



香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen



SCHOOL OF
DATA SCIENCE
數據科學學院

Introduction to Computer Science: Programming Methodology

Lecture 4 Function

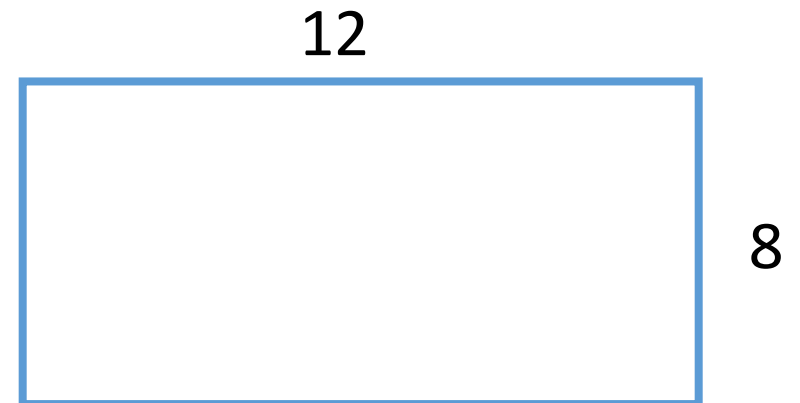
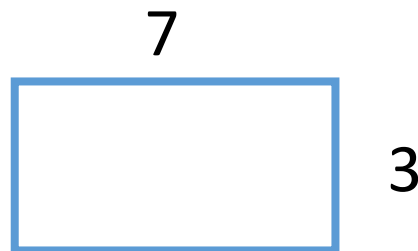
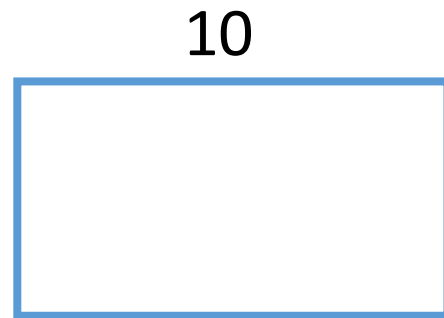
Tongxin Li

School of Data Science

Function Basics

Exercise

Imagine you need to calculate the area of multiple rectangles throughout your program. Print the areas of the following rectangles.



A Python Script

```
# Calculate area of rectangle 1  
length1 = 10  
width1 = 5  
area1 = length1 * width1  
print(f"Area of rectangle 1: {area1}")
```

```
# Calculate area of rectangle 2  
length2 = 7  
width2 = 3  
area2 = length2 * width2  
print(f"Area of rectangle 2: {area2}")
```

```
# Calculate area of rectangle 3  
length3 = 12  
width3 = 8  
area3 = length3 * width3  
print(f"Area of rectangle 3: {area3}")
```

A Python Script

```
# Calculate area of rectangle 1
length1 = 10
width1 = 5
area1 = length1 * width1
print(f"Area of rectangle 1: {area1}")
```

```
# Calculate area of rectangle 2
length2 = 7
width2 = 3
area2 = length2 * width2
print(f"Area of rectangle 2: {area2}")
```

```
# Calculate area of rectangle 3
length3 = 12
width3 = 8
area3 = length3 * width3
print(f"Area of rectangle 3: {area3}")
```

Same Pattern Applies!

Can we define a template and reuse it?

A reusable area calculation

```
def calculate_rectangle_area(length, width):  
    return length * width
```

```
area1 = calculate_rectangle_area(10, 5)  
area2 = calculate_rectangle_area(7, 3)  
area3 = calculate_rectangle_area(12, 8)
```

```
print(area1)  
print(area2)  
print(area3)
```

Output

50

21

96

The indented return statement belongs to the function body. Each call supplies new dimensions.

Stored (and reused) steps

Program

```
def Hello():  
    print('Hello')  
    print('Funny')
```

```
Hello()  
print('Something in the middle.')  
Hello()
```

Output

```
Hello  
Funny  
Something in the middle.  
Hello  
Funny
```

This reusable paragraph of code is usually called a **function**

Python functions

- There are (mainly) two types of functions in **Python**
 - ✓ **Built-in functions** which are part of Python, such as `print()`, `int()`, `float()`, etc
 - ✓ Functions that we define **ourselves** and then use
- The names of built-in functions are usually considered as **new reserved words**, i.e. we **do not** use them as variable names

Function definition

- In Python, a function is some reusable code which can take **arguments** as input, perform some computations, and then output some results
- Functions are defined using reserved word **def**
- We **call/invoke** a function by using the function name, parenthesis and arguments in an expression

Building our own functions

- We create a new function using the **def** key word, followed by **optional parameters** in parenthesis
- We **indent** the body of the function
- This defines the function, but **does not execute** the body of the function

A sample code

Program

```
x=5
print(' Hello')

def print_lyrics():
    print(' I am a lumberjack, and I am okay.')
    print(' I sleep all night and I work all day.')

print(' Yo')
x=x+2
print(x)
```

Output

```
Hello
Yo
7
```

A sample code

Program

```
x=5
print('Hello')

def print_lyrics():
    print('I am a lumberjack, and I am okay.')
    print('I sleep all night and I work all day.')

print('Yo')
print_lyrics()
x=x+2
print(x)
```

Output

```
Hello
Yo
I am a lumberjack, and I am okay.
I sleep all night and I work all day.
7
```



Function definition

Code

```
def rectangle_area(length, width):  
  
    area = length * width  
  
    return area
```

Meaning

def: keyword
Name: rectangle_area
Parameters: length, width
: ends the header
Assignment

Return statement

Function anatomy

Code

```
def add(x, y):  
  
    """Return the sum of two numbers."""  
  
    total = x + y  
  
    return total
```

Meaning

Function: add
Parameters: x, y

Docstring

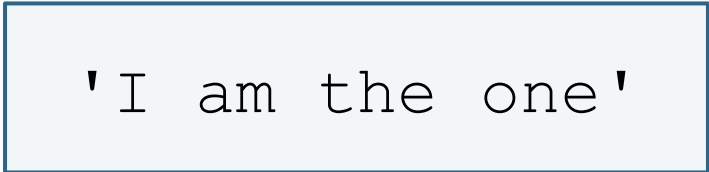
Assignment

Return statement

Arguments

- An argument is a value supplied when a function is called.
- Different arguments let the same function work with different data.

Argument
↓
`big = max( 'I am the one' )`

A diagram illustrating the concept of an argument in a function call. The word "Argument" is written in red text above a blue arrow pointing down to a light blue rectangular box. Inside the box is the string "'I am the one'", which is the argument passed to the function "max" in the code snippet "big = max('I am the one')".

Parameters

- A **parameter** is a **variable** which we use in the function definition that is a '**handle**' that allows the code in the function to **access the arguments** for a **particular** function invocation

```
def greet(lang):  
    if lang=='es':  
        print('Hola')  
    elif lang=='fr':  
        print('Bonjour')  
    else:  
        print('Hello')
```

```
>>> greet('en')  
Hello  
>>> greet('es')  
Hola  
>>> greet('fr')  
Bonjour
```

Multiple parameters/arguments

- We can define **more than one** parameter in a function definition
- We simply add **more arguments** when we **call** the function
- We **match** the **number** and **order** of arguments and parameters

```
def AddTwo(a, b):  
    total = a+b  
    return total
```

```
x=AddTwo(3, 5)  
print(x)
```

Return values

- Often a function will take its **arguments**, do some computation and **return** a value to be used as the value of the function call in the **calling expression**. The **return** keyword is for this purpose.

Program

```
def greet():  
    return 'Hello'  
  
print(greet(), 'Glenn')  
print(greet(), 'Sally')
```

Output

```
Hello Glenn  
Hello Sally
```

Return values

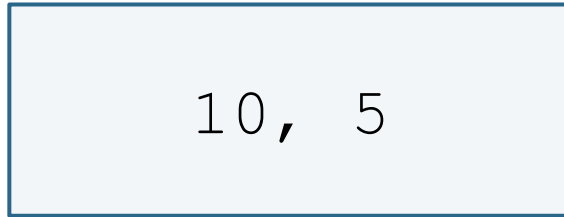
- A fruitful function is one that produces a **result** (or **return value**)
- The **return** statement **ends** the function execution and 'sends back' the **result** of the function

```
def greet(lang):  
    if lang=='es':  
        return 'Hola'  
    elif lang=='fr':  
        return 'Bonjour'  
    else:  
        return 'Hello'
```

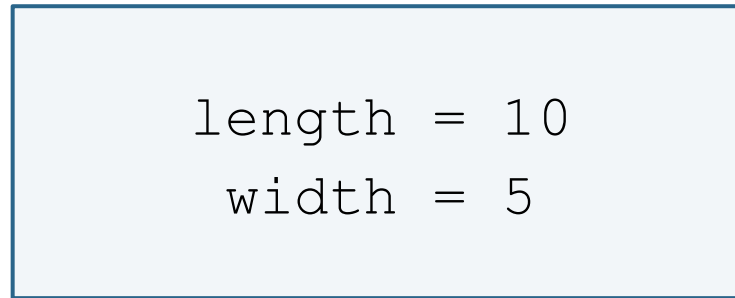
```
>>> print(greet('en'),'Glenn')  
Hello Glenn  
>>> print(greet('es'),'Sally')  
Hola Sally  
>>> print(greet('fr'),'Michael')  
Bonjour Michael
```

Arguments, parameters, and result

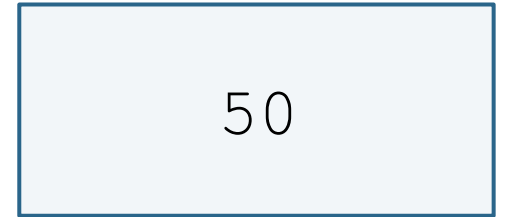
Arguments



Parameters for this call



Returned value



```
def rectangle_area(length, width):  
    return length * width
```

```
area = rectangle_area(10, 5)
```

The function returns 50. The assignment stores that returned value in `area`.

Does every function need a return in Python?

Functions that return None

- A function **without** a return statement returns None.
- A return statement with no expression also returns None.
- These functions can still perform actions, such as printing text.

Functions without return

- When a function has **no return statement**, it will return **None**

```
# Print grade for the score
def printGrade(score):
    if score >= 90.0:
        print('A')
    elif score >= 80.0:
        print('B')
    elif score >= 70.0:
        print('C')
    elif score >= 60.0:
        print('D')
    else:
        print('F')

def main():
    score = eval(input("Enter a score: "))
    print("The grade is ", end = " ")
    printGrade(score)

main() # Call the main function
```

Scope of variables

- The **scope** of a variable is the part of program where this variable can be accessed
- A variable created inside a function is referred to as a **local variable**
- **Global variables** are created outside all functions and are accessible to all functions in their scope

```
globalVar = 1
def f1():
    localVar = 2
    print(globalVar)
    print(localVar)
```

```
f1()
print(globalVar)
print(localVar) # Out of scope, so this gives an error
```

Scope of variables

- Different variables may **share a name** if they have **different scopes**

```
x = 1
def f1():
    x = 2
    print(x) # Displays 2
```

```
f1()
print(x) # Displays 1
```

Global variable

- In a function, you can use keyword `global` to specify that a variable is a `global variable`
- Be very careful when define and use global variable

```
x = 1
def increase():
    global x
    x = x + 1
    print(x) # Displays 2
```

```
increase()
print(x) # Displays 2
```

Default argument

- Python allows you to define functions with **default argument values**
- The default argument values will be passed to the function, when it is invoked **without arguments**

```
def printArea(width = 1, height = 2):  
    area = width * height  
    print("width:", width, "\theight:", height, "\tarea:", area)
```

```
printArea() # Default arguments width = 1 and height = 2  
printArea(4, 2.5) # Positional arguments width = 4 and height = 2.5  
printArea(height = 5, width = 3) # Keyword arguments width  
printArea(width = 1.2) # Default height = 2  
printArea(height = 6.2) # Default width = 1
```

Return multiple values

- Python allows a function to return **multiple values**
- The sort function returns two values; when it is invoked, you need to pass the returned values in a **simultaneous assignment**

```
def sort(number1, number2):  
    if number1 < number2:  
        return number1, number2  
    else:  
        return number2, number1
```

```
n1, n2 = sort(3, 2)  
print("n1 is", n1)  
print("n2 is", n2)
```

To function or not function...

- Organize your code into paragraphs - capture a complete thought and name it
- Don't repeat yourself – name it to work once and reuse it
- If something goes too complex, break up them into several blocks, and put each of them into a function
- Make a library of common stuffs that you do over and over again – perhaps share with other people

Practice

- Write a **function** to instruct the user to input the working hours and hourly rate, and then **return** the salary. If the working hours exceed 40 hours, then the extra hours received 1.5 times pay.

Practice 2: Personal income tax

Write a function that returns tax on an annual taxable salary using these marginal rates.

Annual taxable salary	Marginal rate
$0 \leq \text{salary} \leq 36,000$	3%
$36,000 < \text{salary} \leq 144,000$	10%
$144,000 < \text{salary} \leq 300,000$	20%
$300,000 < \text{salary} \leq 420,000$	25%
$420,000 < \text{salary} \leq 660,000$	30%
$660,000 < \text{salary} \leq 960,000$	35%
$\text{salary} > 960,000$	45%

Example

Taxable salary: 60,000

$$\begin{aligned} &36,000 \times 3\% \\ &+ 24,000 \times 10\% \end{aligned}$$

Tax = 3,480

Each rate applies only to the part of salary within its band. Rates are for this exercise.

Tax calculation

```
def tax(salary):  
    if salary <= 36000:  
        return salary * 0.03  
    elif salary <= 144000:  
        return 1080 + (salary - 36000) * 0.10  
    elif salary <= 300000:  
        return 11880 + (salary - 144000) * 0.20  
    elif salary <= 420000:  
        return 43080 + (salary - 300000) * 0.25  
    elif salary <= 660000:  
        return 73080 + (salary - 420000) * 0.30  
    elif salary <= 960000:  
        return 145080 + (salary - 660000) * 0.35  
    else:  
        return 250080 + (salary - 960000) * 0.45
```

Example

`tax(60000)`

3,480

Each constant is the tax from the completed lower bands.

Exercise assumptions: salary is nonnegative and already excludes tax-free amounts.

String Methods

Python strings

Length and tests

```
len(s)  
part in s  
s.startswith(prefix)
```

Case and replacement

```
s.lower()  
s.upper()  
s.replace(old, new)
```

Whitespace

```
s.strip()  
s.lstrip()  
s.rstrip()
```

Characters and slices

```
s[i]  
s[start:stop]  
s[start:stop:step]
```

```
s = "banana"
```

0	1	2	3	4	5
b	a	n	a	n	a
-6	-5	-4	-3	-2	-1

String type

- A **string** is a sequence of **characters**
- A string literal uses quotes " or ""
- For strings, + means "concatenate"
- When a string contains numbers, it is still a string
- We can convert numbers in a string into a number using int() or float()

Reading and converting

- We prefer to read data in using strings and then parse and convert the data as we need
- This gives us more control over error situations and/or bad user inputs
- Raw input numbers must be converted from strings

Looking inside strings

Code

```
fruit = "banana"
letter = fruit[1]
print(letter)
```

```
n = 3
w = fruit[n - 1]
print(w)
```

Characters and indexes

0	1	2	3	4	5
b	a	n	a	n	a

Output

a index 1
n index 3 - 1 = 2

- Indexes begin at 0.
- An index expression is evaluated before selecting the character.

Index out of range

A string with five characters has nonnegative indexes 0 through 4.

```
name = "Alice"  
name[6]
```

0	1	2	3	4
A	l	i	c	e

There is no character at index 6.

Result

```
IndexError: string index out of range
```

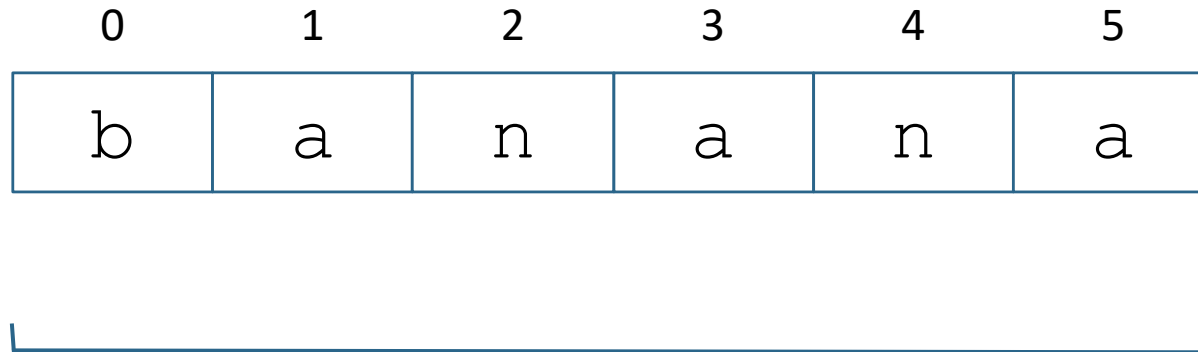
For a nonempty string, the last nonnegative index is `len(string) - 1`.

Strings have length

```
fruit = "banana"  
print(len(fruit))
```

Output

6



6 characters

The len() function

```
fruit = "apple"  
length = len(fruit)  
print(length)
```

Output
5

Input string

"banana"

Built-in function

len(...)

Returned integer

6

len() returns an integer. The original string stays the same.

Looping through strings

- Using a **for** statement, we can easily loop through **each character** in a string

```
fruit = 'I am the one, Morpheus'

n = 0
for i in fruit:
    print(n, i)
    n=n+1

print(' finished' )
```

- String is essentially a **list** in Python

```
0 I
1
2 a
3 m
4
5 t
6 h
7 e
8
9 o
10 n
11 e
12 ,
13 M
14 o
15 r
16 p
17 h
18 e
19 u
20 s
finished
```

Practice

- Write a program to use a **while** statement together with **len()** function to loop through a given string.

Loop and counting

- This is a simple statement that loops through each letter in a string and counts the number of times the loop encounters the 'a' character

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print("The number of 'a' we have seen is:", count)
```

Parts of a for loop

Iteration variable



Six-character string



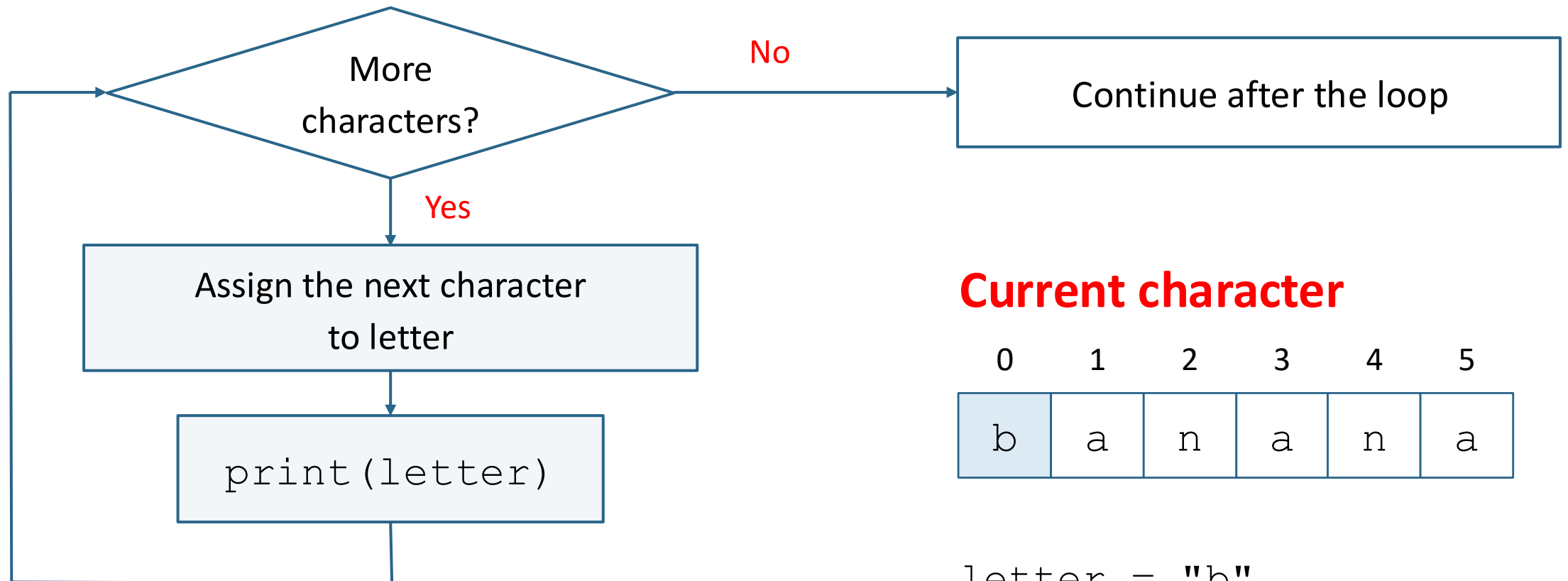
```
for letter in "banana":  
    print(letter)
```

← Loop body

Values assigned to letter

"b" "a" "n" "a" "n" "a"

Looping through a string



Slicing strings

```
s = "Monty Python"
```

Highlighted: `s[0:4]`

0	1	2	3	4	5	6	7	8	9	10	11
M	o	n	t	y		P	y	t	h	o	n

space

Expression

Result

Selected portion

`s[0:4]`

"Mont"

Indexes 0–3

`s[6:7]`

"P"

Index 6

`s[6:20]`

"Python"

Indexes 6–11

Include start and exclude stop. A stop past the end is allowed.

Slicing with omitted bounds

```
s = "Monty Python"
```

Highlighted: `s[:6]`

0	1	2	3	4	5	6	7	8	9	10	11
M	o	n	t	y		P	y	t	h	o	n
space											

Expression	Result	Selected portion
<code>s[:6]</code>	"Monty "	Indexes 0–5 (space at 5)
<code>s[3:]</code>	"ty Python"	Indexes 3–11
<code>s[:]</code>	"Monty Python"	The whole string

Missing start means 0. Missing stop means `len(s)`.

Using 'in' in conditional statement

- The **in** keyword can also be used to check whether one string is in another string
- The **in** expression is a **logical expression** and returns **True** or **False**
- It can be used in **if** or **while** statement

```
>>> fruit = 'banana'
>>> 'n' in fruit
True
>>> 'm' in fruit
False
>>> 'nan' in fruit
True
>>> if 'a' in fruit:
        print('Got cha!')
```

```
Got cha!
```

String library

- Strings provide **methods** for common text operations.
- Call a method using a dot, for example **`greet.lower()`**.
- Methods **do not** change the original string. `lower()` returns text, `find()` an integer, and `startswith()` a Boolean.

```
greet = 'Hello, Students in CSC1001!'
zap = greet.lower()
print(zap)
```

```
hello, students in csc1001!
```

```
print(greet)
```

```
Hello, Students in CSC1001!
```

```
print('Hello, Alice'.lower())
```

```
hello, alice
```

```
>>> stuff = 'hello world'
>>> type(stuff)
<class 'str'>
>>> dir(stuff)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattr__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

<https://docs.python.org/3/library/stdtypes.html#string-methods>

4.7.1. String Methods

Strings implement all of the [common](#) sequence operations, along with the additional methods described below.

Strings also support two styles of string formatting, one providing a large degree of flexibility and customization (see `str.format()`, [Format String Syntax](#) and [String Formatting](#)) and the other based on C `printf` style formatting that handles a narrower range of types and is slightly harder to use correctly, but is often faster for the cases it can handle ([printf-style String Formatting](#)).

The [Text Processing Services](#) section of the standard library covers a number of other modules that provide various text related utilities (including regular expression support in the `re` module).

`str.capitalize()`

Return a copy of the string with its first character capitalized and the rest lowercased.

`str.casefold()`

Return a casefolded copy of the string. Casefolded strings may be used for caseless matching.

Casefolding is similar to lowercasing but more aggressive because it is intended to remove all case distinctions in a string. For example, the German lowercase letter 'ß' is equivalent to "ss". Since it is already lowercase, `lower()` would do nothing to 'ß'; `casefold()` converts it to "ss".

The casefolding algorithm is described in section 3.13 of the Unicode Standard.

New in version 3.3.

`str.center(width[, fillchar])`

Return centered in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

`str.count(sub[, start[, end]])`

Return the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

Searching a string

```
fruit = "banana"
```

First "na" starts at index 2

0	1	2	3	4	5
b	a	n	a	n	a

Match found

```
pos = fruit.find("na")  
print(pos)
```

Output: 2

No match

```
aa = fruit.find("z")  
print(aa)
```

Output: -1

find returns the **first matching** index, or **-1** when the text is not found.

Making everything upper or lower case

- You can convert a string into **upper case** or **lower case**

- Hint: often when we use `find()` to find a substring, we convert the original string into lower case first, so that we don't need to worry about case

```
>>> myStr = 'I am the one, I will beat Matrix'
>>> newStr = myStr.upper()
>>> print(newStr)
I AM THE ONE, I WILL BEAT MATRIX
>>> newStr = myStr.lower()
>>> print(newStr)
i am the one, i will beat matrix
```

Search and replace

- The `replace()` function is like a “search and replace” operation in a word processor
- It **replaces all occurrences** of the search string with the replacement string

```
>>> greet = 'Hello, Bob'
>>> newStr = greet.replace('Bob', 'Jane')
>>> print(newStr)
Hello, Jane
>>> newStr = greet.replace('o', 'X')
>>> print(newStr)
HellX, BXb
>>> newStr = greet.replace('z', 'X')
>>> newStr
'Hello, Bob'
```

Stripping whitespace

- Sometimes we want to take a string and remove **whitespaces** at the beginning and/or end
- `lstrip()` and `rstrip()` to the left and right only
- `strip()` removes **both** beginning and ending whitespaces

```
>>> greet = '  Hello Bob  '
>>> greet.lstrip()
'Hello Bob'
>>> greet.rstrip()
'  Hello Bob'
>>> greet.strip()
'Hello Bob'
....
```

Prefixes

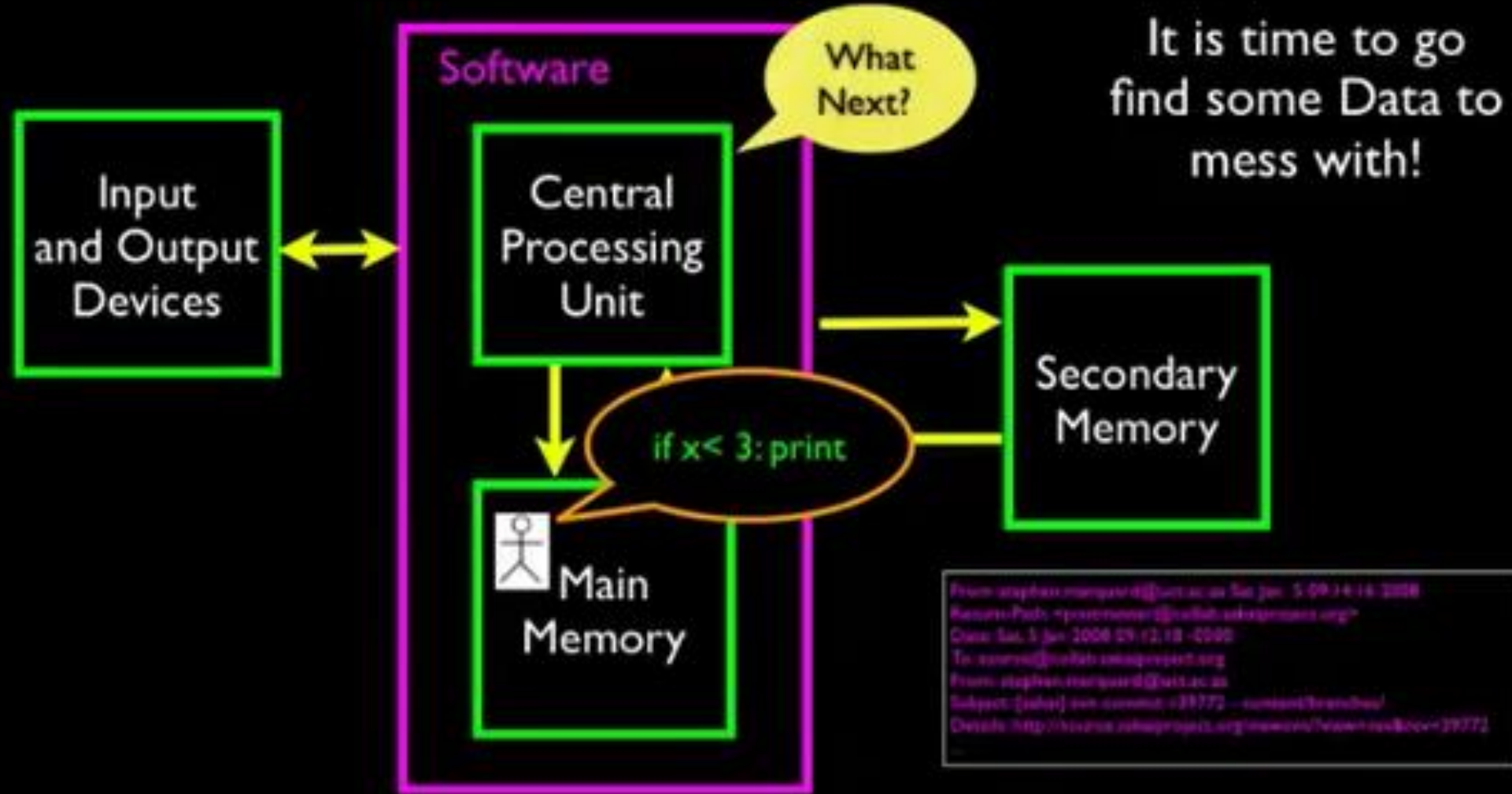
- `startswith()` function checks whether a string is starting with a given string

```
>>> line = 'Please submit your application'
>>> line.startswith('Please')
True
>>> line.startswith('p')
False
```

Example

```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2016'
>>> atpos = data.find(' @' )
>>> print(atpos)
21
>>> sppos = data.find(' ', atpos)
>>> print(sppos)
31
>>> host = data[atpos+1:sppos]
>>> print(host)
uct.ac.za
```

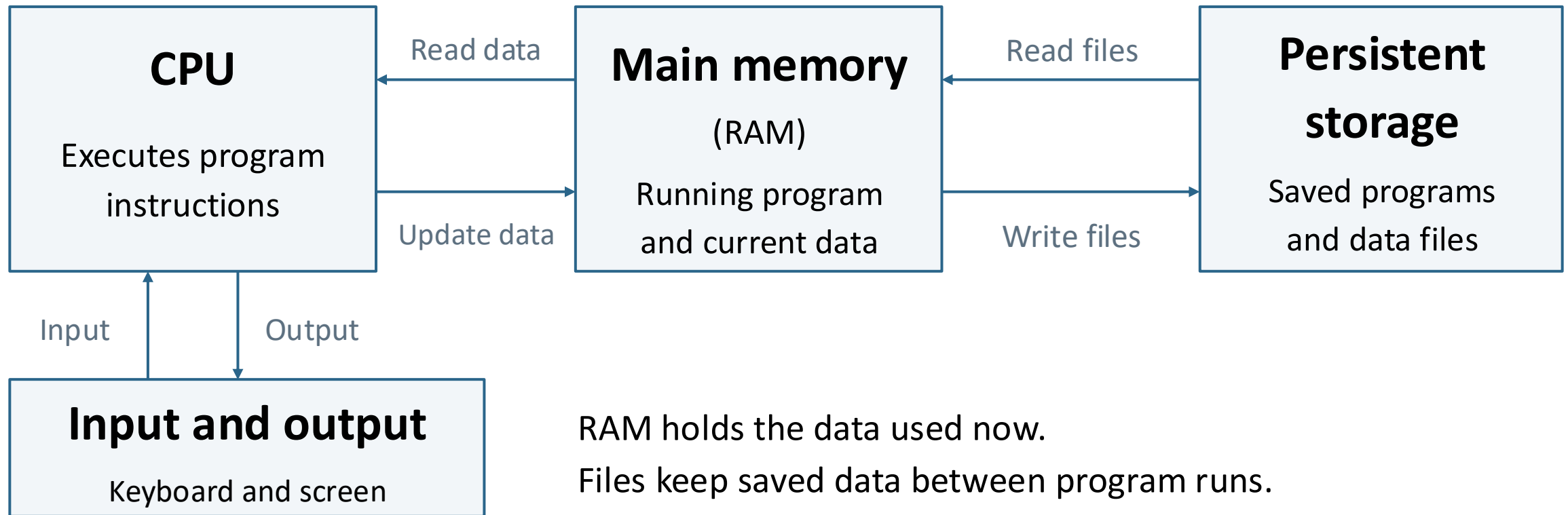
File Operations



From: stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
Return-Path: <marquard@uwaterloo.ca>
Date: Sat, 5 Jan 2008 09:12:18 -0500
To: <marquard@uwaterloo.ca>
From: stephen.marquard@uct.ac.za
Subject: [what] python -> 33772 -- comment/branch
Details: <http://source.sakaiproject.org/viewsvn/?view=rev&rev=33772>
...

Programs, memory, and files

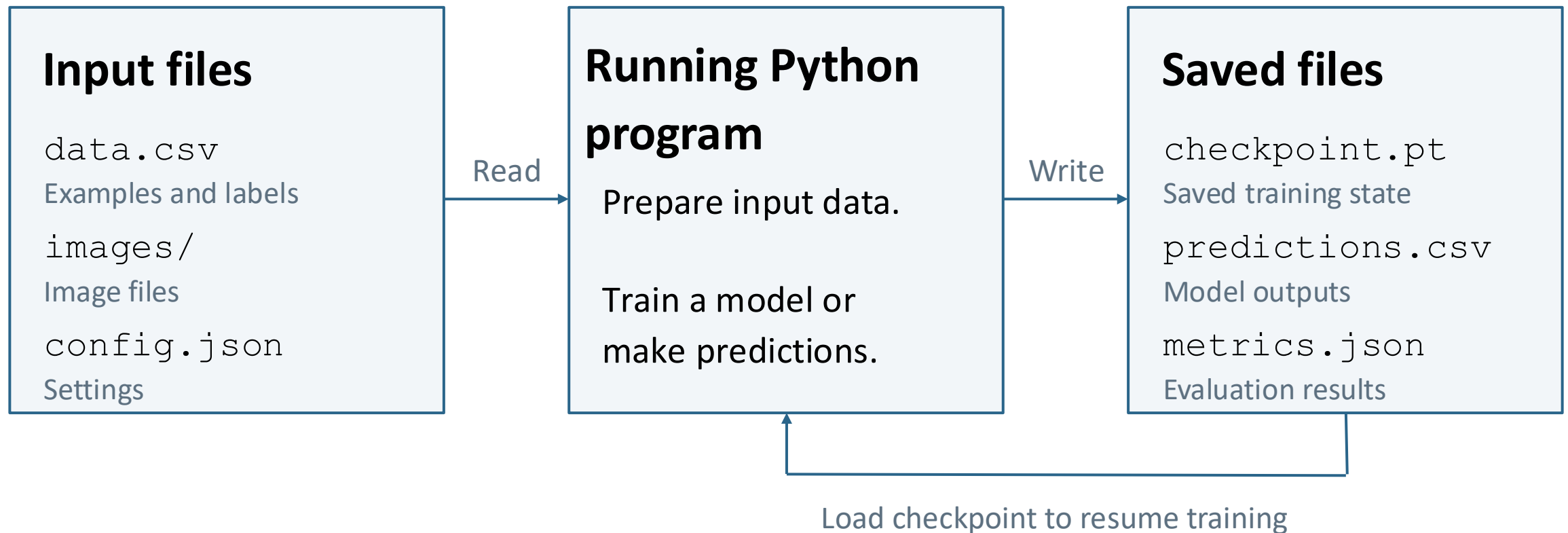
A program uses current data in memory and saves data in files.



Conceptual data flow: file operations transfer data between memory and storage.

File operations in machine learning

A Python program reads inputs and saves results for later use.



Saved files preserve data, model outputs, and training progress between runs.

File processing

- A text file can be thought of as a sequence of lines

```
# Gmail web Start
216.239.38.125 chatenabled.mail.google.com
216.239.38.125 filetransferenabled.mail.google.com
216.239.38.125 gmail.com
216.239.38.125 gmail.google.com
216.239.38.125 googlemail.l.google.com
216.239.38.125 inbox.google.com
216.239.38.125 isolated.mail.google.com
216.239.38.125 m.gmail.com
216.239.38.125 m.googlemail.com
216.239.38.125 mail.google.com
216.239.38.125 www.gmail.com
# Gmail web End
```

Opening files

- Before we can read the contents of a file, we must tell Python **which file** we are going to work with and **what we will do** with that file
- This is done with the **open()** function
- Open() returns a “**file handle**” - a variable used to perform operations on files
- Kind of like “File -> Open” in a word processor

Using open()

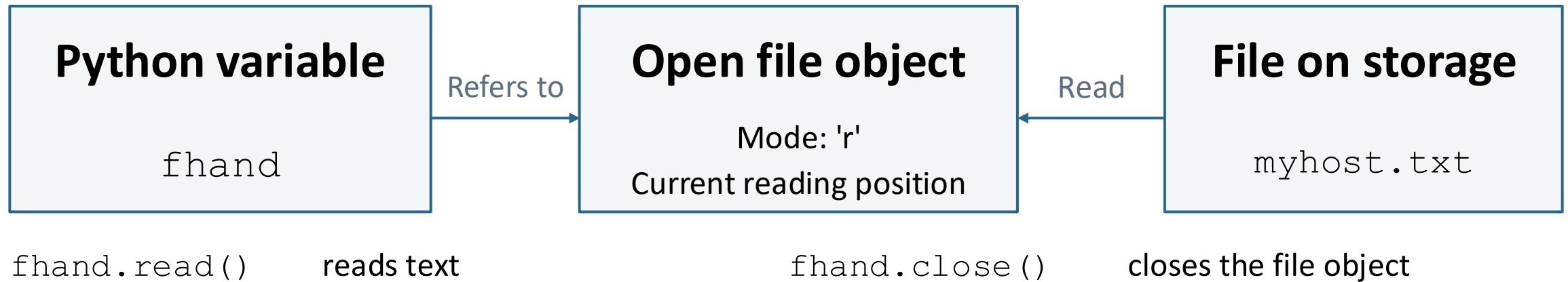
- `handle = open(filename, mode)`
- Returns a `handle` used to manipulate the file
- `Filename` is a string
- `Mode` is optional, use 'r' if we want to read the file, and 'w' if we want to write to the file ('r', 'w', 'a', 'r+', 'w+', 'a+', more for binary files ...)

File handles

```
fhand = open('myhost.txt', 'r')  
print(fhand)
```

Example output (abbreviated)

```
<_io.TextIOWrapper name='myhost.txt' mode='r' ...>
```



`open()` returns a file object. `close()` ends access through that open object.

When files are missing

```
>>> fhand = open('notExisting.txt', 'r')
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    fhand = open('notExisting.txt', 'r')
FileNotFoundError: [Errno 2] No such file or directory: 'notExisting.txt'
```

The newline character

- We use a new character to indicate when a line ends called “**newline**”
- We represent it as ‘**\n**’ in strings
- Newline is still **one** character, not two

```
>>> stuff = 'Hello\nWorld'
>>> stuff
'Hello\nWorld'
>>> print(stuff)
Hello
World
>>> stuff = 'X\nY'
>>> print(stuff)
X
Y
>>> len(stuff)
3
```

File processing

- A text file can be thought of as a **sequence of lines**
- A text file has **newline** at the end of each line

```
# Gmail web Start
216.239.38.125 chatenabled.mail.google.com
216.239.38.125 filetransferenabled.mail.google.com
216.239.38.125 gmail.com
216.239.38.125 gmail.google.com
216.239.38.125 googlemail.l.google.com
216.239.38.125 inbox.google.com
216.239.38.125 isolated.mail.google.com
216.239.38.125 m.gmail.com
216.239.38.125 m.googlemail.com
216.239.38.125 mail.google.com
216.239.38.125 www.gmail.com
# Gmail web End
```

File handle as a sequence

- A file **handle** open for read can be treated as a **sequence of strings** where each line in the file is a string in the sequence
- We can use the **for** statement to loop through a sequence

```
fhand = open('myhost.txt', 'r')  
  
for line in fhand:  
    print(line)  
  
fhand.close()
```

Practice

- Write a program to open a file and count how many lines are included in this file

Reading the whole file

- We can read the whole file into a single string

```
fhand = open('myhost.txt', 'r')
allText = fhand.read()
print('The length of the file:', len(allText))
print('The first 20 characters of the file:', allText[:20])
```

Reading the whole file

A **handle** can be considered as a “pointer”, which goes to the end of the character in the file after a single data reading process.

Reading the whole file

- We can read the whole file into a single string

```
fhand = open('myhost.txt', 'r')
allText = fhand.read()
print('The length of the file:', len(allText))
print('The first 20 characters of the file:', allText[:20])
```

Reading the whole file

A file object tracks the reading position. `read()` without a size reads from that position to the end of the file.

Searching through a file

- We can put an if statement in the for loop to print the lines which satisfy certain conditions

```
fhand = open('myhost.txt', 'r')

for line in fhand:
    if line.startswith('#') == True:
        print(line)

print('finished.')
fhand.close()
```

Writing to a file

- To write a file, use the `open()` function with 'w' argument
- Use the `write()` method to write to the file

```
fhand = open('test.txt', 'w')  
fhand.write('The first line\n')  
fhand.write('The second line\n')  
fhand.write('The third line\n')  
fhand.close()
```

Practice

Read a file and make all letters be lower-cased (to a new file).

with ... as (optional)

Benefits:

Automatic Resource Management: The with statement guarantees that the close() method of the file object is called automatically when the block is exited, regardless of whether the block completes successfully or an exception occurs.

Error Handling: If an error occurs within the with block, the file is still safely closed before the exception is propagated.

Handle `tell()` and `seek()` (optional)

```
handle = open('test.txt', 'r')  
  
s = handle.read()  
print(handle.tell())  
handle.close()
```

28

A **handle** can be considered as a “pointer”, which goes to the end of the character in the file after a single data reading process.

`tell()` returns the counter of the **handle**

Handle `tell()` and `seek()` (optional)

```
handle = open('test.txt', 'r')

s = handle.read()
print('First read:\n', s)

handle.seek(0)

s = handle.read()
print('Second read:\n', s)

handle.close()
```

```
First read:
  My name is Bob.
Hello world!
Second read:
  My name is Bob.
Hello world!
```

`seek(0)` resets the counter of **handle**