Data Types for Data Science in Python
by datacamp

Container sequences
used for aggregation, sorting, order, and more
such as tuples, lists, sets, and more
some are mutable (like lists and sets) and some are immutable (like tuples)
immutability allows us to protect our reference data
immutability also allows us to replace individual data points with sums, averages, etc
iterate, also called looping

Lists
hold data in order it was added
mutable
access an individual element using an index
add to the list using the .append() method
example
starting list > cookies = [‘chocolate chip’, ‘peanut butter’, ‘sugar’]
cookies.append(‘Tirggel’)
print(cookies)
output > [‘chocolate chip’, ‘peanut butter’, ‘sugar’, ‘Tirggel’]

Combining lists
can use operators like + to add lists together
.extend() method merges a list into another list at the end

Finding elements in a list
.index() methond locates the position of a data element in a list
example
position = cookies.index(‘sugar’)
print(position)
output = 3

Removing elements in a list
.pop() method removes an item from a list and allows you to save it
example
pass the above index
name = cookies.pop(position)
print(name)
output > sugar

**now print(cookies) > ouput has sugar removed

Iterating over lists
list comprehensions are a common way of iterating over a list to perform some action on them
allows us to work on that list one element at a time
titlecase_cookies = [cookie.title() for cookie in cookies]
print(titiecase_cookies)
output > Chocolate Chip, Peanut Butter, Tirggel

Sorting lists
.sorted() function sorts data in numerical or alphabetical order and returns a new list

examples
```
# Create a list containing the names: baby_names
baby_names = ['Ximena', 'Aliza', 'Ayden', 'Calvin']

# Extend baby_names with 'Rowen' and 'Sandeep'
baby_names.extend(['Rowen', 'Sandeep'])
print(baby_names)

# Find the position of 'Rowen': position
position = baby_names.index('Rowen')

# Remove 'Rowen' from baby_names
baby_names.pop(position)

# Create the list comprehension: baby_names
baby_names = [row[3] for row in records]

# Print the sorted baby names in ascending alphabetical order
print(sorted(baby_names))
```

Tuples
like lists in that they hold data in order and you can access elements inside a tuple with an index
tuples are easier to process and more memory efficient than lists
tuples are immutable (you cannot add or remove elements from them)
this means we can use them to ensure that our data is not altered
can create tuples by pairing up elements
can ‘unpack’ to expand a tuple into named variables that represent each element in the tuple

Zipping tuples
often we'll have lists where we want to matchup elements into pairs
zip function enables us to do that

```
top_pairs = list(zip(us_cookies, in_cookies))
print(top_pairs)
```

```
[('Chocolate Chip', 'Punjabi'), ('Brownies', 'Fruit Cake Rusk'),
('Peanut Butter', 'Marble Cookies'), ('Oreos', 'Kaju Pista Cookies'),
('Oatmeal Raisin', 'Almond Cookies')]
```

we have two lists, one for the most popular cookies in India and another for the most popular cookies in US
want to build a list of pairs by the popularity rank of the cookie in each country
outputs a list of tuples (this is really an iterator)
tuples use parentheses as their object representation

Unpacking tuples (also called tuple expansion)
allows us to assign the elements of a tuple to a named variable for later use
this syntax allows us to create more readable and less error prone code
example
us_num_1, in_num_1 = top_pairs[0]
print(us_num_1)
output > Chocolate Chip
print(in_num_1)
output > Punjabi

Tuple unpacking with for loops
separate a list of tuples into their elements as we loop over them
for us_cookie, in_cookie in top_pairs:
#this splits each tuple in the list into its Indian and US cookie elements
    print(in_cookie)
    print(us_cookie
output > prints number 1 India cookie then number 1 US cookie then number 2 India cookie then number 2 US cookie and so on

Enumerating positions
often we want to know what the indext is of an element in the iterable is
enumerate function enabled us to do that by creating tuples where the first element of the tuple is the index of the element in the original list, then the element itself

enumeration is used in loops to return the position and the data in that position while looping
we can use this to track rankings in our data or skip elements we are not interested in
example

```
for idx, item in enumerate(top_pairs):
    us_cookie, in_cookie = item
    print(idx, us_cookie, in_cookie)
```

```
(0, 'Chocolate Chip', 'Punjabi')
(1, 'Brownies', 'Fruit Cake Rusk')
# ..etc..
```

example

```
# Pair up the girl and boy names: pairs
pairs = list(zip(girl_names, boy_names))

# Iterate over pairs
for rank, pair in enumerate(pairs):
    # Unpack pair: girl_name, boy_name
    girl_name, boy_name = pair
    # Print the rank and names associated with each rank
    print(f'Rank {rank+1}: {girl_name} and {boy_name}')
```

Strings
can loop over them
several different string types indicated by a letter in front of the opening quote of the string
f-strings (formatted string literals) - f""
allows you to place an f in front of the opening quote, and then you can use python expressions wrapped in curly braces inside them to access additional data points and incorporate them into the string
example
cookie_name = 'Anzac'
cookie_price = '$1.99'
print(f'Each {cookie_name} cookie cost {cookie_price}.')
output > 'Each Anzac cookie costs $1.99.'

.join method
“”.join() uses the string it’s called on to join an iterable
example
child_ages = [‘3’, ‘4’, ‘5’, ‘6’]
print(‘, ‘.join(child_ages)
output > ‘3, 4, 5, 6’
tood four elements joined them into a string with a space between each element
can also use indexing elements
example
print(f’The children are ages {‘,’. join(child_ages[0:3])}, and {child_ages[-1]}.’)
ouput > ‘The children are ages 3, 4, 7, and 8.’

Matching parts of a string
finding strings within strings
.startswith() and .endswith() tell you if a string starts or ends with another character or string
example
boy_names = [‘Mohamed’, ‘Youssef’, ‘Ahmed’]
print([name for name in boy_names if name.startswith(‘A’)])
output > [‘Ahmed’]
**be careful as these and most string functions are case-sensitive

**’in’ operator
the in operator searchs for some value in some iterable type like a string
example
‘long’ in ‘Life is a long lesson in humility.’
output > True
‘life’ in ‘Life is a long lesson in humility.’
output > False
*because case sensitive
can fix this with .lower() method
‘life’ in Life is a long lesson in humility.’.lower()
output > True

example
# The top ten boy names are: as preamble
preamble = "The top ten boy names are: "

# , and as conjunction
conjunction = ', and'

Combines the first 9 names in boy_names with a comma and space as

first_nine_names
first_nine_names = ", ".join(boy_names[0:9])

# Print f-string preamble, first_nine_names, conjunction, the final item in boy_names and a period
print(f"{preamble}{first_nine_names}{conjunction} {boy_names[-1]}.")

***list comprehension > this is what it is saying > [action for item in list if somethin is true]

example
# Store a list of girl_names that start with s: names_with_s
names_with_s = [name for name in girl_names if name.lower().startswith('s')]

print(names_with_s)

# Store a list of girl_names that contain angel: names_with_angel
names_with_angel = [name for name in girl_names if 'angel' in name.lower()]

print(names_with_angel)

Dictionaries
useful for storing key/value pair, grouping data by time or structuring hierarchical data like org charts
*key must be alphanumeric but the value can be any other data type
nestable > can use a dictionary as the value of a key within a dictionary
can also iterate over the keys and values of a dictionary
*can also iterate over the items of a dictionary, which are tuples of the key and value pairs
can create dictionaries with dict() or more common shortcut {}
nice example - list of tuples containing the name and zip for New York Art Galleries, turn into a dictionary
#create an empty dicitonary
art_galleries = {}
#next use tuple unpacking as we loop over the galleries in the list that contain the data
for name, zip_code in galleries:
#inside the loop set the name of the gallery as the key in my dictionary and the zip code as the value
    art_galleries[name] = zip_code
#find the last 5 art gallery names
#*by default when using sorted or looping over a dictionary, we loop over the keys
for name in sorted(art_galleries)[-5:]:

#print the keys which are the names
    print(name)

Get a value from a dictionary by using the key as an index
*getting/finding keys safely
.get() method allows you to safely access a key without error or exception handling
you want this to ensure your programs execute properly
*if a key is not in the dicitonary, .get() returns 'None' by default or you can supply a value to return
example
art_galleries.get('Lourve', 'Not Found')
output > 'Not Found' (Lourve is not in the dictionary so Python returns 'Not Found')

Example
# Create an empty dictionary: squirrels_by_park
squirrels_by_park = {}

# Loop over the squirrels list and unpack each tuple
for park, squirrel_details in squirrels:
    # Add each squirrel_details to the squirrels_by_park dictionary
    squirrels_by_park[park] = squirrel_details

# Sort the names_by_rank alphabetically dict by park
for park in sorted(squirrels_by_park):
    # Print each park and it's value in squirrels_by_park
    print(f'{park}: {squirrels_by_park[park]}')

Adding to a dictionary
add a new key/value to a dictionary
can also supply a dictionary, list of tuples, or a set of keyword arguments to the update() method to add values into a dictionary
example - adding to above example dictionary (art galleries)
#new dictionary called galleries_10007
#add it to art_galleries
art_galleries['10007'] = galleries_10007
#adding tuples to the dictionary
#new tuple
galleries_11234 = [('AJ Arts LTD', '718) 763-5473'), Doug Meyer Fine Art', '(718) 375-8006')]
art_galleries['11234'].update(galleries_11234)

Popping and deleting from dictionaries
del instruction deletes a key/value, however if key is not found then a KeyError will get thrown
del art_galleries[‘11234’]
.pop() method provides a safe way to remove keys from a dictionary
galleries_10310 = art_galleries.pop(‘10310’)

Example
Assign squirrels_madison as the value to the 'Madison Square Park' key
squirrels_by_park['Madison Square Park'] = squirrels_madison

# Update the 'Union Square Park' key with the squirrels_union tuple
squirrels_by_park.update([squirrels_union])

# Loop over the park_name in the squirrels_by_park dictionary
for park_name in squirrels_by_park:
    # Safely print a list of all primary_fur_colors for squirrels in the park_name
    print(park_name, [squirrel.get('primary_fur_color', 'N/A') for squirrel in squirrels_by_park[park_name]])

Pythonically using dictionaries
.items() method returns a dict_items object that we can iterate over as a list of key/value tuples
this is the preferred manner
#using tuple unpacking
for gallery, phone_num in art_galleries.items():
        print(gallery)
        print(phone_num)

Checking dictionaries for data
.get() does a lot of work to check for a key
the ‘in’ operator is much more efficient
’11234’ in art_galleries
output > boolean
operators that return booleans are often used in conditional statements
example
if ‘10010’ in art_galleries:
        print(‘I found: %s’ % art_galleries[‘10010’])
else:
        print(‘No galleries found.’)
output > I found: {‘Nyabinghi African Gift Shop’: ‘(212) 566-3336’}

example - using .items()

```
# Iterate over the first squirrel entry in the Madison Square Park list
for field, value in squirrels_by_park["Madison Square Park"][0].items():
    # Print field and value
    print(field, value)

print('-' * 13)

# Iterate over the second squirrel entry in the Union Square Park list
for field, value in squirrels_by_park['Union Square Park'][1].items():
    # Print field and value
    print(field, value)
```

example - using ‘in’ operator and conditional statements

```
# Check to see if Tompkins Square Park is in squirrels_by_park
if "Tompkins Square Park" in squirrels_by_park:
    # Print 'Found Tompkins Square Park'
    print('Found Tompkins Square Park')

# Check to see if Central Park is in squirrels_by_park
if "Central Park" in squirrels_by_park:
    # Print 'Found Central Park' if found
    print('Found Central Park')
else:
    # Print 'Central Park missing' if not found
    print('Central Park missing')
```

Mixed data types in dictionaries
keys() method to see the list of keys
**for example reorganized art_galleries dictionary to be keyed by zip code and then gallery name with value of their phone number
example accessing nested data

```
#accessing secondary index
art_galleries[‘10027’][‘Inner City Art Gallery Inc’]
```

output > ‘(212) 368-4941’
nesting dictionaries is a very common way to deal with repeating data structures
examples > yearly data, grouped or hierachical data such as organization reporting structures

example - pulling keys

```
# Print a list of keys from the squirrels_by_park dictionary
print(squirrels_by_park.keys())

# Print the keys from the squirrels_by_park dictionary for 'Union Square Park'
```

```
print(squirrels_by_park['Union Square Park'].keys())

# Loop over the dictionary
for park_name in squirrels_by_park:
    # Safely print the park_name and the highlights_in_fur_color or 'N/A'
    print(park_name, squirrels_by_park[park_name].get('highlights_in_fur_color', 'N/
A'))

# Use a for loop to iterate over the squirrels in Tompkins Square Park:
for squirrel in squirrels_by_park["Tompkins Square Park"]:
    # Safely print the activities of each squirrel or None
    print(squirrel.get("activities"))

# Print the list of 'Cinnamon' primary_fur_color squirrels in Union Square Park
print([squirrel for squirrel in squirrels_by_park["Union Square Park"] if "Cinnamon"
in squirrel["primary_fur_color"]])
```

Numeric data types
Integer > for whole numbers and large values
Float > for approximations (fractional amounts) and scientific notation

Decimals > for exact precision and currency operations
to use decimals need to import them
from decimal import Decimal
will not convert to scientific notation
Decimal()

Printing floats
default
print(0.00001)
output > scientific notation 1e-05
f-strings allow us to pass a format specifier
print(f'{0.00001:f}')
output > 0.000010
**need to be aware the bare float format specifier stops at six decimal places
what does that mean?
print(f'{0.0000001:f}')
output > 0.000000 #will only print out the first six decimal places
if you want more precision you have to specify further in the format specifier
print(f'{0.0000001:.7f}')
ouput > 0.0000001

Python division types

float division with single backslash (/)
4/2
output > 2.0
floored divsion (//)
7//3
output > 2
it floors the result

example - printing floats
```
# Use a for loop to iterate over the squirrels in Tompkins Square Park:
for squirrel in squirrels_by_park["Tompkins Square Park"]:
    # Safely print the activities of each squirrel or None
    print(squirrel.get("activities"))

# Print the list of 'Cinnamon' primary_fur_color squirrels in Union Square Park
print([squirrel for squirrel in squirrels_by_park["Union Square Park"] if "Cinnamon"
in squirrel["primary_fur_color"]])
```

Booleans
truthy values are ones that will return true
falsey values will evaluate to false
```
apples=2
if apples:
        print(“We have apples.’)
```
output > ‘We have apples.’)
```
apples=0
if apples:
        print(‘We have apples.’)
```
ouput > False

further examples of truthy and falsey
truthy > 1, ‘cookies’, [‘cake’, ‘pie’], {‘key’:’value’}
falsey > 0, “ ”, [], {}, None
*in general, something is truthy if it’s not empty of value

Be careful with floats and approximations
example
```
#remember we python operators are often used as booleans
x = 0.1 + 1.1
```
output > False
why?
```
print(x)
```
output > 1.2000000000000002

need to do
x == 1.2

example - evaluating truthiness
```
# Create an empty list
my_list = []

# Check the truthiness of my_list
print(bool(my_list))

# Append the string 'cookies' to my_list
my_list.append('cookies')

# Check the truthiness of my_list
print(bool(my_list))
```

example
```
# Use a for loop to iterate over the penguins list
for penguin in penguins:
  # Check the penguin entry for a body mass of more than 3300 grams
  if penguin["body_mass"] > 3300:
    # Print the species and sex of the penguin if true
    print(f"{penguin['species']} - {penguin['sex']}")
```

Sets
unordered data with optimized logic operations
excellent for finding all the unique values in a column of your data, a list of elements, or even rows from a file
use when we want to store unie data elements in an unordered fashion
mutable
many capabilities that align with set theory from math

Creating sets
almost always created from a list
once a list is made into a set only ‘unique’ items remain
example
```
cookies = [‘choco chip’, choco chip’, ‘oatmeal’]
types_of_cookies = set(cookies)
print(type_of_cookies)
output > set([‘choco chip’, ‘oatmeal’])
```

Modifying sets
use .add() method to add single elements

if we were to add >
example
types_of_cookies.add(‘choco chip’)
output > set([‘choco chip’, ‘oatmeal’])
if we were to add >
types_of_cookies.add(‘biscotti’)
ouput > set([‘choco chip’, ‘oatmeal’, ‘biscotti’])
**use update() method to add multiple items
merges in another set or list

Removing data from sets
.discard() safely removes an element from the set by value
*a KeyError will not be thrown if the value is not found
**.pop() works a little different here
removes and returns an arbitrary element from the set
*will be a KeyError is set is empty
why would you want this?
example - what cookie should I eat next?
types_of_cookies.pop()
output > ‘choco chip’

The power of sets
finding similarities
.union() method on a set acepts a set as an argumen adn returns all the unique elements from both sets as a new one
.intersection() method accepts a set and returns the overlppaing elements found in both sets
*this is great for comparing data year over year or month over month, etc
example

```
cookies_jason_ate = set(['chocolate chip', 'oatmeal cream',
'peanut butter'])
cookies_hugo_ate = set(['chocolate chip', 'anzac'])
cookies_jason_ate.union(cookies_hugo_ate)
```

```
set(['chocolate chip', 'anzac', 'oatmeal cream', 'peanut butter'])
```

```
cookies_jason_ate.intersection(cookies_hugo_ate)
```

```
set(['chocolate chip'])
```

finding differences
.difference() method accepts a set to find elements in one set that are not present in another set
**the ‘target’ we call the difference method on is important as that will be the basis for our differences
so here I first want to see what Jason ate and Hugo didn’t
then I want to see what Hugo ate and Jason didn’t

```
cookies_jason_ate.difference(cookies_hugo_ate)
```

```
set(['oatmeal cream', 'peanut butter'])
```

```
cookies_hugo_ate.difference(cookies_jason_ate)
```

```
set(['anzac'])
```

example
# Use a list comprehension to iterate over each penguin in penguins saved as female_species_list
# If the the sex of the penguin is 'FEMALE', return the species value
female_species_list = [penguin["species"] for penguin in penguins if penguin["sex"] == 'FEMALE']

# Create a set using the female_species_list as female_penguin_species
female_penguin_species = set(female_species_list)

# Find the difference between female_penguin_species and male_penguin_species. Store the result as differences
differences = female_penguin_species.difference(male_penguin_species)

# Print the differences
print(differences)

example - union and intersection
# Find the union: all_species
all_species = female_penguin_species.union(male_penguin_species)

```
# Print the count of names in all_species
print(len(all_species))

# Find the intersection: overlapping_species
overlapping_species =
female_penguin_species.intersection(male_penguin_species)

# Print the count of species in overlapping_species
print(len(overlapping_species))
```

Counting with Python
collections module is part of Python standard library
holds severar advanced data containers

Counter
special dictionaryt used for counting data, measuring frequency
powerful python object
based on the dictionary object
accepts a list and counts the number of times a value is found within the elements of that list
you can access it using all the normal dictionary features
example

```
from collections import Counter
nyc_eatery_count_by_types = Counter(nyc_eatery_types)
print(nyc_eatery_count_by_type)
```

output >

```
Counter({'Mobile Food Truck': 114, 'Food Cart': 74, 'Snack Bar': 24,
'Specialty Cart': 18, 'Restaurant': 15, 'Fruit & Vegetable Cart': 4})
```

```
print(nyc_eatery_count_by_types['Restaurant'])
```

output > 15

.most_common() method on a Counter returns the counter values in descending order
returns a list of tuples containing the items and their count
great for frequency analytics (how often something occurs)

```
print(nyc_eatery_count_by_types.most_common(3))
```

```
[('Mobile Food Truck', 114), ('Food Cart', 74), ('Snack Bar', 24)]
```

example - Counter with list comprehension

```
# Import the Counter object
from collections import Counter

# Create a Counter of the penguins sex using a list comp
penguins_sex_counts = Counter(penguin['Sex'] for penguin in penguins)

# Print the penguins_sex_counts
print(penguins_sex_counts)

# Import the Counter object
from collections import Counter

# Create a Counter of the penguins list: penguins_species_counts
penguins_species_counts = Counter(penguin['Species'] for penguin in penguins)

# Find the 3 most common species counts
print(penguins_species_counts.most_common(3))
```

Dictionaries of unknown structure

example - we want every key to have a list of values

```
#initialize every key with an empty list
#then add the values to the list
#start by looping over a list of tuples
for park_id, name in nyc_eateries_parks:
#check to see if we have a list for that park already in our dictionary
        if park_id not in eateries_by_park:
#if not create an empty list
                eateries_by_park[park_id] = [ ]
#append the name of the eatery to the list for that park id
        eateries_by_park[park_id].append(name)
print(eateries_by_park['M010'])
```

**an easier way to do this is using defaultdict

*defaultdict accepts a type that every value will default to if the key is not present in the dictionary

can override that typ by setting the key manually to a value of different type

example

create a list of eateries by park

data is tuples of park id and name of an eatery

```
from collections import defaultdict
#create dictionary that defaults to list
```

```
eateries_by_park = defaultdict(list)
#iterate over data and unpack it into park_id and name
for park_id, name in nyc_eateries_parks:
#append each eatery name into list for each park id
      eateries_by_park[park_id].append(name)
print(eateries_by_park['M010'])
```

another related and nice example
making dictionaries showing how many have websites and/or phone numbers

```
from collections import defaultdict
eatery_contact_types = defaultdict(int)
for eatery in nyc_eateries:
    if eatery.get('phone'):
        eatery_contact_types['phones'] += 1
    if eatery.get('website'):
        eatery_contact_types['websites'] += 1
print(eatery_contact_types)
```

```
defaultdict(<class 'int'>, {'phones': 28, 'websites': 31})
```

example
```
# Create an empty dictionary: female_penguin_weights
female_penguin_weights = {}

# Iterate over the weight_log entries
for species, sex, body_mass in weight_log:
   # Check to see if species is already in the dictionary
   if species not in female_penguin_weights:
      # Create an empty list for any missing species
      female_penguin_weights[species] = []
   # Append the sex and body_mass as a tuple to the species keys list
   female_penguin_weights[species].append((sex, body_mass))

# Print the weights for 'Adlie'
print(female_penguin_weights['Adlie'])
```

example - defaultdict

```
# Import defaultdict
from collections import defaultdict

# Create a defaultdict with a default type of list: male_penguin_weights
male_penguin_weights = defaultdict(list)

# Iterate over the weight_log entries
for species, sex, body_mass in weight_log:
    # Use the species as the key, and append the body_mass to it
    male_penguin_weights[species].append(body_mass)

# Print the first 2 items of the male_penguin_weights dictionary
print(list(male_penguin_weights.items())[:2])
```

Namedtuple
another Python container
nametuple which is a tuple
has names for each position of the tuple
when to use?
you don't need the nested structure of a dictionary
or desire each item to look identical
don't want to add the overhead of a Pandas DF row
create nametuple by passing a name for the tuple type and a list of field names
*common practice to use Pascalcase (capitalizing each word when naming namedtuples

```
from collections import namedtuple
Eatery = namedtuple('Eatery', ['name', 'location', 'park_id',
   ...: 'type_name'])
eateries = []
for eatery in nyc_eateries:
    details = Eatery(eatery['name'],
                     eatery['location'],
                     eatery['park_id'],
                     eatery['type_name'])
    eateries.append(details)
```

Leveraging namedtuples
each field is available as an attribute of the namedtuple

an attribute is basically a named field or data storage location
can depend on every instance of a namedtuple to have all the fields (some may be empty or None)
what this means?
we can always have safe access to a field without the need for a get method like a dictionary

```
for eatery in eateries[:3]:
    print(eatery.name)
    print(eatery.park_id)
    print(eatery.location)
```

for the first three entries

example
# Import namedtuple from collections
from collections import namedtuple

# Create the namedtuple: SpeciesDetails
SpeciesDetails = namedtuple('SpeciesDetails', ['species', 'sex', 'body_mass'])

# Create the empty list: labeled_entries
labeled_entries = []

# Iterate over the weight_log entries
for species, sex, body_mass in weight_log:
    # Append a new SpeciesDetails namedtuple instance for each entry to labeled_entries
    labeled_entries.append(SpeciesDetails(species, sex, body_mass))

print(labeled_entries[:5])

example
# Iterate over the first twenty entries in labeled_entries
for entry in labeled_entries[:20]:
    # if the entry's species is Chinstrap
    if entry.species == 'Chinstrap':
        # Print each entry's sex and body_mass separated by a colon
        print(f'{entry.sex}:{entry.body_mass}')

Dataclasses
can think of as more powerful namedtuple

can set default values for particular fields to ensure that each time you use a dataclass those fields are preset
custom representations of the objects
easy tuple or dictionary conversion
custom properties that do more than just store a value
frozen instances do not allow any edits to the properties after the dataclass has been created

@dataclass
need to create a decorator for the class you are about to make
a decorator is a wrapper around some code that adds extra behaviors

```
from dataclasses import dataclass
```

```
@dataclass
class Cookie:
    name: str
    quantity: int = 0
```

```
chocolate_chip = Cookie("chocolate chip", 13)
print(chocolate_chip.name)
print(chocolate_chip.quantity)
```

```
chocolate chip
13
```

define class name and field names with their types and default values

Easy tuple or a dictionary conversion

```
from dataclasses import asdict, astuple

ginger_molasses = Cookie("ginger molasses", 8)
asdict(ginger_molasses)
```

```
{'name': 'ginger molasses', 'quantity': 8}
```

```
astuple(ginger_molasses)
```

```
('ginger molasses', 8)
```

Custom properties

```
from decimal import Decimal



@dataclass
class Cookie:
    name: str
    cost: Decimal
    quantity: int
```

```
    @property
    def value_of_goods(self):
        return int(self.quantity) * self.cost
```

Frozen instances

```
@dataclass(frozen=True)
class Cookie:
    name: str
    quantity: int = 0

c = Cookie("chocolate chip", 10)
```

example
# Import dataclass
from dataclasses import dataclass

@dataclass
class WeightEntry:

```
    # Define the fields on the class
    species: str
    sex: int
    body_mass: int
    flipper_length: str

    # Define a property that returns the body_mass / flipper_length
    @property
    def mass_to_flipper_length_ratio(body_mass):
        return int.body_mass / int.flipper_length

example
# Create the empty list: labeled_entries
labeled_entries = []

# Iterate over the weight_log entries
for species, flipper_length, body_mass, sex in weight_log:
    # Append a new WeightEntry instance to labeled_entries
    labeled_entries.append(WeightEntry(species, flipper_length, body_mass, sex))

# Print a list of the first 5 mass_to_flipper_length_ratio values
print([entry.mass_to_flipper_length_ratio for entry in labeled_entries[:5]])
```