layout: post
title: Computing Systems for Machine Learning
date: 2022-01-02 12:18 +0800
tags: [Programming]

toc: true

Chapter 1 Introduction

skipped :3

Chapter 2 Python Fundamentals

Syntax

hello. py

Python does not have a main method like Java

-The program's main code is just written directly in the file

Python statements do not end with semicolons

```
print("Hello, world!")
```

Multi-Line Statements

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For example:

```
total = item_one + \
        item_two + \
        item_three
```

Statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example:

```
days = ['Monday', 'Tuesday', 'Wednesday',
                'Thursday', 'Friday']
```

Quotation in Python

Python accepts single ('), double (") and triple ('' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string

The triple quotes can be used to span the string across multiple lines. For example, all the following are legal:

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is made up
  of multiple lines and sentences."""
```

Comments

As programs get bigger and more complicated, they become more difficult to read

It is a good idea to add notes to your programs

A # symbol that is not inside a string starts a comment

All characters after the # and up to the physical line end are part of the comment

Python interpreter ignores them

Using Blank Lines

A line containing only whitespace, possibly with a comment, is known as a blank line, and is ignored by the Python interpreter

In an interactive interpreter session, you must enter an empty physical line to terminate a multiline statement

Multiple Statements on a Single Line

The semicolon (;) allows multiple statements on the single line given that neither statement starts a new code block. Here is a sample snip using the semicolon:

```
import sys; x = 'foo'
```

Multiple Statement Groups as Suites

Groups of individual statements making up a single code block are called suites in Python

Compound or complex statements, such as if, while, def, and class, are those which require a header line and a suite

Header lines begin the statement (with the keyword) and terminate with a colon (:) and are followed by one or more lines which make up the suite

```
if expression1 :
suite1
elif expression2 :
  suite2
else :
  suite3
```

Suites and Indentation

One of the first caveats programmers encounter when learning Python is that there are no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount

```
if x == True:
  print ("Answer")
  print ("True")
else:
  print ("Answer")
  print ("False")
```

Reserved Words

Keywords contain lowercase letters only

```
and exec not
assert finally or
break for pass
class from print
continue global raise
def if return
del import try
elif in while
else is with
except lambda yield
```

Variables and Expressions

Variable Name

Variable names can be both letters and numbers

They have to begin with a letter though

Uppercase letters are allowed, generally we use lowercase

Underscore character (_) can also appear in a name

Variable name cannot be a keyword

An illegal name results in a syntax error

Assigning Values to Variables

Python variables do not have to be explicitly declared to reserve memory space

The declaration happens automatically when you assign a value to a variable.

The equal sign (=) is used to assign values to variables

```
counter = 100  # An int assignment
miles = 1000.0 # A float
```

```
name = "John"  # A str
print (counter)
print (miles)
print (name)
```

Constants

Python doesn't really have constants

Instead, declare a "global" variable at the top of your code

All methods will be able to use this value

```
MAX_VALUE = 3
```

Multiple Assignments

You can also assign a single value to several variables simultaneously

```
a = b = c = 1
a, b, c = 1, 2, "john"
```

Data Types

Number types

• int

• float

• complex

Sequence type

• str (immutable)

• list

• tuple (immutable)

• range

• zip (sequence of tuples)

Boolean type

• bool

Set type

• set

• frozenset

Mapping type

• dict

Binary type

• bytes

• bytearray

• memoryview

Data Types: Java vs Python

Python is looser about types than Java

Variables' types do not need to be declared

Variable can change type as a program is running

If the variable value is set to a value of a different type

```
a = 1       # integer type
a = "john"  # string type
```

Expressions and Statements

An expression is a combination of values, variables, and operators

A value or variable all by itself is also an expression

```
Examples
21
x
x+21
```

A statement is a unit of code that a Python interpreter can execute, for example,

• print

• assignment

Operators and Operands

Operators represent computations like addition, multiplication, division, etc.

+ - * / % **

The values the operator is applied to are called operands

```
>>> 1 + 1
2
>>> 1 + 3 * 4 - 2
11
>>> 7 / 2
3
>>> 7.0 / 2
3.5
>>> 10 ** 6
1000000
```

Modulus Operator

Works on integers
Yields the remainder of the first operand divided by the second

Indicated by %

• Quotient = 7 // 3

• Remainder = 7 % 3

Utility: find the last digits of a number

```
>>> 97856 % 100
56
```

Order of Operations

When multiple operators appear in an expression, the order of evaluation depends on the rules of precedence

Multiplication and division have the same precedence; addition and subtraction have the same precedence

Operators with the same precedence are evaluated from left to right (except exponentiation which has right-to-left precedence)

Operators: Java vs Python

No ++ or -- operators (must manually adjust by 1)

Java

```
nt x = 2;
x++;
System.out.println(x);
x = x * 8;
System.out.println(x);
double d = 3.2;
d = d / 2;
System.out.println(d);
```

Python

```
x = 2
x = x + 1
print(x)
x = x * 8
print(x)
d = 3.2
d = d / 2
print(d)
```

str Data Type

Strings in Python are identified as a contiguous set of characters in between quotation marks

Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end

The plus (+) sign is the string concatenation operator, and the asterisk (*) is the repetition operator

```
str1 = 'Hello World!'
print str1 # Prints complete string
print str1[0] # Prints first character of the string
print str1[2:5] # Prints characters starting from 3rd to 6th
print str1[2:] # Prints string starting from 3rd character
print str1 * 2 # Prints string two times
print str1 + "TEST" # Prints concatenated string

Output:
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

String Operations

You can’t perform mathematical operations on strings

Examples

• ‘2’ - ’1’

• ‘eggs’ / ‘dozens’

The + operator works with strings and performs concatenations

```
first = 'hello'
second = 'class'
print(first + second)

Output:
helloclass
```

String Multiplication

The * operator works with strings and performs repetition

Examples

• ‘Spam’*3 is ‘SpamSpamSpam’

If one of the operands is a string, the other has to be an integer

```
>>> "hello" * 3
"hellohellohello"
>>> print(10 * "yo ")
yo yo yo yo yo yo yo yo yo yo
>>> print(2 * 3 * "4")
444444
```

String Concatenation

Integers and strings cannot be concatenated in Python

str(value) - converts a value into a string

print(expr, expr) - prints two items on the same line

```
>>> x = 4
>>> print("Thou shalt not count to " + x + ".")
TypeError: cannot concatenate 'str' and 'int' objects
>>> print("Thou shalt not count to " + str(x) + ".")
Thou shalt not count to 4.
>>> print(x + 1, "is out of the question.")
5 is out of the question.
```

list Data Type

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([])

To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type

The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end-1

The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']

print list          # Prints complete list
print list[0]       # Prints first element of the list
print list[1:3]     # Prints elements starting from 2nd till 3rd
print list[2:]      # Prints elements starting from 3rd element
print tinylist * 2  # Prints list two times
print list + tinylist # Prints concatenated lists
Output:
```

```
['abcd', 786, 2.23, 'john', 70.2]
abcd
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
```

tuple Data Type

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses

The main differences between lists and tuples are: Lists are enclosed in brackets ([ ]), and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as read-only lists

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')

print(tuple)        # Prints complete list
print(tuple[0])        # Prints first element of the list
print(tuple[1:3])     # Prints elements starting from 2nd till 3rd
print tuple[2:]       # Prints elements starting from 3rd element
print tinytuple * 2   # Prints list two times
print tuple + tinytuple # Prints concatenated lists


OUTPUT:
('abcd', 786, 2.23, 'john', 70.2)
abcd
(786, 2.23)
(2.23, 'john', 70.2)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.2, 123, 'john')
```

dict Data Type

Python 's dictionaries are hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs

Keys can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([])

```
dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two“
tinydict = {'name': 'john','code':6734, 'dept': 'sales'}
print(dict['one'])      # Prints value for 'one' key
print (dict[2])          # Prints value for 2 key
print (tinydict)         # Prints complete dictionary
print (tinydict.keys())  # Prints all the keys
print (tinydict.values()) # Prints all the values

OUTPUT:
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'}
['dept', 'code', 'name']
['sales', 6734, 'john']
```

Data Type Conversion

```
Function Description
int(x [,base]) Converts x to an integer. base specifies the base if x is a string
long(x [,base] ) Converts x to a long integer. base specifies the base if x is a string
float(x) Converts x to a floating-point number
complex(real [,imag]) Creates a complex number
str(x) Converts object x to a string representation
repr(x) Converts object x to an expression string
eval(str) Evaluates a string and returns an object
tuple(s) Converts s to a tuple
list(s) Converts s to a list
set(s) Converts s to a set
dict(d) Creates a dictionary. d must be a sequence of (key,value) tuples
frozenset(s) Converts s to a frozen set
chr(x) Converts an integer to a character
unichr(x) Converts an integer to a Unicode character
ord(x) Converts a single character to its integer value
hex(x) Converts an integer to a hexadecimal string
oct(x) Converts an integer to an octal string
```

Conditionals

Boolean Expressions

A boolean expression is an expression that is either true or false

It uses the operator '==', which compares the operands to the left and the right of this operator

It produces either True or False

• for example, 5==5 will return True

• and 5==6 will return False

True and False are of type bool

• x != y # x is not equal to y

• x > y # x is greater than y

• x < y # x is less than y

• x >= y # x is greater than or equal to y

• X <= y # x is less than or equal to y

Logical Operators

Three logical operators: and, or, and not. For example:

• x > 0 and x < 10

• This is true only if x is greater than 0 and less than 10

• n%2 == 0 or n%3 == 0 is true if either of the conditions is true, that is, if the number is divisible by 2 or 3

• The not operator negates a boolean expression, so not(x > y) is true if x > y is false, that is, if x is less than or equal to y

Conditional Execution

We need to check conditions and change the behavior of the program

The simplest conditional statement is if, for example:

The Boolean expression after 'if' is called the condition

```
if x>0:
  print('x is positive')

OUTPUT:
x is positive
```

if Statement

if statements have a header followed by an indented body

Statements like these are called compound statements

There is no limit on the number of statements that can appear in the body, but there must be at least one

Sometimes it is useful to have a body with no statements, usually as a place holder

In that case use the pass statement, which does nothing

```
if x < 0: pass  # need to handle negative values
```

Alternative Execution

When there are two possibilities and the condition determines which one gets executed

The alternatives are called branches, because they are branches in the flow of execution

```
if x%2 == 0:
  print('x is even')
else:
  print('x is odd')
```

Chained Conditionals

Sometimes there are more than two possibilities and we need more than two branches

One way to express a computation like that is a chained conditional:
elif is an abbreviation of "else if"

```
if(x <y):
  print('x is less than y')
elif(x>y):
  print('x is greater than y')
else:
  print('x and y are equal')
```

There is no limit to the number of elif statements

If there is an else clause, it must be at the end

Each condition is checked in order. If the first one is false, the next is checked, and so on

We don’t need an else statement:

```
if choice == 'a':
  draw_a()
elif choice == 'b':
  draw_b()
elif choice == 'c':
  draw_c()
```

Functions

Function

A function is some reusable code that

• takes arguments(s) as input

• does some computation and

• returns a result or results

There are two kinds of functions in Python

• Built-in functions that are provided as part of Python - raw_input(), type(), float(), int() …

• Functions that we define ourselves and then use

We treat the built-in function names as "new" reserved words (i.e., we avoid them as variable names)

A Built-in Function: max()

A built-in function is some stored code that we use. It takes some input and produces an output.

```
>>> big = max('Hello world')
>>> print big
'w'
```

Type Conversion Functions

There are built-in functions in Python that convert from one data type to another

• The int() function takes any value and converts it into an integer, if it can

• int() can convert floating-point values to integers, but it doesn't round off, and instead chops off the fraction part

• float() converts integers and strings to floating-point numbers

• str() converts its arguments to string

Math Functions

Python has a math module that provides mathematical functions

Before using the module, it needs to be imported `import math`

This module contains the functions and variables defined in the module

To access one of the functions, you have to specify the name of the module and the name of the function

Import

Python provides 2 ways to import modules

• We have already seen import math

• import statement imports all the functions from a module into the code

We can also import functions using from

• Imports only the specified function

• Syntax

from [module] import [function or value]

from random import choice

Counting and Iterating Functions

len: returns the number of items of an enumerable object

```
>>> len( ['c', 'm', 's', 'c', 3, 2, 0] )
7
```

range: returns an iterable object

```
>>> list( range(10) )
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

enumerate: returns iterable tuple (index, element) of a list

```
>>> enumerate( [“311”, “320”, “330”] )
[(0, “311”), (1, “320”), (2, “330”)]
```

map: apply a function to a sequence or iterable

```
>>>arr = [1, 2, 3, 4, 5]
>>>map(lambda x: x**2, arr)
[1, 4, 9, 16, 25]
```

```
>>> map(lambda x, y: x + y, [1, 3, 5, 7, 9], [2, 4, 6, 8, 10])
[3, 7, 11, 15, 19]
```

filter: returns a list of elements for which a predicate is true

```
>>>arr = [1, 2, 3, 4, 5, 6, 7]
>>> filter(lambda x: x % 2 == 0, arr)
[2, 4, 6]
```

User-Defined Functions

A named sequence of statements that perform a certain computation

After defining a function, you can call it by name

• Example: type(34)

Why Define & Use Functions?

• A group of statements gets a name

• Modular code

• Easier debugging

• Code reuse

```
# Prints a helpful message.
def hello():
  print("Hello, world!")
# main (calls hello twice)

hello()
hello()
```

Must be declared above the 'main' code

Statements inside the function must be indented

Whitespace Significance

Python uses indentation to indicate blocks, instead of {}

• Makes the code simpler and more readable

• In Java, indenting is optional. In Python, you must indent

```
# Prints a helpful message.
def hello():
  print("Hello, world!")
# main (calls hello twice)

hello()
hello()
```

Arguments, Parameters, Variables

Most functions require arguments

• E.g.: math functions

An expression can also be used as an argument

Inside the function, arguments are assigned to variables called parameters

• Parameters are local to the function

Some functions return a value and others return nothing (void)

A variable created inside a function is local

```
def summation(x,y):
  z = x + y
  return(z)

print(summation(2,4))

m = summation(2,4)
print(m)
```

```
def decide(x,y):
  if x=='green' and y=='Red':
    print("You can go!")

y = 'Red'
y = 'green'
y = 'green'
z = 'Red'

decide(z,y)
```

Iterations

Iterations

Involves repetition

A statement or group of statements that need to be repeated

Help in automation of repeated tasks

The for Loop

Uses an iterator (name) and repeats for values 0 (inclusive) to max (exclusive)

```
for name in range(max):
    statements
```

```
>>> for i in range(5):
...     print(i)
0
1
2
3
4
```

for Loop Variations

Can specify a min other than 0, and a step other than 1

```
for name in range(min, max):
    statements

for name in range(min, max, step):
    statements
```

```
>>> for i in range(2, 6):
...     print(i)
2
3
4
5
>>> for i in range(15, 0, -5):
...     print(i)
15
10
5
```

Nested Loops

Nested loops are often replaced by string * and +

```
for line in range(1, 6):
  print((5 - line) * "." + str(line))
```

```
output:
....1
...2
..3
.4
5
```

```
def bar():
  print "#" + 16 * "=" + "#"
def top():
  for line in range(1, 5):        # split a long line by ending it with \
    print("|" + (-2 * line + 8) * " " + \
         "<>" + (4 * line - 4) * "." + "<>" + \
         (-2 * line + 8) * " " + "|")
def bottom():
  for line in range(4, 0, -1):
    print("|" + (-2 * line + 8) * " " + \
         "<>" + (4 * line - 4) * "." + "<>" + \
         (-2 * line + 8) * " " + "|")

# main
bar()
top()
bottom()
bar()
```

```
output:
#================#
|      <><>      |
|    <>....<>    |
|  <>........<>  |
|<>............<>|
|<>............<>|
|  <>........<>  |
|    <>....<>    |
|      <><>      |
#================#
```

Concatenating Ranges

Ranges can be concatenated with +

• Can be used to loop over a disjoint range of numbers

```
>>> range(1, 5) + range(10, 15)
[1, 2, 3, 4, 10, 11, 12, 13, 14]

>>> for i in range(4) + range(10, 7, -1):
...     print(i)
0
1
2
3
10
9
8
```

while Loop

In the definition there is no starting iterator

We need to use a condition with the while

It requires a condition, e.g.,

```
n = 12
while(n >0):
  print(n)
  n = n - 1
```

Execution Flow for a while Loop

At the start of each iteration, evaluate the condition, yielding True or False

If the condition is false, exit the while statement and continue execution at the next statement

If the condition is true, execute the body and then go to the condition again

The body of the loop will change the value of one or more variables so that condition eventually becomes false and the loop terminates

break Statement

Sometimes we want to break out of a loop if some

condition is satisfied

```
n = 12
while(n >0):
  print(n)
  n = n - 1
  if n == 7:
    break

output:
12
11
10
9
8
```

Chapter 3 Python Advanced

Strings

String

A sequence type

A sequence of zero or more characters

course = “MSML 605”

A string is delimited (begins and ends) by single or double quotes

The empty string has zero characters (" or "")

```
poem = 'Ode to a Nightingale'
lyric = "Roll on, Columbia, roll on"
exclamation = "That makes me !#? "
```

Quote Characters in Strings

 You can include a single quote in a double quoted string or a double quote in a single quoted string

```
will = "All the world's a stage"
ben = 'BF: "A penny saved is a penny earned"'
```

 To put a single quote in a single quoted string, precede it with the backslash ('\') or 'escape' character.

```
>>> will = 'All the world\'s a stage'
>>> print(will)
All the world's a stage
```

 The same goes for double quotes

```
>>> ben = "BF: \"A penny saved is a penny earned\""
>>> print(ben)
BF: "A penny saved is a penny earned"
```

Putting a Format Character in a String

 A format character is interpreted by the print() function to change the layout of text

 To put a format character in a string, precede it with the backslash ('\')

 A newline is represented by '\n'

```
>>> juliette = 'Good night, good night\nParting is such sweet sorrow'
>>> print(juliette)
Good night, good night
Parting is such sweet sorrow
```

 A tab is represented by '\t'

```
>>> tabs = 'col0\tcol1\tcol2'
>>> print(tabs)
col0 col1 col2
```

Bracket operator and Index

You can access the characters one at a time with the bracket operator

The first character has index 0

`second_character = course[1]`

```
>>> course = "MSML 605"
>>> second_character = course[1]
>>> print(second_character)
S
```

index has to be an integer

Negative Indices

Using negative indices: counts backwards from the string end

```
>>> course[-1]
5
```

len() Function

last index using length

```
>>> len(course)
8
```

Traversal with while

Processing one character at a time

```
course = "MSML 605"
index = 0
while index < len(course):
  letter = course[index]
  print(letter)
  index +=1

output:
M
S
M
L

6
```

```
0
5
```

Traversal with for

A more Pythonic way to traverse a string using for

```
course = "MSML 605"
for c in course:
  print(c)

output:
M
S
M
L

6
0
5
```

```
course = "MSML 605"
for c in course:
  print(c, end = ' ')

output:
M S M L   6 0 5
```

String Concatenation

Two strings can be concatenated using the '+' operator

```
word1 = 'abc'
word2 = 'def'
word = word1 + word2
print(word)

output:
abcdef
```

String Slices

A segment of a string is a slice

Selecting a slice is similar to selecting a character

The operator [n:m] returns the part of the string from the "n-eth" character to the "m-eth" character

It includes the first but excludes the last

```
>>> greeting = 'hello, world'
>>> greeting[1:3]
'el'
```

```
>>> greeting[-3:-1]
'rl'
```

If the first index before the colon is omitted, the slice starts at the beginning of the string.

If you omit the second index, the slice goes to the end of the string

```
>>> print(greeting[:4], greeting[7:])
hell world
```

If the first index >= second index, the result is an empty string

You can pick every kth element if you like

```
>>> greeting[3:10:2]
'l,wr'
```

Strings Are Immutable

Once created, a string cannot be modified

What happens if [ ] operators are used on the left side of the assignment operator?

```
>>> greeting = 'hello, world'
>>> greeting = 'J' + greeting[1:]
>>> greeting
'Jello, world'
```

String Search

find is the opposite of the [ ] operator

Instead of taking an index and extracting the corresponding character, it takes a character and finds an index

```
def find(word,letter):
  index = 0
  while index < len(word):
    if word[index] == letter:
      return(index)
    index +=1
  return(-1)
```

Looping and Counting

The following program counts the number of times the letter ‘a’ appears in a string

```
word = 'banana'
count = 0
for leter in word:
```

```
    if letter == 'a':
      count +=1
print(count)

output:
3
```

upper() Method

A method is a function that is bundled together with an object: it takes arguments and returns a value

A method call is called an invocation

We are invoking upper() on word

```
word = 'banana'
word = word.upper()
word

OUTPUT
'BANANA'
```

find() Method

There is a string method named ‘find()’

We invoke find() on word

```
word = 'banana'
index = word.find('a')
print (index)  # will print 1, the first instance of 'a'
```

find() can also find substrings not just characters

`word.find('na')`

It can take as a second argument the index where it

should start:

`word.find(‘na’,3)`

As a third argument the index where it should stop:

```
name = ‘bob’
name.find(‘b’,1,2)
```

in Operator

The word ‘in’ is a Boolean operator that takes two strings and returns true if the first appears as a substring in the second

`‘a’ in ‘banana’`

`‘seed’ in ‘banana’`

String Comparison

Relational operators work on strings

Equality operator ‘==’

Other relational operations are useful for putting words in alphabetical order:

```
if word < 'banana’:
   print('Your word, ' + word + ', comes before banana.')
elif word > 'banana’:
   print('Your word, ' + word + ', comes after banana.')
else:
   print('All right, bananas.')
```

String Operationss

• "hello"+"world" "helloworld" # concatenation

• "hello"*3 "hellohellohello" # repetition

• "hello"[0] "h" # indexing

• "hello"[-1] "o" # (from end)

• "hello"[1:4] "ell" # slicing

• len("hello") 5 # size

• "hello" < "jello" 1 # comparison

• "e" in "hello" 1 # search

• New line: "escapes: \n "

• Line continuation: triple quotes ’’’

• Quotes: ‘single quotes’, "raw strings"

String Methods

• upper()

• lower()

• capitalize()

• count(s)

• find(s)

• rfind(s)

• index(s)

 strip(), lstrip(), rstrip()

 replace(a, b)

 expandtabs()

 split()

 join()

 center(), ljust(), rjust()

Lists

A sequence type

A sequence of values

These values can be of any type

Values in a list are called items or elements

Lists are Mutable

The syntax for accessing list elements is the same as for accessing string characters

The expression inside brackets specifies the index

```
>>> numbers = [7, 34, 56]
>>> numbers[1] = 36        # list is mutable
>>> print(numbers)
[7, 36, 56]
```

Mapping

You can think of a list as a mapping between indices and elements

Each index “maps to” one of the elements

```
num = [2, 34, 56]
```

Indices

List indices work the same way as string indices

• Any integer expression can be used as an index

• if you try to read or write an element that does not exist, you get an IndexError

• If an index has a negative value, it counts backward from the end of the list

‘in’ Operator

The ‘in’ operator also works on lists

```
cheeses = [“Cheddar”, “Mozzarella”, “Blue”]
“Blue” in cheeses   # True
“Brie” in cheeses   # False
```

Traversing a List

The most common way is with a ‘for’ loop

Syntax is the same as for strings

This works well if you only need to read the elements

```
cheeses = [“Cheddar”, “Mozzarella”, “Blue”]
for cheese in cheeses:
    print(cheese)
```

If you want to write or update the elements, you need the indices

Common way is to combine functions ‘range’ and ‘len'

This loop traverses the list and updates each element

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

Nested Lists

A list can contain another list

```
[‘spam’, 1, [‘Brie’, 5, 3.2], 2, [2, 5, 6]]
```

Each internal list still counts as a single element

List Operations

‘+’ operator concatenates lists

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
```

‘*’ operator repeats a list a given number of times

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

List Slices

Slice operator also works on lists

```
>>> t = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’]
>>> t[1:3]
[‘b’, ‘c’]
>>> t[:4]
[‘a’, ‘b’, ‘c’, ‘d’]
>>> t[3:]
[‘d’, ‘e’, ‘f’]
```

If you omit the first index, the slice starts at the beginning

If you omit the second, the slice goes to the end

If you omit both, the slice is a copy of the whole list

```
>>> t[:]
[‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’]
```

append() Method

Python provides methods that operate on lists

append adds a new element to the end of a list

```
>>> t = [‘a’, ‘b’, ‘c’]
>>> t.append(‘d’)
>>> print(t)
[‘a’, ‘b’, ‘c’, ‘d’]
```

extend() Method

extend takes a list as an argument and appends all of the elements

t2 is unmodified

```
>>> t1 = [‘a’, ‘b’, ‘c’]
>>> t2 = [‘d’, ‘e’]
>>> t1.extend(t2)       # append list t2 to t1
```

```
>>> print(t1)
['a', 'b', 'c', 'd', 'e']
```

sort() Method

sort arranges the elements of the list from low to high

List methods are all void; they modify the list and return None

```
>>> t = ['d', 'b', 'c', 'a', 'e']
>>> t.sort()
>>> print(t)
['a', 'b', 'c', 'd', 'e']
```

```
# Create a list t, 1,5,6,7
t = [1,5,6,7]
# Print t
print (t)
# copy t to r list
r = t[:]
# not r = t, will only create another pointer to the list
# print r
print (r)
# Modify 2nd element of r
r[1] = 4
# print r
print (r)
# print t
print (t)
# What do you notice?
# t[1] and r[1] are different
r = t[:]
```

Deleting Elements

pop() - takes the last element

```
>>> t = ['a', 'b', 'c']
>>> x  = t.pop()
>>> x
'c'
```

pop() modifies the list and returns the element that was removed

`t.pop(0)` removes the first element

del statement also deletes elements, when you don't need them
`del t[1]`

remove() Method

If you know the element you want to remove (but not the index), use remove():

```
t = ['a', 'b', 'c']
t.remove('b')
```

The return value from remove is None

To remove more than one element, use del statement

```
t = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’]
del t[1:5]
```

Strings vs. Lists

• A string is a sequence of characters

• A list is a sequence of values

• A list of characters is not the same as a string

• A string is immutable whereas a list is mutable

```
>>> s = 'spam'
>>> t = list(s)
>>> print(t)
['s','p','a','m']
```

split() Method

split() method takes a string and parses its words to form a list of strings

```
>>> s = 'This is an ML class'
>>> t = s.split()
>>> print(t)
['This', 'is', 'an', 'ML', 'class']
```

Delimiter

A delimiter specifies which characters to use as word boundaries while splitting

```
>>> s = 'spam-spam-spam'
>>> t = s.split('-')
>>> t
['spam', 'spam', 'spam']
```

join() Method

join() is the inverse of split()

It takes a list of strings and concatenates the elements to form a single string

```
>>> t = ['This', 'is', 'an', 'ML', 'class']
>>> delimiter = ' '
>>> delimiter.join(t)
'This is an ML class'
```

Objects and Values

a and b both refer to a string, but we don't know whether they refer to the same string

To determine, we can use the 'is' operator

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

When you create two lists, you get two distinct objects, even if they have the same elements

```
>>> a = [1,2,3]
>>> b = [1,2,3]
>>> a is b
False

>>> b = a
>>> b is a
True

>>> b[0] =  17
>>> a
[17, 2, 3]
```

List Arguments

When you pass a list to a function, the function gets a reference to the list

If the function modifies a list parameter, the caller sees the change

Some operations modify lists and other operations create new lists

append() method modifies a list, but the + operator creates a new list

```
t1 = [1,2]
t1.append(3)
t1
```

The difference is important when you write functions that are supposed to modify lists

```
>>> t3 = t1 +[4]
>>> t3
[1, 2, 3, 4]
```

The slice operator creates a new list and the assignment makes t refer to it

```
def bad_delete_head(t):
  t = t[1:]
t1 = [1,2,3]
bad_delete_head(t1)
t1

output:
[1, 2, 3]
```

None of that has any effect on the list passed as an argument

If we want to slice a list we can return it

The list leaves the original list unmodified

```
def tail(t):
  return(t[1:])
t1 = [1, 2, 3]
t2 = tail(t1)
print(t1)
print(t2)

output:
[1, 2, 3]
[2, 3]
```

List Operations

□ append

□ insert

□ index

□ count

□ sort

□ reverse

□ remove

□ pop

□ extend

• Indexing e.g., L[i]

• Slicing e.g., L[1:5]

• Concatenation e.g., L + L

• Repetition e.g., L * 5

• Membership test e.g., ‘a’ in L

• Length e.g., len(L)

List Comprehension

• List comprehensions offer a succinct way to create lists based on existing lists

• When using list comprehensions, lists can be built by leveraging any iterable, including strings and tuples

• Syntactically, list comprehensions consist of an iterable containing an expression followed by a for clause

• General Syntax

`[<output value> <iterator> <conditional stmt>]`

Squaring Numbers

Creating a list of squares of numbers from 1 to 5, without and with list comprehension

```
squares = []
for i in range(1,6):
  squares.append(i**2)
squares

output:
[1, 4, 9, 16, 25]
```

```
[x**2 for x in range(1,6)]

output:
[1, 4, 9, 16, 25]
```

Even numbers

```
[x**2 for x in range(1,6) if x%2 == 0]

output:
[4, 16]
```

Odd numbers

```
[x**2 for x in range(1,6) if x%2]

output:
[1, 9, 25]
```

Using if-else Conditionals

Make the values between 5 and 10 negative (both included)

```
a = [3,4,5,6,7,8,9,10,1,12,13,14,15]
[-val if 5<=val <=10 else val for val in a]

output:
[3, 4, -5, -6, -7, -8, -9, -10, 1, 12, 13, 14, 15]
```

2D List Creation w/out List Comprehension

```
import random

data = []
for i in range(5):
  row = []
  for j in range(5):
    row.append(random.randint(1,10))
  data.append(row)

data

output:
[[2, 4, 8, 1, 5],
 [10, 1, 2, 8, 9],
 [7, 4, 1, 8, 2],
 [4, 2, 6, 2, 3],
 [10, 9, 7, 4, 10]]
```

2D List Creation w/ List Comprehension

```
import random

datal = [[random.randint(1,10) for j in range(5)] for i in range(5)]
datal

output:
[[5, 5, 8, 3, 5],
 [6, 5, 6, 10, 2],
 [8, 9, 7, 4, 9],
 [9, 10, 6, 2, 1],
 [10, 2, 1, 4, 8]]
```

Flattening 2D List

```
datal_flattened = [val for row in datal for val in row]

datal_flattened

output:
[5, 5, 8, 3, 5, 6, 5, 6, 10, 2, 8, 9, 7, 4, 9, 9, 10, 6, 2, 1, 10, 2, 1, 4, 8]
```

Tuples

Introduction

• A sequence type

• A sequence of values

• They are indexed and a lot like lists

• A comma-separated list of values

• It is common to enclose tuples in parentheses

```
t = 'a', 'b', 'c' # or t = ('a', 'b', 'c')
t

output:
('a', 'b', 'c')
```

Tuples

What is a tuple?

• A sequence of values whose elements cannot be modified once it is created

• In other words, it's a read-only array

```
>>> t = ()
>>> t = (1, 2, 3)
>>> t = (1, )
>>> t = 1,
>>> a = (1, 2, 3, 4, 5)
>>> print a[1]   # 2
```

List vs. Tuple

What are common characteristics?

• Both store arbitrary data objects

• Both are of sequence data type

What are differences?

• Tuple doesn’t allow modification

• Tuple doesn’t have methods??

• Tuple supports format strings

• Tuple supports variable length parameter in function call

• Tuple slightly faster

Tuples

To create a tuple with a single element, you have to include a final comma

```
>>> t = 'a',
>>> t
('a',)
```

A single value in parentheses is not a tuple

```
>>> t = ('a')
>>> t
'a'
```

Index Operator

If the argument is a sequence (string, list, or tuple), the result is a tuple with the elements of the sequence

```
>>> t = ('l','o','g','i','c')
>>> print(t)
('l', 'o', 'g', 'i', 'c')
```

Most list operators also work on tuples

```
>>> print(t[0])
l
```

Slice Operator

Slicing

If you try to modify one of the elements of the tuple

Tuples are immutable

```
>>> print(t[0])
l
>>> t[0] = 'a'

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [50], in <module>
----> 1 t[0] = 'a'

TypeError: 'tuple' object does not support item assignment
```

Assignment

If we want to swap two variables we will need a third variable, for example

```
>>> a = 25
>>> b = 45
>>> temp = a
>>> a = b
>>> b = temp
>>> print(a)
>>> print(b)
45
23
```

With tuples it is more elegant

```
>>> print(a,b)
>>> a,b = b,a
>>> print(a,b)
45 25
25 45
```

The right side can be any kind of sequence (string, list, or tuple)

```
email = 'nayeem@cd.umd.edu'
uname,domain = email.split('@')
print("name:",uname,", domain: ", domain)

output:
name: nayeem , domain: cd.umd.edu
```

Tuples as Return Values

```
quot, rem = divmod(9,4)
print(quot)
print(rem)

output:
2
1
```

Variable-Length Argument Tuples

Functions can take a variable number of arguments

A parameter name that begins with a * gathers arguments into a tuple, for example

```
def printall(*args):
  print(*args)
printall(1,'3.5',"test")

output:
(1, '3.5', 'test')
```

Variable Length Arguments

Many of the built-in functions use variable-length argument tuples

For example, max() and min() can take any number of arguments:

```
>>> max(3,4,7)
7

>>> min(1,3,6)
1
```

sum cannot

```
>>> sum(1,3,6)

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [54], in <module>
----> 1 sum(1,3,6)

TypeError: sum() takes at most 2 arguments (3 given)
```

Variable Length Tuples

Write a function called sumall() that takes any number of arguments and returns their sum

```
def sumall(*args):
  s = 0
  for i in args:
    s += i
  return(s)
print(sumall(2,3,4,5))

output:
14
```

Tuple Operations

• Indexing e.g., T[i]

• Slicing e.g., T[1:5]

• Concatenation e.g., T + T

• Repetition e.g., T * 5

• Membership test e.g., ‘a’ in T

• Length e.g., len(T)

Lambda Functions

Anonymous (Lambda) Functions

• An anonymous function is a function that is defined without a name

• While normal functions are defined using the def keyword, anonymous functions are defined using the lambda keyword

• Anonymous functions are typically small, and are used to parameterize an expression

• They are typically created near where they are used

• They can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions

• They are used in combination with the functions filter(), map(), and reduce()

Lambda Function Syntax

• Syntax

• lambda <argumentlist>: <expression>

• Argument list consists of a comma separated list of arguments

• Expression is an arithmetic expression using these arguments

```
>>> p = lambda x,y: x*y
>>> p(3,4)
12
```

```
def m(x,y):
  return(x*y)
m(3,4)

output:
12
```

Why Use Lambda Functions?

The power of a lambda function is evident when it is used (as an anonymous function) inside another function

```
def findlarger():
  value =  lambda x,y: "x is larger" if x >y else "y is larger"
  return(value)

outpuut = findlarger()
print(type(output))
print(output(3,5))

output:
<class 'function'>
y is larger
```

```
def myfunc(n):
  return lambda a: a*n
```

```
mydoubler = myfunc(2) #n=2
print(mydoubler(11)) #a=11

output:
22
```

Functions as parameters

Have you ever wanted to pass an entire function as a parameter

Python has functions as first-class citizens, so you can do this

You simply pass the functions by name

Higher-Order Functions

A higher-order function is a function that takes another unction as a parameter

They are “higher-order” because it’s a function of a function

Examples

– Map

– Reduce

– Filter

Lambda works great as a parameter to higher-order functions if you can deal with its limitations

map Function

`map()` is a function with two arguments

`r = map(func, seq)`

the first argument func is the name of a function

the second is a sequence (e.g., a list) seq

map Function without lambda

```
def celsius(T):
  return((5/9)*(T-32.))
def fahrenheit(T):
  return((9/5)*T + 32)

temperatures = (-10, -20, -30, 30, 40)
F = map(fahrenheit,temperatures)
temp_in_fahrenheit = list(F)
print("Temperature in Fahrenheit: ", temp_in_fahrenheit)

output:
Temperature in Fahrenheit:  [14.0, -4.0, -22.0, 86.0, 104.0]
```

```
C = map(celsius, temp_in_fahrenheit)
temp_in_celsius = list(C)
print(temp_in_celsius)

output:
[-10.0, -20.0, -30.0, 30.0, 40.0]
```

map Function with lambda

```
C = [-10, -20, -30, 30, 40]
F = list(map(lambda x: ((9/5)*x + 32), C))
print("Fahrenheit temp: ", F)
C = list(map(lambda x:((5/9) * (x - 32)), F))
print("Celsius: ", C)

output:
Fahrenheit temp:  [14.0, -4.0, -22.0, 86.0, 104.0]
Celsius:  [-10.0, -20.0, -30.0, 30.0, 40.0]
```

map() with Multiple Lists

• map() can be applied to more than one list

• The lists have to have the same length

• map() will apply its lambda function to the elements of the argument lists

• It first applies to the elements of the 0th index, then to the elements with the 1st index, so on

```
a = [1,2,3,4]
b = [17,12,11,10]
c = [-1,-4,5,9]
sumAB = list(map(lambda x,y: x+y,a,b))
print("sumAB: ", sumAB)

sumABC = list(map(lambda x,y,z:x+y+z,a,b,c))
print("sumABC: ", sumABC)

expABC = list(map(lambda x,y,z:2.5*x+2*y-z,a,b,c))
print("expABC: ", expABC)

output:
sumAB:  [18, 14, 14, 14]
sumABC:  [17, 10, 19, 23]
expABC:  [37.5, 33.0, 24.5, 21.0]
```

Filtering

• filter function filters out all the elements of a list, for which function returns True

- `filter(<function>, list)`

• function, f, is the first argument.

• f returns a Boolean value, i.e. either True or False

• This function will be applied to every element of the list

• Only if f returns True will the element of the list be included in the result list

```
data = [1,3,4,8,5,26]
odd_numbers = list(filter(lambda x : x%2, data))
even_numbers = list(filter(lambda x: x%2==0, data))
print(odd_numbers) # prints [1, 3, 5]
print(even_numbers) # prints [4, 8, 26]
```

reduce

• Function reduce continually applies function to the sequence reduce (func, seq)

• If seq = [s1,s2,s3,…,sn], calling reduce(func, seq) works like this

- At first, func will be applied to s1 and s2
- In the next step, func will be applied to the result of step 1 result and s3, and so on

```
from functools import reduce

data1 = [34,43,56,76]
m = reduce(lambda x,y:x+y, data1)
print(m) # prints sum of list elements (209)

sum = reduce(lambda x,y: x+y, range(1,101))
print(sum) #5050

largest = reduce(lambda x,y : x if x > y else y, [3,25,23,12,4,9])
print(largest) #25
```

array

• Arrays are sequence types and behave very much like lists, except that the type of objects stored in them is constrained

• The type is specified at object creation time by using a type code, which is a single character.

```
from array import array

a = array('f',[2,4,6,8])
print(a) #array('f', [2.0, 4.0, 6.0, 8.0])

array('f', [2.0, 4.0, 6.0, 8.0]) #array('f', [2.0, 4.0, 6.0, 8.0])

help(array)
```

Zip

• zip is a built-in function that takes two or more sequences (lists, sets, etc.), and

• “zips” them into a list of tuples, where each tuple contains one element from each sequence

• Returns a zip object

Lists and Tuples

The result is a list of tuples, where each tuple contains a character from the string and the corresponding element from the list

```
s = 'abc'
t = [0,1,2]
zip(s,t)

output:
<zip at 0x1c9de3ba080> # Need to use tuple() or list() to print
```

Hashing from Two Arrays

```
keys = ['x','y','z']
values = [24,25,26]

d = zip(keys,values)
list(d) #[('x', 24), ('y', 25), ('z', 26)]

d1 = list(d)
list(d1)
d1[0] #('x', 24)
```

```
s1 = {1,3,2}
s2 = {'c','b','a'}
s3 = list(zip(s1,s2))
s3 #[(1, 'a'), (2, 'b'), (3, 'c')]
S1_new, s2_new = zip(*s3)
S1_new #(1, 2, 3)
s2_new #('a', 'b', 'c')
```

Hashing from More than 2 Arrays

```
l1 = [1,2,3,4]
l2 = ['a','b','c','d']
l3 = [2.0,3.0,4.0,5.0]

l4 = zip(l1,l2,l3)
l = list(l4)
l #[(1, 'a', 2.0), (2, 'b', 3.0), (3, 'c', 4.0), (4, 'd', 5.0)]
```

Unzip a list of tuples

```
x,y,z = zip(*l)
print(x) #(1, 2, 3, 4)
print(y) #('a', 'b', 'c', 'd')
print(z) #(2.0, 3.0, 4.0, 5.0)
```

Hashing from Diff. Sized Arrays

```
list(zip(range(5),range(50))) #[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]
```

```
from itertools import zip_longest

a = [1,2,3]
b = ['x','y','z']
c = range(5)

d = zip_longest(a,b,c,fillvalue='*')
list(d) #[(1, 'x', 0), (2, 'y', 1), (3, 'z', 2), ('*', '*', 3), ('*', '*', 4)]
```

Sorting in Parallel

```
a = [1,3,2]
b = ['c','b','a']

c = list(zip(a,b))
print(c) #[(1, 'c'), (3, 'b'), (2, 'a')]
c.sort()
print(c) #[(1, 'c'), (2, 'a'), (3, 'b')]

d = list(zip(b,a))
print(d) #[('c', 1), ('b', 3), ('a', 2)]
d.sort()
print(d) #[('a', 2), ('b', 3), ('c', 1)]
```

Dictionaries

An unordered set of key: value pairs

• A mapping between keys and values

• Like an array indexed by a string

• Values of any type; keys of almost any type

• curly brackets

```
{"name":"Guido", "age":43, ("hello","world"):1, 42:"yes", "flag": ["red","white","blue"]}
```

```
d = {"foo" : 1, "bar" : 2}
print (d["bar"])       #2

dict1 = {}
dict1["foo"] = "yow!"
print (dict1.keys())   #dict_keys(['foo'])
```

Dictionary Details

Keys are immutable

• numbers, strings, tuples of immutables, these cannot be changed after creation

• reason is hashing (fast lookup technique)

• **not** lists or other dictionaries, these types of objects can be changed "in place"

• no restrictions on values

Keys will be listed in arbitrary order

• again, because of hashing

Dictionary vs. List

• A dictionary is like a list

• In a list, the indices must be integers

• In a dictionary they can be almost any type

• And a dictionary is a mapping between keys and values

Methods in Dictionary

• keys()

• values()

• items()

• has_key(key)

• clear()

• copy()

• get(key[,x])

• setdefault(key[,x])

• update(D)

• popitem()

Initialization

The ‘in’ operator works on the keys in a dictionary

• ‘one’ in en2Ks

```
en2Ks = dict() # Either one will create an empty dictionary
en2Ks = {}

en2Ks = {'one':'akh','two':'ze','three':'tre'}
'one' in en2Ks #True
```

Using zip()to form a Dictionary

Form a dictionary from a keys list and a values list using zip

```
keys = ['x','y','z']
values = [24,25,26]
d = {k:v for k,v in zip(keys,values)}
d #{'x': 24, 'y': 25, 'z': 26}
```

Dictionary Comprehension

Like list comprehension, we can create dictionaries using simple expressions

A dictionary comprehension takes the form '{key: value for (key, value) in iterable}'

E.g.: Convert the values to floating point values

```
weight = {'a':35,'b':100,'c':175}
float_weight = {key:float(value) for key,value in weight.items()}
float_weight #{'a': 35.0, 'b': 100.0, 'c': 175.0}
```

Dictionary Comprehension: Alphabet

Create a list of the alphabet

```
import string

alphabet = list(string.ascii_lowercase)
print(alphabet,) #['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u',
```

Create a dictionary with keys as the letters and the values as their position in the alphabet

```
print({alphabet[i-1]:i for i in range(1,len(alphabet)+1)})
#{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6, 'g': 7, 'h': 8, 'i': 9, 'j': 10, 'k': 11, 'l': 12, 'm': 13, 'n': 14, 'o':
```

Values

To see whether a value exists, use a method called values()

```
'ze' in en2Ks.values() #True
```

‘in’ Operator Algorithms

‘in’ operator uses different algorithms for lists and dictionaries

For lists, it uses a search algorithm

For dictionaries, it uses a hashtable

In a hashtable, the ‘in’ operator takes about the same time no matter how many items there are in a dictionary

Looping and Dictionaries

You can use a ‘for’ loop to traverse the keys of a dictionary

Dictionaries have a method called keys that returns the keys of the dictionary, in no particular order, as a list

```
for key in en2Ks:
  print(key,en2Ks[key])

output:
one akh
two ze
three tre
```

Reverse LookUp

Given a dictionary ‘d’ and a key ‘k’

We can find the value using

v = d[k]

This is called lookup

If you have v and you want to find k, you have two problems

• there might be more than one key that maps to the value v

• there is no method for reverse lookup, you have to search for it

Dictionaries and Lists

Lists can appear as values in a dictionary

Consider a dictionary that maps frequencies to letters

A frequency may be mapped to several letters

• In order to represent such a mapping, the values (letters)
should be a list of letters

lists cannot be keys

```
t = [1,2,3]
d = dict()
d[t] = 'oops'
```

Data Type Wrap Up

• Integers: 2323, 3234

• Floating Point: 32.3, 3.1E2

• Complex: 3 + 2j, 1j

• Lists: l = [1,2,3]

• Tuples: t = (1,2,3)

• Dictionaries: d = {‘hello’ : ‘there’, 2 : 15}

• Lists, Tuples, and Dictionaries can store any type (including other lists, tuples, and dictionaries!)

• Only lists and dictionaries are mutable

• All variables are references

Files

Introduction

Most of the programs written so far run for a short
duration. Once the program ends, the data is gone. If we want to see the results again, we have to run the
program again

Persistence

Some programs run for a long time. They store data permanently. The data is available even after the program ends

For example, operating systems and web servers. One way to read and write data is using files. Another way to store data is using a database

Reading a File

Using a built-in function ‘open’

It takes the name of a file and returns a file object

```
fin = open('../Lectures/words.txt')
fin
```

readline()

It can read one line

```
fin = open('words.txt')
fin.readline() #'MSML 605\n'
```

readlines() reads lines into a list

```
fin.readlines()
['Course\n','Spring 2020']
```

Removing End of Line Character

remove end line character

```
fin = open('words.txt')
fin.readline().strip('\n') # 'MSML 605'
```

File Traversal

```
fin = open('words.txt')

for line in fin:
  print(line)

output:
MSML 605
Course
Spring 2020
```

Writing

To write to a file, you have to open it with mode ‘w’ as a second parameter

If the file already exists, opening it in write mode clears out the old data and starts fresh

```
fout = open("output.txt", 'w')
```

Write to a File

```
Line1 = "This is a ML class\n"
fout.write(line1)

Line2 = "We Program in Python language\n"
fout.write(line2)
fout.close()
```

Format Operator

The argument of write has to be a string. If we want to put other values in a file, we have to convert them to strings

An alternative is to use the format operator, %

```
f = open('output.txt', 'w')
x = 53
```

```
  f.write(str(x))
```

Format Sequence

For example, the format sequence ‘%d’ means that the second operand should be formatted as an integer

```

```

####

```

```

####

```

```

####

```

```

####

```

```

####

```

```

####

```

```

####

```

```

####

```

```

####

```

```

####

```

```

####

```

```