When you run the file hello\_world.py, the ending .py indicates that the file is a Python program. Your editor then runs the file through the Python interpreter, which reads through the program and determines what each word in the program means. For example, when the interpreter sees the word print, it prints to the screen whatever is inside the parentheses.

When an error occurs in your program, the Python interpreter does its best to help you figure out where the problem is. The interpreter provides a traceback when a program cannot run successfully. A traceback is a record of where the interpreter ran into trouble when trying to execute your code.

A name error usually means we either forgot to set a variable’s value before using it, or we made a spelling mistake when entering the variable’s name.

Input () is used to get values from the console

name = "tinshu "

print(name.rstrip().\_\_len\_\_())

print(name.\_\_len\_\_())

# to remove it permanently , I think immutability comes here

name= name.rstrip();

print(name.\_\_len\_\_())

But be aware that you can sometimes get an arbitrary number of decimal

places in your answer:

>>> **0.2 + 0.1**

0.30000000000000004

>>> **3 \* 0.1**

0.30000000000000004

This happens in all languages and is of little concern. Python tries to find a way to represent the result as precisely as possible, which is sometimes difficult given how computers have to represent numbers internally.

***Avoiding Type Errors with the str() Function***

Often, you’ll want to use a variable’s value within a message. For example, say you want to wish someone a happy birthday. You might write code like this:

*birthday.py* age = 23

message = "Happy " + age + "rd Birthday!"

print(message)

You might expect this code to print the simple birthday greeting, Happy 23rd birthday! But if you run this code, you’ll see that it generates an error:

Traceback (most recent call last):

File "birthday.py", line 2, in <module>

message = "Happy " + age + "rd Birthday!"

u TypeError: Can't convert 'int' object to str implicitly

This is a *type error*. It means Python can’t recognize the kind of information you’re using. In this example Python sees at u that you’re using a variable that has an integer value (int), but it’s not sure how to interpret that value. Python knows that the variable could represent either the numerical value 23 or the characters *2* and *3*. When you use integers within strings like this, you need to specify explicitly that you want Python to use the integer as a string of characters. You can do this by wrapping the variable in the str() function, which tells Python to represent non-string values as strings:

age = 23

message = "Happy " + str(age) + "rd Birthday!"

print(message)

age = 20;  
msg = 'happy ' + str(age) + ' Birthday'  
print(msg)

***Integers in Python 2***

Python 2 returns a slightly different result when you divide two integers:

>>> **python2.7**

>>> **3 / 2**

1

Instead of 1.5, Python returns 1. Division of integers in Python 2 results in an integer with the remainder truncated. Note that the result is not a rounded integer; the remainder is simply omitted. To avoid this behavior in Python 2, make sure that at least one of the numbers is a float. By doing so, the result will be a float as well:

>>> **3 / 2**

1

>>> **3.0 / 2**

1.5

>>> **3 / 2.0**

1.5

>>> **3.0 / 2.0**

1.5

data = 3/2  
print(data)\_

# ouput is 1.5

What Is a List?

A list is a collection of items in a particular order

firstList = [1, 2, 3, 4, 5, 6]  
print(firstList)  
print(firstList[0])  
# gives you the last element in the array  
print(firstList[-1])

Python has a special syntax for accessing the last element in a list. By asking for the item at index -1, Python always returns the last item in the list:

This code returns the value 'specialized'. This syntax is quite useful, because you’ll often want to access the last items in a list without knowing exactly how long the list is. This convention extends to other negative index values as well. The index -2 returns the second item from the end of the list, the index -3 returns the third item from the end, and so forth.

**Changing, Adding, and Removing Elements**

Most lists you create will be dynamic, meaning you’ll build a list and

then add and remove elements from it as your program runs its course

Inserting Elements into a List

You can add a new element at any position in your list by using the insert() method

**Removing an Item Using the del Statement**

If you know the position of the item you want to remove from a list, you can use the del statement

print(firstList)  
firstList.append(300)  
print(firstList)  
firstList.insert(1,500)  
print(firstList)  
del firstList[1]  
print(firstList)

**Removing an Item Using the pop() Method**

Sometimes you’ll want to use the value of an item after you remove it from a list

ele = firstList.pop()  
print(ele)  
print(firstList)

The pop() method removes the last item in a list, but it lets you work

with that item after removing it

**Popping Items from any Position in a List**

You can actually use pop() to remove an item in a list at any position by

including the index of the item you want to remove in parentheses

ele1 = firstList.pop(0)  
print(ele1)  
print(firstList)

**Sorting a array**

print(firstList)  
firstList.sort()  
firstList.sort(reverse=True)  
print(firstList)  
print(sorted(firstList))

The reverse() method changes the order of a list permanently, but you

can revert to the original order anytime by applying reverse() to the same list a second time

Slicing a List

To make a slice, you specify the index of the first and last elements you

want to work with. As with the range() function, Python stops one item

before the second index you specify.

players = ['charles', 'martina', 'michael', 'florence', 'eli']

print(players[1:4])

This time the slice starts with 'martina' and ends with 'florence':

['martina', 'michael', 'florence']

If you omit the first index in a slice, Python automatically starts your

slice at the beginning of the list:

players = ['charles', 'martina', 'michael', 'florence', 'eli']

print(players[:4])

A similar syntax works if you want a slice that includes the end of a list.

For example, if you want all items from the third item through the last item,

you can start with index 2 and omit the second index:

players = ['charles', 'martina', 'michael', 'florence', 'eli']

print(players[2:])

Python returns all items from the third item through the end of the list:

['michael', 'florence', 'eli']

***Copying a List***

To copy a list, you can make a slice that includes the entire original list

by omitting the first index and the second index ([:]). This tells Python to make a slice that starts at the first item and ends with the last item, producing a copy of the entire list.

Don’t worry about the details in this example for now. Basically, if you’re trying to work with a copy of a list and you see unexpected behavior, make sure you are copying the list using a slice

**Tuples**

sometimes you’ll want to create a list of items that cannot change. Tuples allow you to do just that. Python refers to values that cannot change as immutable, and an immutable list is called a tuple

A tuple looks just like a list except you use parentheses instead of square brackets. Once you define a tuple, you can access individual elements by using each item’s index, just as you would for a list

dimensions = (200, 50)

>>> age\_0 = 22

>>> age\_1 = 18

>>> age\_0 >= 21 or age\_1 >= 21

True

>>> age\_0 = 18

>>> age\_0 >= 21 or age\_1 >= 21

False

>>> age\_0 = 22

>>> age\_1 = 18

>>> age\_0 >= 21 and age\_1 >= 21

False

***Checking Whether a Value Is in a List***

***To find out whether a particular value is already in a list, use the keyword in***

>>> requested\_toppings = ['mushrooms', 'onions', 'pineapple']

>>> 'mushrooms' in requested\_toppings

True

>>> 'pepperoni' in requested\_toppings

False

Checking Whether a Value Is Not in a List

Other times, it’s important to know if a value does not appear in a list.

if user not in banned\_users:

Checking That a List Is Not Empty

requested\_toppings = []

if requested\_toppings:

A Simple Dictionary

Consider a game featuring aliens that can have different colors and point values.

alien\_0 = {'color': 'green', 'points': 5}

**Working with Dictionaries**

A dictionary in Python is a collection of key-value pairs. Each key is connected to a value, and you can use a key to access the value associated with that key. A key’s value can be a number, a string, a list, or even another dictionary.In fact, you can use any object that you can create in Python as a value in a dictionary.

**Adding New Key-Value Pairs**

Dictionaries are dynamic structures, and you can add new key-value pairs to a dictionary at any time

***Removing Key-Value Pairs***

When you no longer need a piece of information that’s stored in a dictionary, you can use the del statement to completely remove a key-value pair. All del needs is the name of the dictionary and the key that you want to remove.

for key, value in user\_0.items():

print("\nKey: " + key)

print("Value: " + value)

Nesting

Sometimes you’ll want to store a set of dictionaries in a list or a list of

items as a value in a dictionary. This is called nesting.

**The while Loop in Action**

You can use a while loop to count up through a series of numbers. For

example, the following while loop counts from 1 to 5:

counting.py

current\_number = 1

while current\_number <= 5:

print(current\_number)

current\_number += 1

Using break to Exit a Loop

To exit a while loop immediately without running any remaining code in the loop, regardless of the results of any conditional test, use the break statement.

Removing All Instances of Specific Values from a List

In Chapter 3 we used remove() to remove a specific value from a list. The remove() function worked because the value we were interested in appeared only once in the list. But what if you want to remove all instances of a value from a list?

pets = ['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']

print(pets)

while 'cat' in pets:

pets.remove('cat')

print(pets)

**Defining a Function**

def print\_name(name):

print(name)

print\_name('nas')

Any indented lines that follow def greet\_user(): make up the body of

the function. The text at is a comment called a docstring, which describes what the function does. Docstrings are enclosed in triple quotes, which Python looks for when it generates documentation for the functions in your programs.

Making an Argument Optional

Sometimes it makes sense to make an argument optional so that people using the function can choose to provide extra information only if they want to. You can use default values to make an argument optional.

Pass the list to function and the passed list is copied

print\_models(unprinted\_designs[:], completed\_models)

**Passing an Arbitrary Number of Arguments**

Sometimes you won’t know ahead of time how many arguments a function needs to accept. Fortunately, Python allows a function to collect an arbitrary number of arguments from the calling statement.

For example, consider a function that builds a pizza. It needs to accept a

number of toppings, but you can’t know ahead of time how many toppings a person will want. The function in the following example has one parameter, \*toppings, but this parameter collects as many arguments as the calling

line provides:

pizza.py def make\_pizza(\*toppings):

"""Print the list of toppings that have been requested."""

print(toppings)

Mixing Positional and Arbitrary Arguments

If you want a function to accept several different kinds of arguments, the parameter that accepts an arbitrary number of arguments must be placed last in the function definition. Python matches positional and keyword arguments first and then collects any remaining arguments in the final parameter

The asterisk in the parameter name \*toppings tells Python to make an

empty tuple called toppings and pack whatever values it receives into this tuple.

Sometimes you’ll want to accept an arbitrary number of arguments, but you won’t know ahead of time what kind of information will be passed to the function. In this case, you can write functions that accept as many key-value pairs as the calling statement provides. One example involves building user profiles: you know you’ll get information about a user, but you’re not sure what kind of information you’ll receive

def build\_profile(first, last, **\*\*user\_info):**

"""Build a dictionary containing everything we know about a user."""

profile = {}

u profile['first\_name'] = first

profile['last\_name'] = last

v for key, value in **user\_info.items():**

profile[key] = value

return profile

Inheritance

You don’t always have to start from scratch when writing a class. If the class you’re writing is a specialized version of another class you wrote, you can use inheritance. When one class inherits from another, it automatically takes on all the attributes and methods of the first class. The original class is called the parent class, and the new class is the child class.

The keyword with closes the file once access to it is no longer needed.

Notice how we call open() in this program but not close()

**Writing to an Empty File**

To write text to a file, you need to call open() with a second argument telling Python that you want to write to the file. To see how this works, let’s write a simple message and store it in a file instead of printing it to the screen:

write\_ filename = 'programming.txt'

message.py

with open(filename, 'w') as file\_object:

file\_object.write("I love programming.")

In the previous example, we informed our users that one of the files

was unavailable. But you don’t need to report every exception you catch. Sometimes you’ll want the program to fail silently when an exception occurs and continue on as if nothing happened. To make a program fail silently, you write a try block as usual, but you explicitly tell Python to do nothing in the except block. Python has a pass statement that tells it to do nothing in a block: