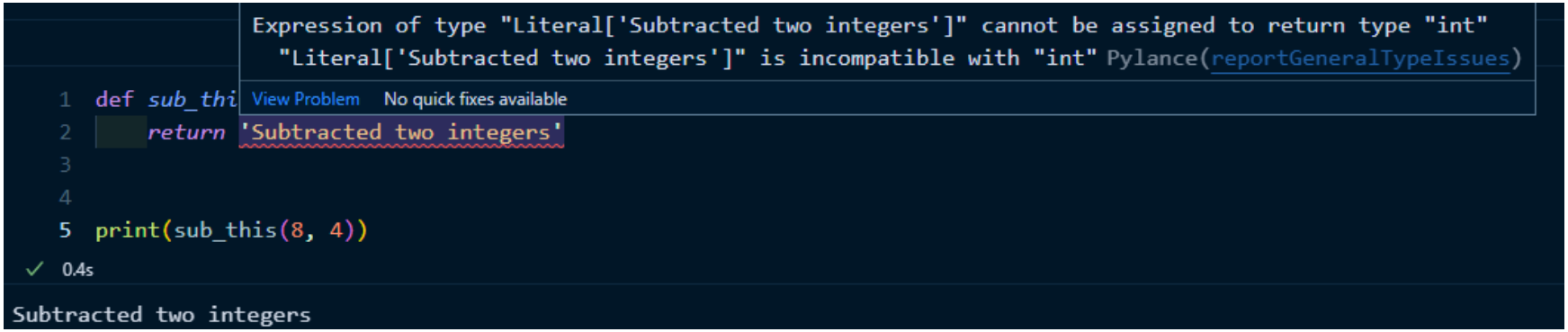
```
1  def sub_this(x, y):
2      return 'Subtraction'
3
4  print(sub_this(8, 'hello'))
5
✓ 0.4s
Subtraction
```

- Also, the return value can be anything where the 'str' value('Subtraction') is returned, but expected was 'int'.
- Let's see the similar implementation of the above program, but by using type hints which can help to implement static type checking and reduce error and bugs in the program quite easily.
- The below code is a simple program that accepts two integers as an input in the parameter and after '**->**' shows the returned data type, which is also an 'int'. However, the function needs to return int, but string 'Subtracted two integers' is returned.
-

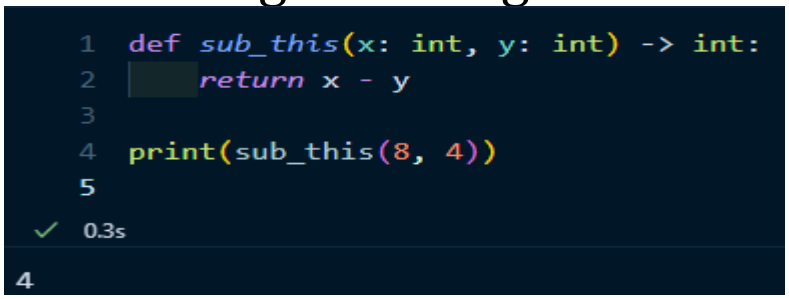
```
1  def sub_this(x: int, y: int) -> int:
2      return 'Subtracted two integers'
3
4
5  print(sub_this(8, 4))
6
✓ 0.5s
Subtracted two integers
```

Annotating functions

- Then the error will be shown, which indicates that the unexpected return value "str" is found and needs to be resolved with "int".



- Let's change the return type to be the subtraction of the two integers so that the integer value gets returned.



- The above results show that the success message gets printed out, and no issues were found.

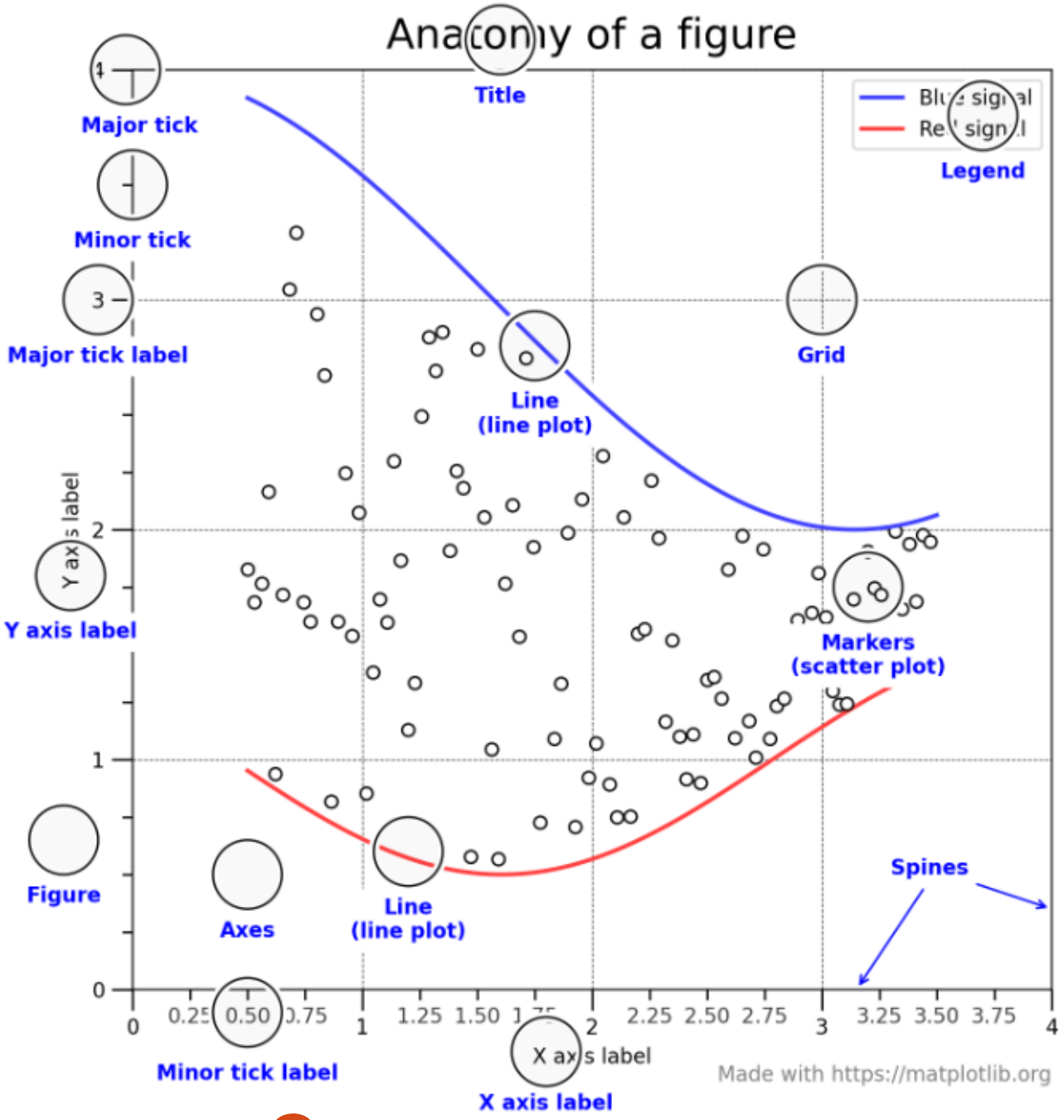
Visualizing Data

- A fundamental part of the data scientist's toolkit is data visualization.
- Data visualization is essential in today's world because it helps people understand complex data quickly and efficiently.
- Here are some key reasons why it is important:
- **Simplifies Complex Data**
- **Enhances Decision-Making**
- **Identifies Trends and Patterns**
- **Saves Time**

matplotlib

- A wide variety of tools exist for visualizing data. We will be using the matplotlib library, which is widely used
- We will be using the ***matplotlib.pyplot*** module.
- In its simplest use, ***pyplot*** maintains an internal state in which you build up a visualization step by step.
- Once you're done, you can save it with ***savefig*** or display it with ***show***.

Matplotlib


Anatomy of a figure
Title
Major tick
Minor tick
Major tick label
Legend
Grid
Line
(line plot)
Y axis label
Markers
(scatter plot)
Figure
Axes
Line
(line plot)
Spines
Minor tick label
X axis label
Made with https://matplotlib.org

Lists

- Probably the most **fundamental data structure in Python is the list,** which is simply an ordered collection.

```
integer_list = [1, 2, 3]
heterogeneous_list = ["string", 0.1, True]
list_of_lists = [integer_list, heterogeneous_list, []]

list_length = len(integer_list)      # equals 3
list_sum    = sum(integer_list)      # equals 6
```

- You can get or set the nth element of a list with square brackets:

```
x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

zero = x[0]          # equals 0, lists are 0-indexed
one = x[1]           # equals 1
nine = x[-1]         # equals 9, 'Pythonic' for last element
eight = x[-2]        # equals 8, 'Pythonic' for next-to-last element
x[0] = -1            # now x is [-1, 1, 2, 3, ..., 9]
```

Lists

- If you don't want to modify x, you can use list addition:

```
x = [1, 2, 3]
y = x + [4, 5, 6]        # y is [1, 2, 3, 4, 5, 6]; x is unchanged
```

- More frequently we will append to lists one item at a time:

```
x = [1, 2, 3]
x.append(0)        # x is now [1, 2, 3, 0]
y = x[-1]          # equals 0
z = len(x)         # equals 4
```

- It's often convenient to unpack lists when you know how many elements they contain:

```
x, y = [1, 2]     # now x is 1, y is 2
```

although you will get a *ValueError* if you don't have the same number of elements on both sides.

- A common idiom is to use an underscore for a value you're going to throw away:

```
_, y = [1, 2]     # now y == 2, didn't care about the first element
```

Modules

- You might also do this if your module has an unwieldy name or if you're going to be typing it a lot.
- For example, a standard convention when visualizing data with matplotlib is:

```
import matplotlib.pyplot as plt

plt.plot(...)
```

If you need a few specific values from a module, you can import them explicitly and use them without qualification:

```
from collections import defaultdict, Counter
lookup = defaultdict(int)
my_counter = Counter()
```

Matplotlib

The syntax of the `plot` function is `plot(xl, yl, options)` where:

- `xl` is the list of x-coordinates of points we want to plot
- `yl` is the list of y-coordinates
- other options may specify how we want the plot to look like.

function	usage
`plot`	creates a plot
`title`	sets a title of the plot
`xlabel`	sets label of the x-axis
`ylabel`	sets label of the y-axis
`xlim`	set the range of x values displayed
`ylim`	set the range of y values displayed
`show`	displays the plot

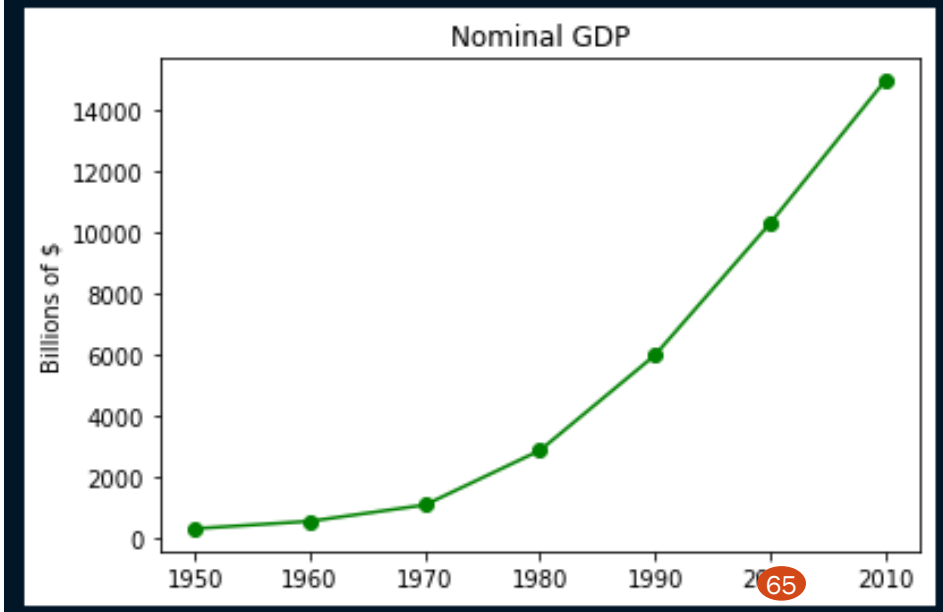
shape	meaning	color	meaning
`'-'`	solid line	`'b'`	blue
`'--'`	dashed line	`'g'`	green
`'.'`	point marker	`'r'`	red
`'o'`	circle marker	`'c'`	cyan
`'s'`	square marker	`'w'`	white
`'+'`	plus marker	`'m'`	magenta
`'x'`	x marker	`'y'`	yellow
`'D'`	diamond marker	`'k'`	black

Visualizing Data

```python
from matplotlib import pyplot as plt
years = [1950, 1960, 1970, 1980, 1990, 2000, 2010]
gdp = [300.2, 543.3, 1075.9, 2862.5, 5979.6, 10289.7, 14958.3]
# create a line chart, years on x-axis, gdp on y-axis
plt.plot(years, gdp, color='green', marker='o', linestyle='solid')
# add a title
plt.title("Nominal GDP")
# add a label to the y-axis
plt.ylabel("Billions of $")
plt.show()

```

✓ 8.7s

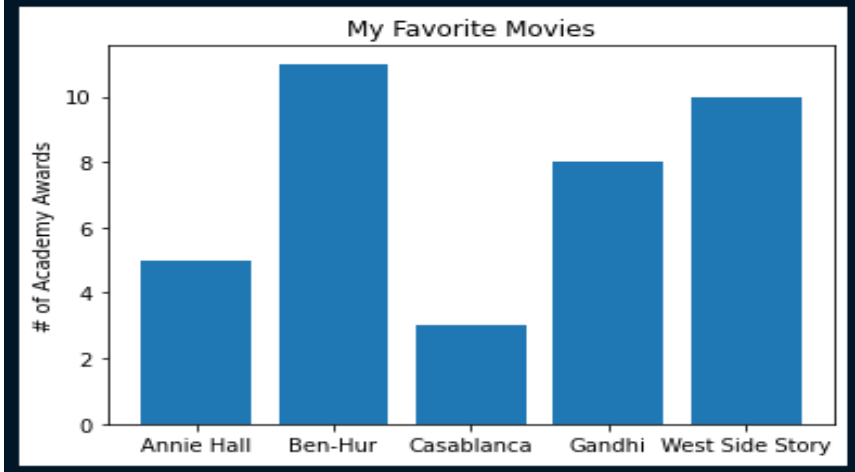


Bar Charts

- A bar chart is a good choice when you want to show how some quantity varies among some discrete set of items.
- ***Syntax: plt.bar(x, height, width, bottom, align)***
- For instance, the below figure shows how many Academy Awards were won by each of a variety of movies.

```
1  from matplotlib import pyplot as plt
2  movies = ["Annie Hall", "Ben-Hur", "Casablanca", "Gandhi", "West Side Story"]
3  num_oscars = [5, 11, 3, 8, 10]
4  # plot bars with left x-coordinates [0, 1, 2, 3, 4], heights [num_oscars]
5  plt.bar(range(len(movies)), num_oscars)
6  plt.title("My Favorite Movies")  # add a title
7  plt.ylabel("# of Academy Awards")  # label the y-axis
8  # label x-axis with movie names at bar centers
9  plt.xticks(range(len(movies)), movies)
10 plt.show()
```

✓ 0.1s

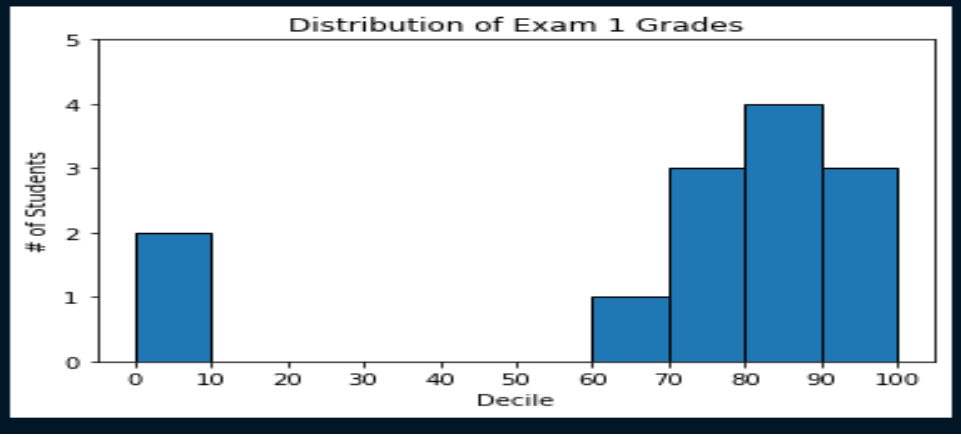


Bar Charts: Histograms

- A bar chart can also be a good choice for plotting histograms of bucketed numeric values, as in the below figure, in order to visually explore how the values are distributed.

```
from matplotlib import pyplot as plt
from collections import Counter
grades = [83, 95, 91, 87, 70, 0, 85, 82, 100, 67, 73, 77, 0]
# Bucket grades by decile, but put 100 in with the 90s
histogram = Counter(min(grade // 10 * 10, 90) for grade in grades)
plt.bar([x + 5 for x in histogram.keys()],  # Shift bars right by 5
        histogram.values(),  # Give each bar its correct height
        10,  # Give each bar a width of 10
        edgecolor=(0, 0, 0))  # Black edges for each bar
plt.axis([-5, 105, 0, 5])  # x-axis from -5 to 105,
# y-axis from 0 to 5
plt.xticks([10 * i for i in range(11)])  # x-axis labels at 0, 10, ...,100
plt.xlabel("Decile")
plt.ylabel("# of Students")
plt.title("Distribution of Exam 1 Grades")
plt.show()
```

0.1s



Bar Charts

- The third argument to ***plt.bar*** specifies the bar width. Here we chose a width of 10, to fill the entire decile.
- We also shifted the bars right by 5, so that, for example, the "10" bar (which corresponds to the decile 10–20) would have its center at 15 and hence occupy the correct range.
- We also added a black edge to each bar to make them visually distinct.
- The call to ***plt.axis*** indicates that we want the x-axis to range from –5 to 105 (just to leave a little space on the left and right), and that the y-axis should range from 0 to 5.
- And the call to ***plt.xticks*** puts x-axis labels at 0, 10, 20, …, 100.

Control Flow

- As in most programming languages, you can perform an action conditionally using if:

```
if 1 > 2:
    message = "if only 1 were greater than two..."
elif 1 > 3:
    message = "elif stands for 'else if'"
else:
    message = "when all else fails use else (if you want to)"
```

- You can also write a ternary if-then-else on one line, which we will do occasionally:

```
parity = "even" if x % 2 == 0 else "odd"
```

- Python has a while loop:

```
x = 0
while x < 10:
    print(f"{x} is less than 10")
    x += 1
```

```
1  x = 3
2  if x % 2 == 0:
3      parity = "even"
4  else:
5      parity = "odd"
6  print(parity)
7
```

✓ 0.4s

odd

```
1  x = 3
2  parity = "even" if x % 2 == 0 else "odd"
3  print(parity)
4
```

✓ 0.4s

odd

Control Flow

- for and in:

```
# range(10) is the numbers 0, 1, ..., 9
for x in range(10):
    print(f"{x} is less than 10")
```

- If you need more complex logic, you can use continue and break:

```
for x in range(10):
    if x == 3:
        continue  # go immediately to the next iteration
    if x == 5:
        break     # quit the loop entirely
    print(x)
```

- This will print 0, 1, 2, and 4.

zip and Argument Unpacking

- Often we will need to zip two or more iterables together.
- The ***zip*** function transforms multiple iterables into a single iterable of tuples of corresponding function:

```
1  list1 = ['a', 'b', 'c']
2  list2 = [1, 2, 3]
3  # zip is lazy, so you have to do something like the following
4  [pair for pair in zip(list1, list2)]
✓ 0.7s
[('a', 1), ('b', 2), ('c', 3)]
```

- If the lists are different lengths, zip stops as soon as the first list ends.
- You can also "unzip" a list using a strange trick:

```
pairs = [('a', 1), ('b', 2), ('c', 3)]
letters, numbers = zip(*pairs)
```

- The asterisk (*) performs argument unpacking, which uses the elements of pairs as individual arguments to zip.

List Comprehensions

- List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

The Syntax

newlist = [expression for item in iterable if condition == True]

- The return value is a new list, leaving the old list unchanged.

Condition

- The condition is like a filter that only accepts the items that valuate to True.

Iterable

- The iterable can be any iterable object, like a list, tuple, set etc.

Expression

- The expression is the current item in the iteration, but it is also the outcome, which you can manipulate before it ends up like a list item in the new list.

List Comprehensions

- Frequently, you'll want to transform a list into another list by choosing only certain elements, by transforming elements, or both.

```
even_numbers = [x for x in range(5) if x % 2 == 0]  # [0, 2, 4]
squares      = [x * x for x in range(5)]            # [0, 1, 4, 9, 16]
```

- list comprehensions:

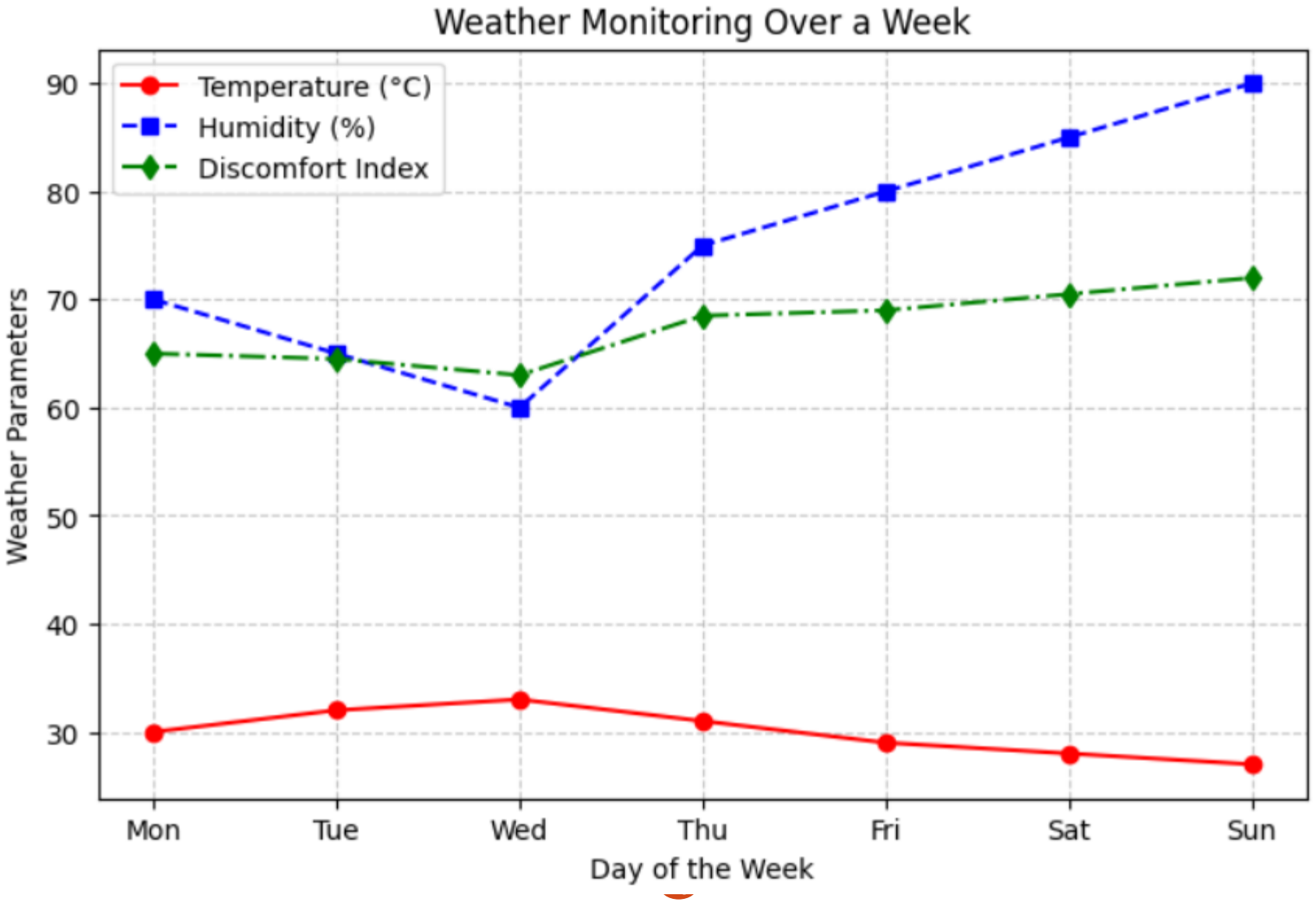
```
even_squares = [x * x for x in even_numbers]        # [0, 4, 16]

square_dict = {x: x * x for x in range(5)}  # {0: 0, 1: 1, 2: 4, 3: 9, 4:
16}
square_set  = {x * x for x in [1, -1]}      # {1}
```

Multiple Line Charts: using plot()

```
import matplotlib.pyplot as plt
# Sample data: Days of the week
days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
temperature = [30, 32, 33, 31, 29, 28, 27] # Temperature (°C)
humidity = [70, 65, 60, 75, 80, 85, 90] # Humidity (%)
# Discomfort Index (A rough measure of heat discomfort:Temp + 0.5 * Humidity)
discomfort_index = [t + 0.5 * h for t, h in zip(temperature, humidity)]
plt.figure(figsize=(8, 5)) # Create the line plot
plt.plot(days, temperature, 'r-o', label="Temperature (°C)")  # Red solid line with circles
plt.plot(days, humidity, 'b--s', label="Humidity (%)")  # Blue dashed line with squares
plt.plot(days, discomfort_index, 'g-.d', label="Discomfort Index")  # Green dot-dashed line with diamonds
plt.xlabel("Day of the Week")
plt.ylabel("Weather Parameters")
plt.title("Weather Monitoring Over a Week")
plt.legend() # Adding legend
plt.grid(True, linestyle="--",) # Grid for better readability
plt.show()
```

Multiple Line Charts



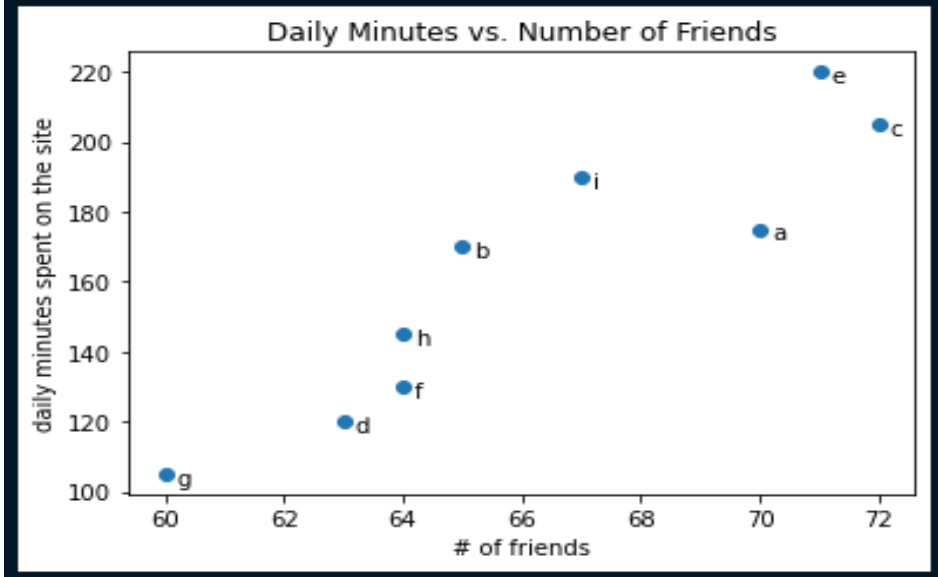
Scatterplots

- A scatterplot is the right choice for visualizing the relationship between two paired sets of data.
- For example, the below figure illustrates the relationship between the number of friends your users have and the number of minutes they spend on the site every day:

```python
from matplotlib import pyplot as plt
friends = [70, 65, 72, 63, 71, 64, 60, 64, 67]
minutes = [175, 170, 205, 120, 220, 130, 105, 145, 190]
labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
plt.scatter(friends, minutes)
# label each point
for label, friend_count, minute_count in zip(labels, friends, minutes):
    plt.annotate(label,
                 xy=(friend_count, minute_count),  # Put the label with its point
                 xytext=(5, -5),  # but slightly offset
                 textcoords='offset points')
plt.title("Daily Minutes vs. Number of Friends")
plt.xlabel("# of friends")
plt.ylabel("daily minutes spent on the site")
plt.show()

```

✓ 0.1s



Linear Algebra

- Linear algebra is the branch of mathematics that deals with vector spaces.

Vectors

- Physical quantity with magnitude and no direction is known as a **scalar** quantity and a physical quantity with magnitude and directions is known as a **vector** quantity.
- Vectors are objects that can be added together to form new vectors and that can be multiplied by scalars (i.e., numbers), also to form new vectors
- Vectors, are often a useful way to represent numeric data.
- **For example**, if you have the heights, weights, and ages of a large number of people, you can treat your data as **three-dimensional vectors** [height, weight, age].
- height_weight_age = [70, # *inches,*
  170, # *pounds,*
  40] # *years*

Vectors

For example, in a 2D space, a vector might look like:

$$v = [3, 4]$$

In a 3D space:

$$w = [1, -2, 5]$$

Vectors

Size (sq.ft)	Number of Rooms	Price ($)
850	2	180,000
1,200	3	250,000
1,500	3	300,000
1,800	4	350,000
2,000	4	400,000
2,500	5	500,000
1,100	2	210,000
1,400	3	275,000
1,700	4	320,000
2,200	4	450,000

Vectors

Uses of Vectors in Data Science

Vectors are fundamental in **data science and machine learning**. Here's why:

1. Representing Data Points

- Each **data point** in a dataset can be represented as a vector.
- Example: A house dataset with features:

$$\text{House} = [\text{Size (sq ft)}, \text{Number of rooms}, \text{Price}]$$

$$[1500, 3, 200000]$$

Each row is a **vector** in a high-dimensional space.

2. Feature Engineering

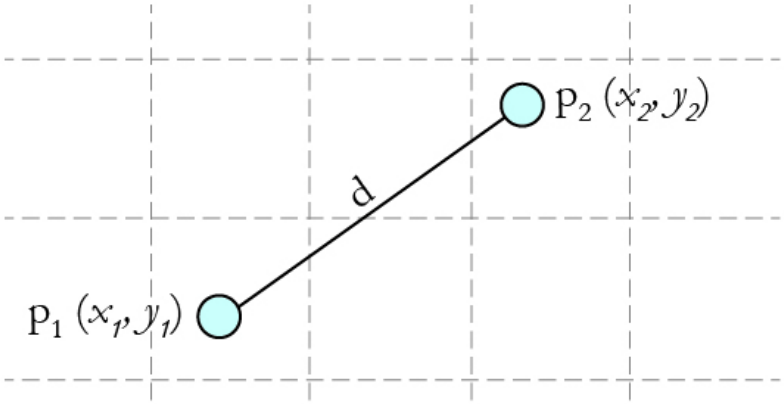
- In machine learning, vectors are used to represent **features** (input variables).
- Example: In an **image dataset**, a 28x28 grayscale image can be represented as a **784-dimensional vector** (flattened pixels).

Vectors

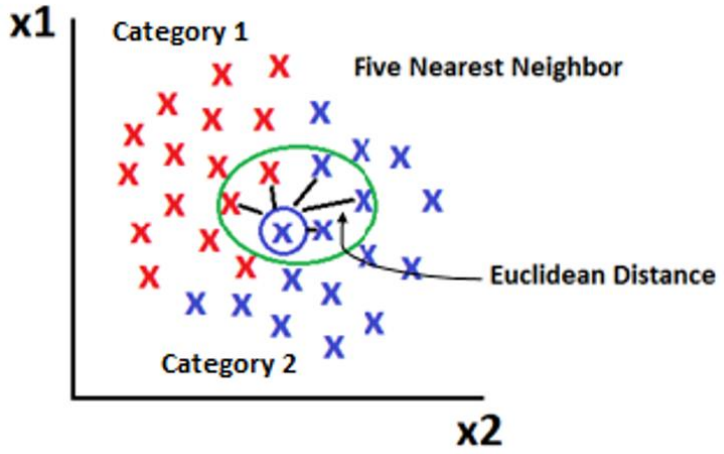
3. Distance Calculation (Similarity & Clustering)

- **Euclidean distance** between two vectors is used to measure similarity.
- Example: In **K-Nearest Neighbors (KNN)**, vectors represent points in space, and the algorithm finds the closest ones.

$$\text{Distance}(v, w) = \sqrt{(v_1 - w_1)^2 + (v_2 - w_2)^2 + ... + (v_n - w_n)^2}$$



Euclidean distance (d) $= \sqrt{(x_2-x_1)^2 + (y_2-y_1)^2}$

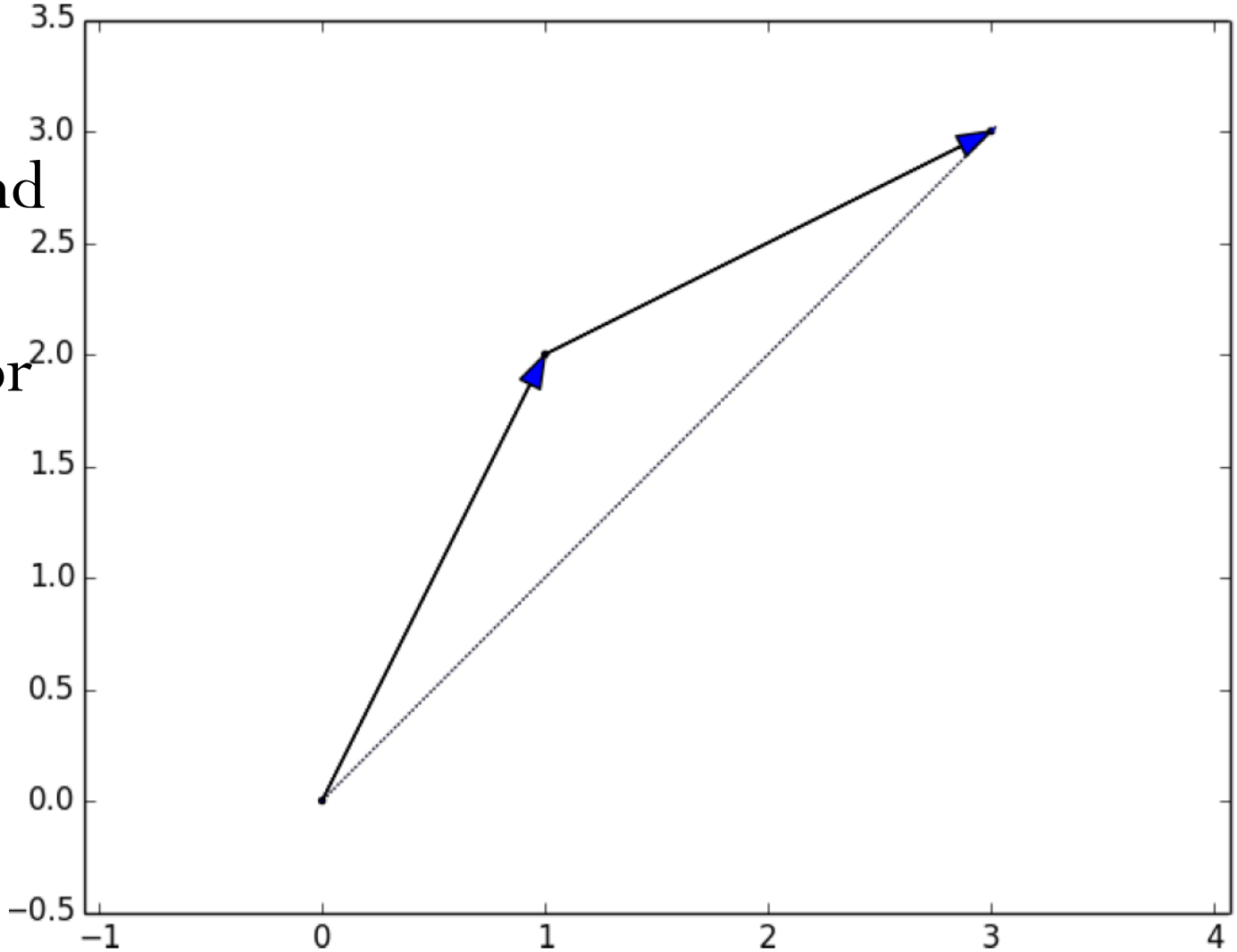


Vectors

- **Perform arithmetic on vectors.**
- Because Python lists aren't vectors (and hence provide no facilities for vector arithmetic), we'll need to build these arithmetic tools ourselves.
- Add two vectors. Vectors add component wise.
- This means that if two vectors **v and w** are the same length, their sum is just the vector whose first element is v[0] + w[0], whose second element is v[1] + w[1], and so on.
- (If they're not the same length, then we're not allowed to add them.)

Adding two Vectors

For example, adding the vectors [1, 2] and [2, 1] results in [1 + 2, 2 + 1] or [3, 3], as shown in Figure 4-1.



Adding two vectors

Add two Vectors

- If the Vectors are of not the same length

```
def vector_add(v, w):
  """adds corresponding elements"""
  return [v_i + w_i   for v_i, w_i in zip(v, w)]
print(vector_add([5, 7, 9], [4, 5, 6]))
```

Subtract two Vectors

- To subtract two vectors we just subtract the corresponding elements:

```
def vector_subtract(v, w):
  """subtracts corresponding elements"""
  return [v_i - w_i   for v_i, w_i in zip(v, w)]
print(vector_subtract([5, 7, 9], [4, 5, 6]))
```

Component wise sum a list of Vectors

- We'll also sometimes want to component wise sum a list of vectors—that is, create a new vector whose first element is the sum of all the first elements, whose second element is the sum of all the second elements, and so on:

```
def vector_sum(vectors):
    """sums all corresponding elements"""
    result = vectors[0] # start with the first vector
    for vector in vectors[1:]: #loop over the others
        result = vector_add(result, vector) #add to result
        return result
print(vector_sum([[1, 2], [3, 4], [5, 6], [7, 8]]))
```

Multiply a vector by a scalar

- We'll also need to be able to multiply a vector by a scalar, which we do simply by multiplying each element of the vector by that number:

```
def scalar_multiply(c, v):
    """c is a number, v is a vector"""
    return [c * v_i for v_i in v]
print(scalar_multiply(2, [1, 2, 3]))
```

Compute Component wise Means

- This allows us to compute the component wise means of a list of (same sized) vectors:

```
def vector_mean(vectors):
  """compute the vector whose ith element is the mean of the
  ith elements of the input vectors"""
  n = len(vectors)
  return scalar_multiply(1/n, vector_sum(vectors))
print(vector_mean([[1, 2], [3, 4], [5, 6]]))
```

```
[1.3333333333333333, 2.0]
```

Dot Product

- The dot product of two vectors is the sum of their component wise products:

1. Dot Product Formula

The **dot product** of two vectors v and w in an **n-dimensional space** is given by:

$$v \cdot w = v_1 \cdot w_1 + v_2 \cdot w_2 + \ldots + v_n \cdot w_n$$

In summation notation:

$$v \cdot w = \sum_{i=1}^{n} v_i \cdot w_i$$

where:

- v and w are vectors.
- v_i and w_i are the **corresponding elements** of each vector.

Dot Product

- The dot product of two vectors is the sum of their component wise products:

2. Example Calculation

Let's calculate the dot product of the two vectors:

$$v = [1, 2, 3], \quad w = [4, 5, 6]$$

Using the formula:

$$(1 \times 4) + (2 \times 5) + (3 \times 6)$$

$$= 4 + 10 + 18 = 32$$

So, the dot product $v \cdot w = 32$.

Dot Product

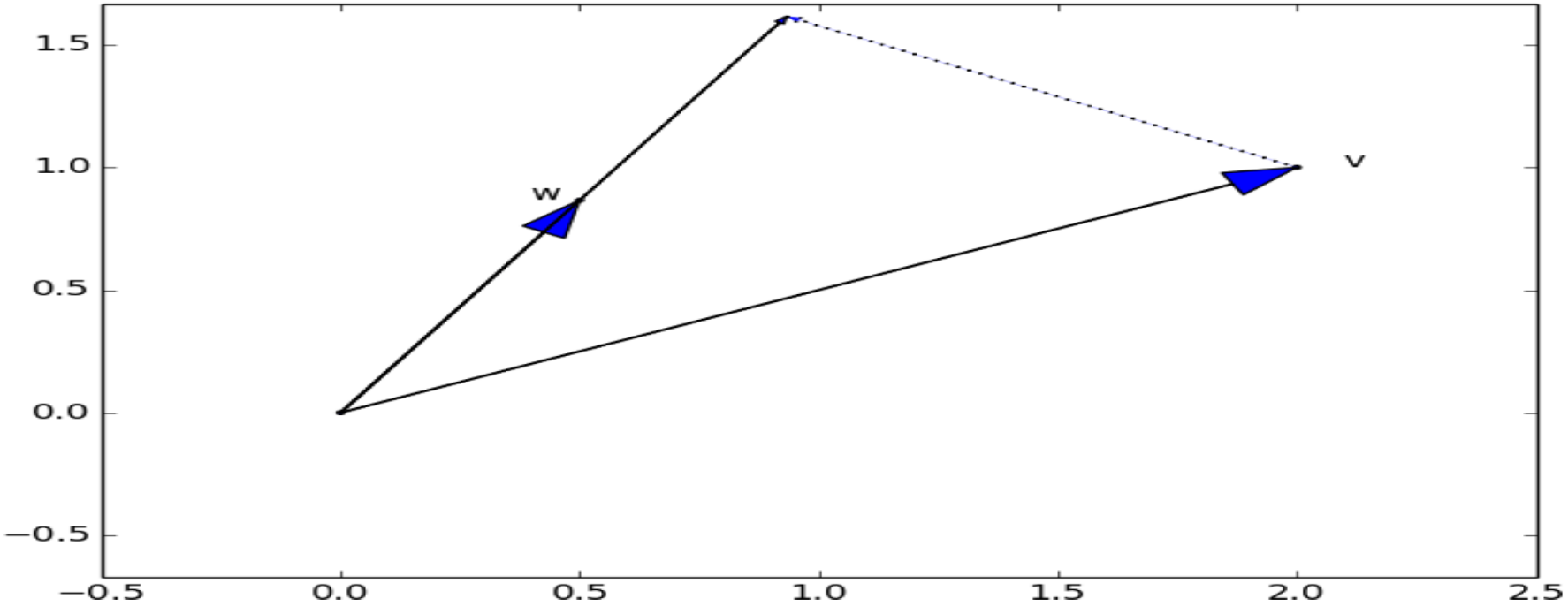
- The dot product of two vectors is the sum of their component wise products:

```
def dot(v, w):
  """v_1 * w_1 + ... + v_n * w_n"""
  return sum(v_i * w_i for v_i, w_i in zip(v, w))
print(dot([1, 2, 3], [4, 5, 6]))
```

```
32
```

Dot Product

- If w has magnitude 1, the dot product measures how far the vector v extends in the w direction.
- For example, if ***w = [1, 0]***, then ***dot(v, w)*** is just the first component of v.
- Another way of saying this is that it's the length of the vector you'd get if you projected ***v*** onto ***w***.



The dot product as vector projection

Sum of Squares

- Using this, it's easy to compute a vector's sum of squares:

```
def sum_of_squares(v):
  """v_1 * v_1 + ... + v_n * v_n"""
  return dot(v, v)
print(sum_of_squares([1, 2, 3]))
```

Compute Magnitude(length)

- Using this, it's easy to compute a vector's sum of squares:

Mathematical Formula of Magnitude (Euclidean Norm)

For a vector **v** with **n** components:

$$v = [v_1, v_2, ..., v_n]$$

The magnitude (or Euclidean norm) is given by:

$$||v|| = \sqrt{v_1^2 + v_2^2 + ... + v_n^2}$$

Example Calculation

Let's take a vector:

$$v = [3, 4]$$

Using the formula:

$$||v|| = \sqrt{3^2 + 4^2} = \sqrt{9 + 16} = \sqrt{25} = 5$$