Getting Started
Python provides a variety of useful built-in data structures, such as lists, sets, and dictionaries. For the most part, the use of these structures is straightforward. However, common questions concerning searching, sorting, ordering, and filtering often arise. Thus, the goal of these scenarios is to discuss common data structures and algorithms involving data. In addition, treatment is given to the various data structures contained in the collections module.

Unpacking a Sequence into Separate Variables
Problem
You have an N-element tuple or sequence that you would like to unpack into a collection of N variables.

Solution
Any sequence (or iterable) can be unpacked into variables using a simple assignment operation. The only requirement is that the number of variables and structure match the sequence. For example:

```
p = (4, 5)
x, y = p
x
y
data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
name, shares, price, date = data
name
date
name, shares, price, (year, mon, day) = data

name
year
mon
day
```

If there is a mismatch in the number of elements, you'll get an error. For example:

```
p = (4, 5)
x, y, z = p
```

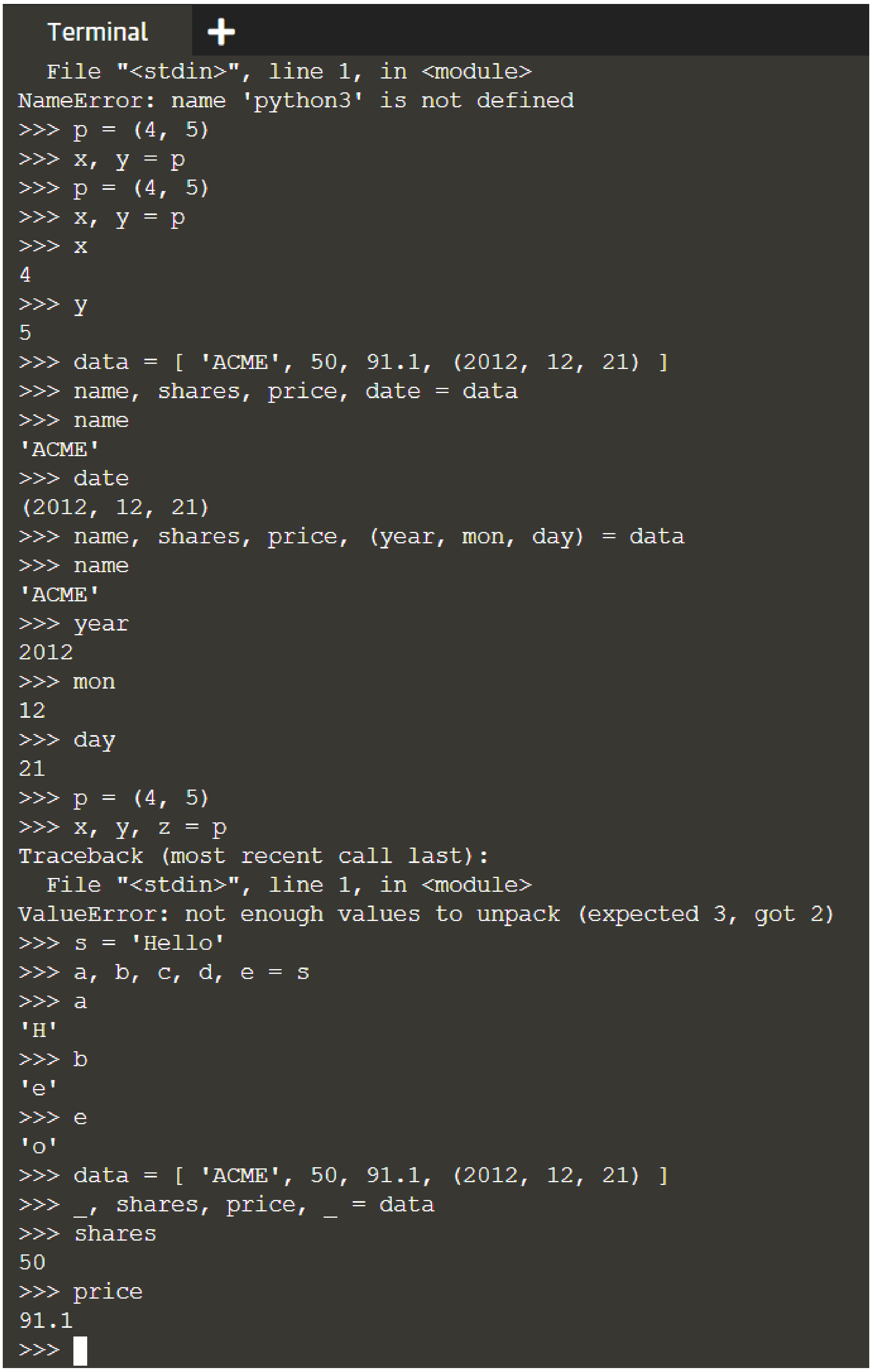
Discussion
Unpacking actually works with any object that happens to be iterable, not just tuples or lists. This includes strings, files, iterators, and generators. For example:

```
s = 'Hello'
a, b, c, d, e = s
a
b
e
```

When unpacking, you may sometimes want to discard certain values. Python has no special syntax for this, but you can often just pick a throwaway variable name for it. For example:

```
data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
_, shares, price, _ = data
shares
price
```

However, make sure that the variable name you pick isn't being used for something else already.



```
  File "<stdin>", line 1, in <module>
NameError: name 'python3' is not defined
>>> p = (4, 5)
>>> x, y = p
>>> p = (4, 5)
>>> x, y = p
>>> x
4
>>> y
5
>>> data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
>>> name, shares, price, date = data
>>> name
'ACME'
>>> date
(2012, 12, 21)
>>> name, shares, price, (year, mon, day) = data
>>> name
'ACME'
>>> year
2012
>>> mon
12
>>> day
21
>>> p = (4, 5)
>>> x, y, z = p
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not enough values to unpack (expected 3, got 2)
>>> s = 'Hello'
>>> a, b, c, d, e = s
>>> a
'H'
>>> b
'e'
>>> e
'o'
>>> data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
>>> _, shares, price, _ = data
>>> shares
50
>>> price
91.1
>>> 
```

Unpacking Elements from Iterables of Arbitrary Length
Problem

You need to unpack N elements from an iterable, but the iterable may be longer than N elements, causing a "too many values to unpack" exception.

Solution
Python "star expressions" can be used to address this problem. For example, suppose you run a course and decide at the end of the semester that you’re going to drop the first and last homework grades, and only average the rest of them. If there are only four assignments, maybe you simply unpack all four, but what if there are 24? A star expression makes it easy:

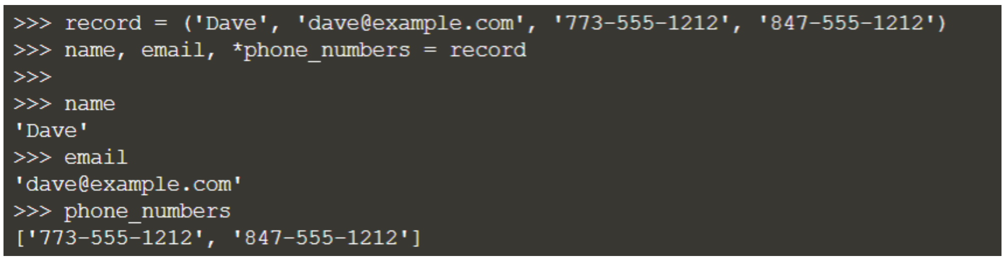
Unpacking Elements from Iterables of Arbitrary Length
Problem
You need to unpack N elements from an iterable, but the iterable may be longer than N elements, causing a "too many values to unpack" exception.

Solution
Python "star expressions" can be used to address this problem. For example, suppose you run a course and decide at the end of the semester that you’re going to drop the first and last homework grades, and only average the rest of them. If there are only four assignments, maybe you simply unpack all four, but what if there are 24? A star expression makes it easy:

```
def drop_first_last(grades):
    first, *middle, last = grades
    return avg(middle)
```
As another use case, suppose you have user records that consist of a name and email address, followed by an arbitrary number of phone numbers. You could unpack the records like this:

```
record = ('Dave', 'dave@example.com', '773-555-1212', '847-555-1212')
name, email, *phone_numbers = record

name
email
phone_numbers
```

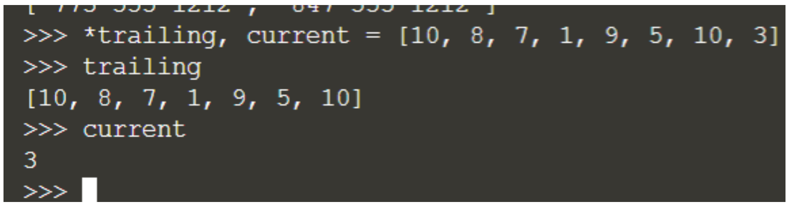

```
>>> record = ('Dave', 'dave@example.com', '773-555-1212', '847-555-1212')
>>> name, email, *phone_numbers = record
>>>
>>> name
'Dave'
>>> email
'dave@example.com'
>>> phone_numbers
['773-555-1212', '847-555-1212']
```

It’s worth noting that the phone_numbers variable will always be a list, regardless of how many phone numbers are unpacked (including none). Thus, any code that uses phone_numbers won’t have to account for the possibility that it might not be a list or perform any kind of additional type checking.

The starred variable can also be the first one in the list. For example, say you have a sequence of values representing your company’s sales figures for the last eight quarters. If you want to see how the most recent quarter stacks up to the average of the first seven, you could do something like this:

```
*trailing_qtrs, current_qtr = sales_record
trailing_avg = sum(trailing_qtrs) / len(trailing_qtrs)
return avg_comparison(trailing_avg, current_qtr)
```
Here’s a view of the operation from the Python interpreter:

```
*trailing, current = [10, 8, 7, 1, 9, 5, 10, 3]
trailing
current
```



```
>>> *trailing, current = [10, 8, 7, 1, 9, 5, 10, 3]
>>> trailing
[10, 8, 7, 1, 9, 5, 10]
>>> current
3
>>> 
```

Discussion
Extended iterable unpacking is tailor-made for unpacking iterables of unknown or arbitrary length. Oftentimes, these iterables have some known component or pattern in their construction (e.g. "everything after element 1 is a phone number"), and star unpacking lets the developer leverage those patterns easily instead of performing acrobatics to get at the relevant elements in the iterable.

It is worth noting that the star syntax can be especially useful when iterating over a sequence of tuples of varying length. For example, perhaps a sequence of tagged tuples:

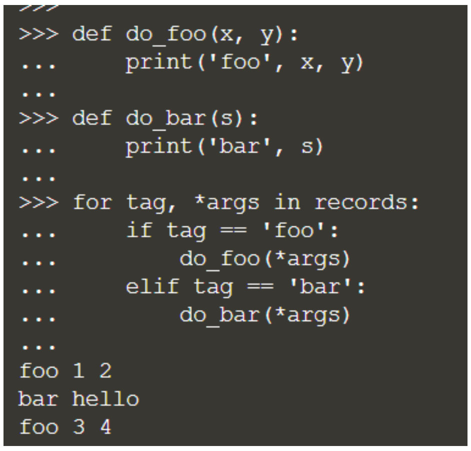
```
# hit 'return' in terminal to execute the following

records = [
    ('foo', 1, 2),
    ('bar', 'hello'),
    ('foo', 3, 4),
]

def do_foo(x, y):
   print('foo', x, y)

def do_bar(s):
   print('bar', s)

for tag, *args in records:
   if tag == 'foo':
      do_foo(*args)
   elif tag == 'bar':
      do_bar(*args)
```

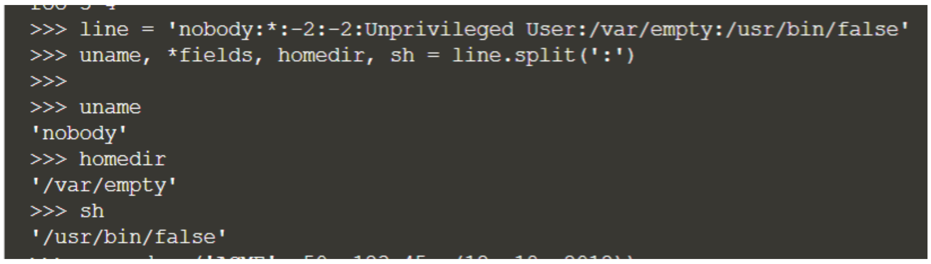


```
>>> def do_foo(x, y):
...     print('foo', x, y)
...
>>> def do_bar(s):
...     print('bar', s)
...
>>> for tag, *args in records:
...     if tag == 'foo':
...         do_foo(*args)
...     elif tag == 'bar':
...         do_bar(*args)
...
foo 1 2
bar hello
foo 3 4
```

Star unpacking can also be useful when combined with certain kinds of string processing operations, such as splitting. For example:

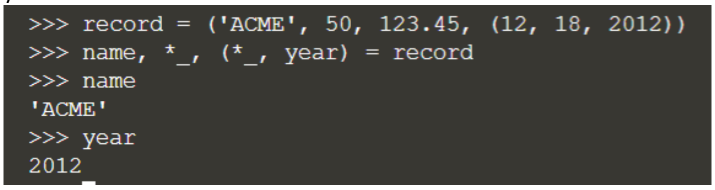
```
line = 'nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false'
uname, *fields, homedir, sh = line.split(':')

uname
homedir
sh
```



```
>>> line = 'nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false'
>>> uname, *fields, homedir, sh = line.split(':')
>>>
>>> uname
'nobody'
>>> homedir
'/var/empty'
>>> sh
'/usr/bin/false'
```

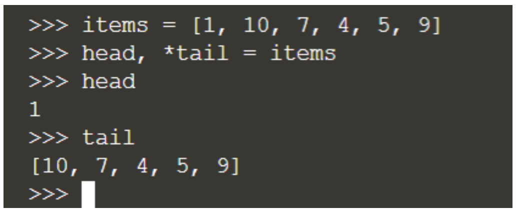
Sometimes you might want to unpack values and throw them away. You can't just specify a bare * when unpacking, but you could use a common throwaway variable name, such as _ or ign (ignored). For example:
record = ('ACME', 50, 123.45, (12, 18, 2012))
name, *_, (*_, year) = record
name
year



```
>>> record = ('ACME', 50, 123.45, (12, 18, 2012))
>>> name, *_, (*_, year) = record
>>> name
'ACME'
>>> year
2012
```

There is a certain similarity between star unpacking and list-processing features of various functional languages. For example, if you have a list, you can easily split it into head and tail components like this:

items = [1, 10, 7, 4, 5, 9]
head, *tail = items
head
tail



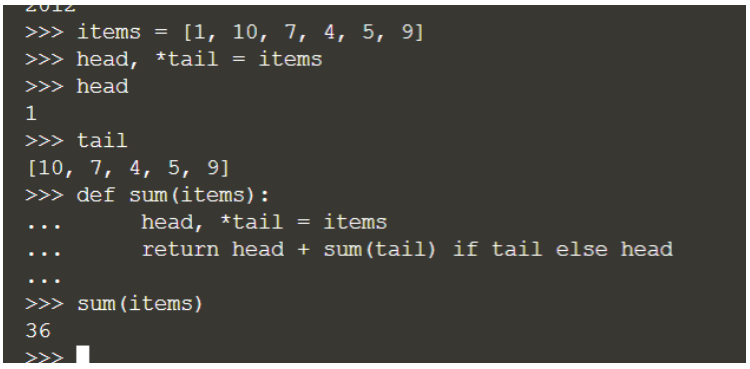
```
>>> items = [1, 10, 7, 4, 5, 9]
>>> head, *tail = items
>>> head
1
>>> tail
[10, 7, 4, 5, 9]
>>>
```

One could imagine writing functions that perform such splitting in order to carry out some kind of clever recursive algorithm. For example:

```
def sum(items):
    head, *tail = items
    return head + sum(tail) if tail else head

sum(items)
```

However, be aware that recursion really isn't a strong Python feature due to the inherent recursion limit. Thus, this last example might be nothing more than an academic curiosity in practice.



```
>>> items = [1, 10, 7, 4, 5, 9]
>>> head, *tail = items
>>> head
1
>>> tail
[10, 7, 4, 5, 9]
>>> def sum(items):
...     head, *tail = items
...     return head + sum(tail) if tail else head
...
>>> sum(items)
36
>>>
```