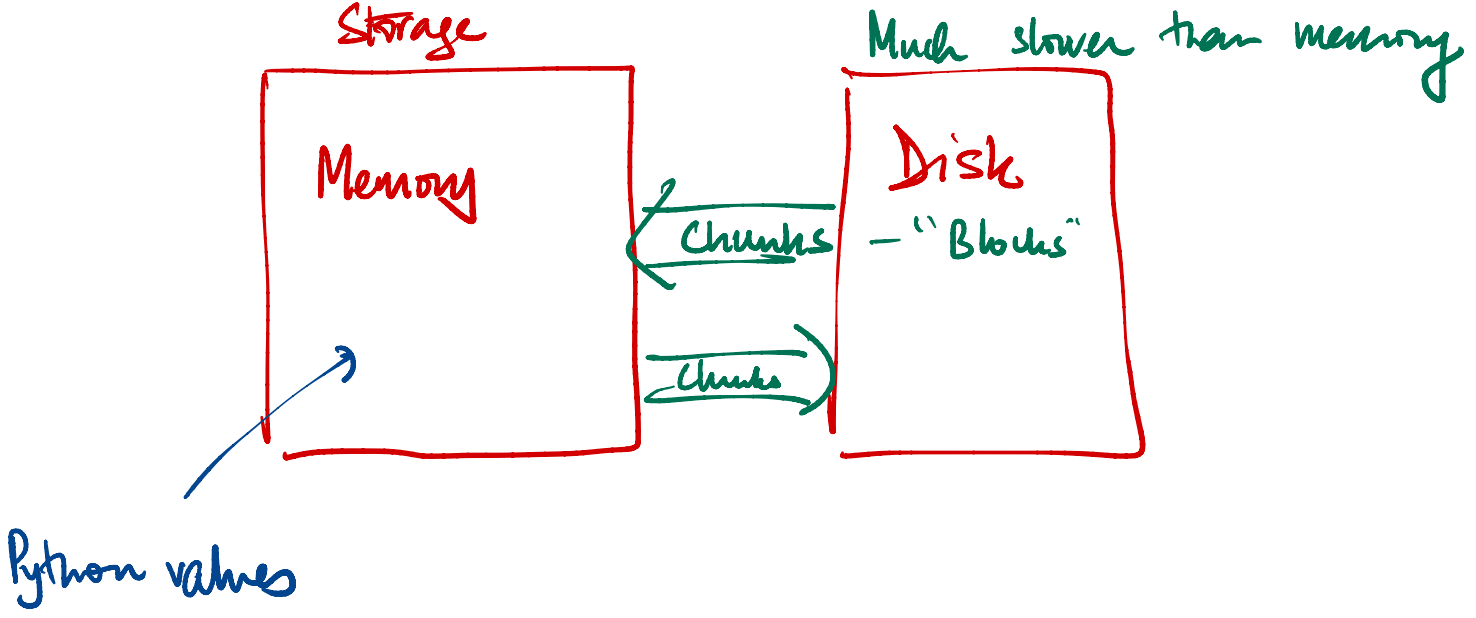
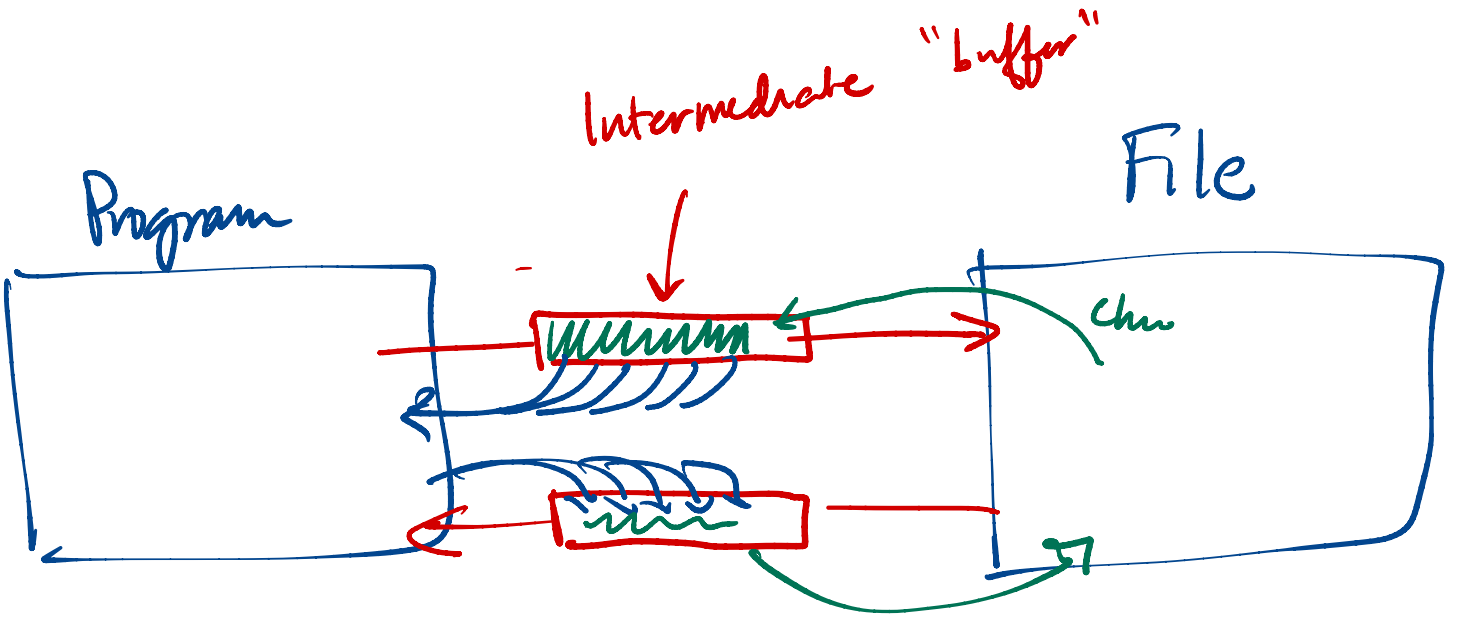
Reading & Writing Files

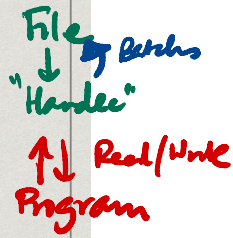
Abstraction of file

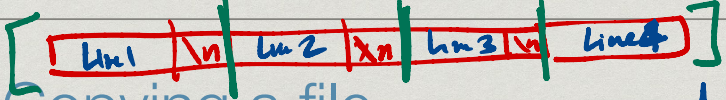



Intermediate "buffer"
Program
File
chunk

Reading/writing disk data

- Open a file — create file handle to file on disk
  - Like setting up a buffer for the file
- Read and write operations are to file handle
- Close a file
  - Write out buffer to disk (flush)
  - Disconnect file handle





Copying a file

File name

Mode of operation: read, write/append

File Handle = name that refer to file

```
infile = open("input.txt", "r")
outfile = open("output.txt", "w")
for line in infile.readlines():
    outfile.write(line)
infile.close()
outfile.close()
```

Open file

close

readlines(): reads the file as a list of strings, line by line

\n

Copying a file

```
infile = open("input.txt", "r")
outfile = open("output.txt", "w")
contents = infile.readlines()
outfile.writelines(contents)
infile.close()
outfile.close()
```

write(line)

list of lines

Functions to read

infile.readline() → a string

infile.readlines() → list of line strings

write()

writelines()

Strip whitespace

space
tab
newline

- `s.rstrip()` removes trailing whitespace

```
for line in contents:
  s = line.rstrip()
```

- `s.lstrip()` removes leading whitespace
- `s.strip()` removes leading and trailing whitespace

Useful to clean up columns in a table

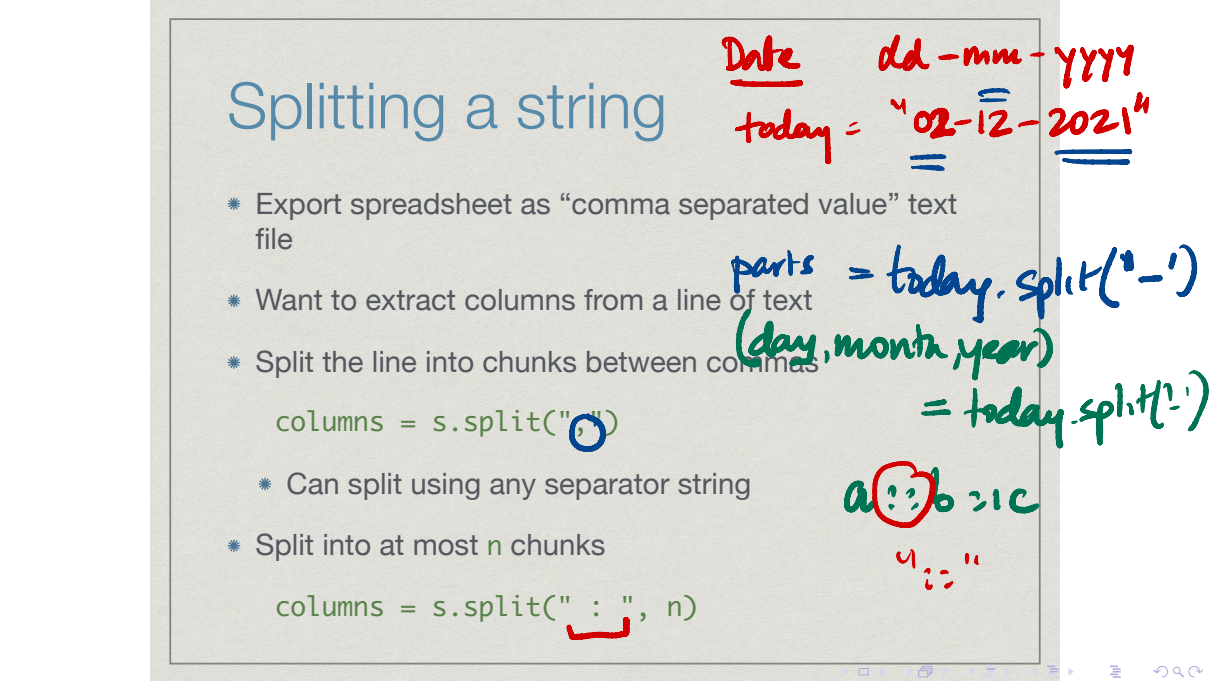
Splitting a string

- Export spreadsheet as “comma separated value” text file
- Want to extract columns from a line of text
- Split the line into chunks between commas

```
columns = s.split(",")
```

  - Can split using any separator string
- Split into at most n chunks

```
columns = s.split(" : ", n)
```



"02-12-2021" ⟶ [2,12,2021]

↓ split('-')

["02", "12", "2021"]

↓int() ↓int() ↓int()

```
x = input()
v = int(x)
```

map(f, [x1, x2 ... xn])

map(int, today.split())

[int(x) for x in today.split()]

≡

[f(x1), f(x2) ... f(xn)]

η if ...

Joining strings

dd day
mm month → "dd-mm-yyyy"
yyyy year

list.join(sep)

"_".join([day, month, year])

- Recombine a list of strings using a separator

```
columns = s.split(",")
joinstring = ","
csvline = joinstring.join(columns)

date = "16"
month = "08"
year = "2016"
today = "-".join([date,month,year])
```

Formatted printing

print(x) - adds a new line

print(y, end="")

print(x, end=",")

- Recall that we have limited control over how `print()` displays output
  - Optional argument `end="..."` changes default new line at the end of print
  - Optional argument `sep="..."` changes default separator between items

space

```
print("[", end="")
for x in l[:-1]:
    print(x, end=",")
print(x[-1], end="")
print("]")
```

$[x_0, x_1, \ldots, x_{n-2}, x_{n-1}]$

```
x = 5
y = 7
z = 10
print(x, y, z)
```

5 , 7 , 10

5␣7␣10

␣ : space

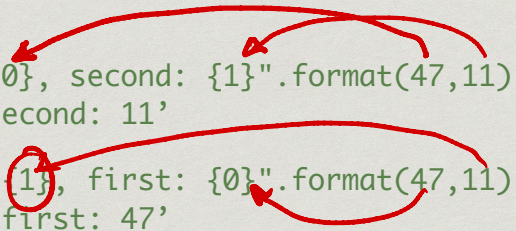
```
print(x, y, z, sep=",")
```

String `format()` method

s.format(v1, v2, ...)

- By example

```
>>> "First: {0}, second: {1}".format(47,11)
'First: 47, second: 11'

>>> "Second: {1}, first: {0}".format(47,11)
'Second: 11, first: 47'
```



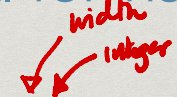
- Replace arguments by position in message string

format() method ...

- Can also replace arguments by name

```
>>> "One: {f}, two: {s}".format(f=47,s=11)
'One: 47, two: 11'

>>> "One: {f}, two: {s}".format(s=11,f=47)
‘One: 47, two: 11'
```

Now, real formatting



add leading 0s

{0:3d}

```
>>> "Value: {0:3d}".format(4)
```

- `3d` describes how to display the value `4`
- `d` is a code specifies that `4` should be treated as an integer value
- `3` is the width of the area to show `4`

`'Value: 004'`

Now, real formatting

```
>>> "Value: {0:6.2f}".format(47.523)
```

- `6.2f` describes how to display the value `47.523`
- `f` is a code specifies that `47.523` should be treated as a floating point value
- `6` — width of the area to show `47.523`
- `2` — number of digits to show after decimal point

```
"Value: _47.52"
```

Real formatting

- Codes for other types of values
  - String, octal number, hexadecimal …
- Other positioning information
  - Left justify
  - Add leading zeroes
- Derived from `printf()` of C, see Python documentation for details

Doing nothing

- Recall: reading a number from the keyboard

```
while(True):
  try:
    userdata = input("Enter a number: ")
    usernum = int(userdata)
  except ValueError:
    print("Not a number. Try again")
  else:
    break
```

Doing nothing

- What if we just want to repeat the loop on an error?

```
while(True):
  try:
    userdata = input("Enter a number: ")
    usernum = int(userdata)
  except ValueError:
    # Do nothing
  else:
    break
```

Can't leave blank

Doing nothing

- Blocks such as `except:`, `else:`, …cannot be empty
- Use `pass` for a null statement

```
while(True):
  try:
    userdata = input("Enter a number: ")
    usernum = int(userdata)
  except ValueError:
    pass
  else:
    break
```

Removing a list entry

- Want to remove `l[4]`?

  ```
  del(l[4])
  ```

- Automatically contracts the list and shifts elements in `l[5:]` left
- Also works for dictionaries
- `del(d[k])` removes the key `k` and its associated value

Undefining a value

- In general, `del(x)` removes the value associated with `x`, makes `x` undefined

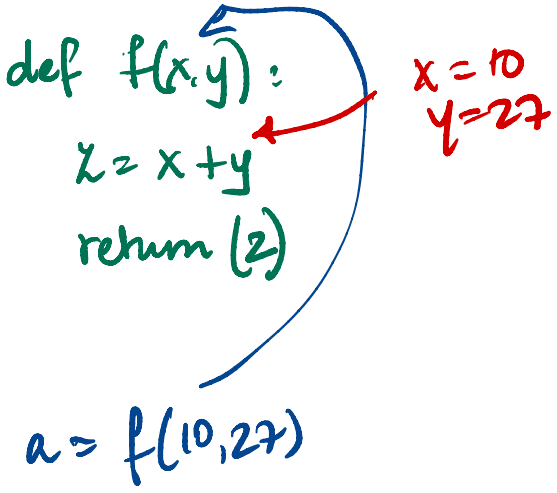
```
x = 7
del(x)
y = x+5

NameError: name 'x' is not defined
```

Checking undefined name

- Assign a value to x only if x is undefined

```
try:
  x
except NameError:
  x = 5
```

Passing parameters to functions



int(s) ⇝ NameError

int(s, base=16)

Convert "AA"

10x16 + 10x1

176

Optional / Default arguments

```
def int(s, b):
    convert s in base b
    to int
```

```
def int(s, b=10):
```

≡

default value of b

```
int(s)      ⇝  int(s, 10)
int(s, 16)  ⇝  b = 16
```

Pass values by name or position

f(a2=v2, a3=v3, ~)

Default values

Named parameter

```
def class Node:
    def __init__(self):
        self.value = None

    def set(self,v):
        self.value = v
```

```
n = Node()
n.set(10)
```

```
def class Node:
    def __init__(self, v=None)
        self.value = v
```

`n = Node()` ⇝ `n.value = None`

`n = Node(10)` ⇝ `n.value = 10`