

## When things go wrong

- $y = x/z$ , but  $z$  has value 0
- $y = \text{int}(s)$ , but string  $s$  is not a valid integer
- $y = 5*x$ , but  $x$  does not have a value
- $y = l[i]$ , but  $i$  is not a valid index for list  $l$
- Try to read from a file, but the file does not exist
- Try to write to a file, but the disk is full

# When things go wrong ...

- Some errors can be anticipated
- Others are unexpected
- Predictable error – exception
  - Normal situation vs exceptional situation
- Contingency plan – exception handling

# Exception handling

- If something goes wrong, provide “corrective action”
  - File not found – display a message and ask user to retype filename
  - List index out of bounds – provide diagnostic information to help debug error
  - Need mechanism to internally trap exceptions

# Types of errors

- Python notifies you of different types of errors
- Most common error, invalid Python code

**SyntaxError:** invalid syntax

- Not much you can do with this!
- We are interested in errors that occur when code is being executed

# Types of errors

Some errors while code is executing (run-time errors)

- Name used before value is defined  
`NameError`: name 'x' is not defined
- Division by zero in arithmetic expression  
`ZeroDivisionError`: division by zero
- Invalid list index  
`IndexError`: list assignment index out of range

# Terminology

- Raise an exception
  - Run time error → signal **error type**, with diagnostic information  
NameError: name 'x' is not defined
- Handle an exception
  - Anticipate and take corrective action based on error type

# Terminology

- Raise an exception
  - Run time error → signal **error type**, with diagnostic information  
`NameError: name 'x' is not defined`
- Handle an exception
  - Anticipate and take corrective action based on error type
- Unhandled exception aborts execution

# Handling exceptions

```
try:  
    . . .  
    . . .  
except IndexError:  
    . . .  
except (NameError, KeyError):  
    . . .  
except:  
    . . .  
else:  
    . . .
```

# Handling exceptions

```
try:  
    . . . ← Code where error may occur  
    . . .  
except IndexError:  
    . . . ← What to do if IndexError occurs  
except (NameError, KeyError):  
    . . . ← Common code to handle multiple errors  
except:  
    . . . ← Catch all other exceptions  
else:  
    . . .
```

# Handling exceptions

```
try:  
    . . . ← Code where error may occur  
    . . .  
except IndexError:  
    . . . ← What to do if IndexError occurs  
except (NameError, KeyError):  
    . . . ← Common code to handle multiple errors  
except:  
    . . . ← Catch all other exceptions  
else:
```

# “Positive” use of exceptions

- Add a new entry to this dictionary

```
scores = {'Dhawan': [3, 22], 'Kohli': [200, 3]}
```

- Batsman `b` already exists, append to list

```
scores[b].append(s)
```

- New batsman, create fresh entry

```
scores[b] = [s]
```

# “Positive” use of exceptions

- Traditional approach

```
if b in scores.keys():
    scores[b].append(s)
else:
    scores[b] = [s]
```

# “Positive” use of exceptions

- Traditional approach

```
if b in scores.keys():
    scores[b].append(s)
else:
    scores[b] = [s]
```

- Using exceptions

```
try:
    scores[b].append(s)
except KeyError:
    scores[b] = [s]
```

# Flow of control

```
..  
x = f(y,z)  
    def f(a,b):  
        ..  
        g(a)  
            def  
            g(m):  
                ..  
                h(m)  
                    def h(s):  
                        ..  
  
IndexError, not handled in h( )→ ..
```

# Flow of control

..  
x = f(y,z)

IndexError  
inherited  
from f( )

def f(a,b):

..

g(a) ← IndexError inherited from g( )

def g(m):  
 ..

IndexError inherited from h( ) → h(m)

Not handled?

def h(s):  
 ..

IndexError, not handled in h( ) → ..

# Summary

- Exception handling allows us to gracefully deal with run time errors
- Can check type of error and take appropriate action based on type
- Can change coding style to exploit exception handling
- When dealing with files and input/output,

# Interacting with the user

- Program needs to interact with the user
  - Receive input.
  - Display output

# Interacting with the user

- Program needs to interact with the user
  - Receive input
  - Display output
- Standard input and output
  - Input from keyboard
  - Output to screen

# Reading from the keyboard

- Read a line of input and assign to userdata

```
userdata = input()
```

- Display a message prompting the user

```
userdata = input("Enter a number")
```

- Add space, newline to make message readable

```
userdata = input("Enter a number: ")
```

```
userdata = input("Enter a number:\n")
```

```
>>> userdata = input()
<empty>
>>> userdata
'<empty>'
>>> userdata = input("Type a number")
Type a number:787
>>> userdata
'787'
>>> userdata = input("Type a number: ")
Type a number: 787
>>> userdata = input("Type a number:\n")
Type a number:
787
>>> userdata
'787'
>>> userdata = input("Type a number:\n")
Type a number:
999
>>> userdata
'999'
>>> |
```

# Reading from the keyboard

- Input is always a string, convert as required

```
userdata = input("Enter a number")
usernum = int(userdata)
```

# Reading from the keyboard

- Use exception handling to deal with errors

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

# Printing to screen

- Print values of names, separated by spaces

```
print(x,y)
```

```
print(a,b,c)
```

- Print a message

```
print("Not a number. Try again")
```

- Intersperse message with values of names

```
print("Values are x:", x, "y:", y)
```

## Fine tuning print()

- By default, `print( )` appends new line character '`\n`' to whatever is printed
  - Each `print( )` appears on a new line
- Specify what to append with argument `end="..."`

```
print("Continue on the", end=" ")
print("same line", end=".\\n")
print("Next line.")
```

# Fine tuning print()

- By default, `print( )` appends new line character '\n' to whatever is printed
  - Each `print( )` appears on a new line
  - Specify what to append with argument `end="..."`

```
print("Continue on the", end=" ") Add space,  
print("same line", end=".\\n") no new line  
print("Next line.")
```

Continue on the same line.  
Next line.

# Fine tuning print()

- By default, `print( )` appends new line character '`\n`' to whatever is printed
  - Each `print( )` appears on a new line
  - Specify what to append with argument `end="..."`

```
print("Continue on the", end=" ") Add space,  
print("same line", end=".\\n") no new line  
print("Next line.") Add full stop,  
new line
```

Continue on the same line.  
Next line.

## Fine tuning print()

- Items are separated by space by default

```
(x,y) = (7,10)
```

```
print("x is",x,"and y is",y,".")
```

```
x is 7 and y is 10 .
```

- Specify separator with argument `sep=""`

```
print("x is",x," and y is ",y,".", sep="")
```

# Formatting print

- May need more control over printing
  - Specify width to align text
  - Align text within width – left, right, centre
  - How many digits before/after decimal point?
- See how to do this later

# Summary

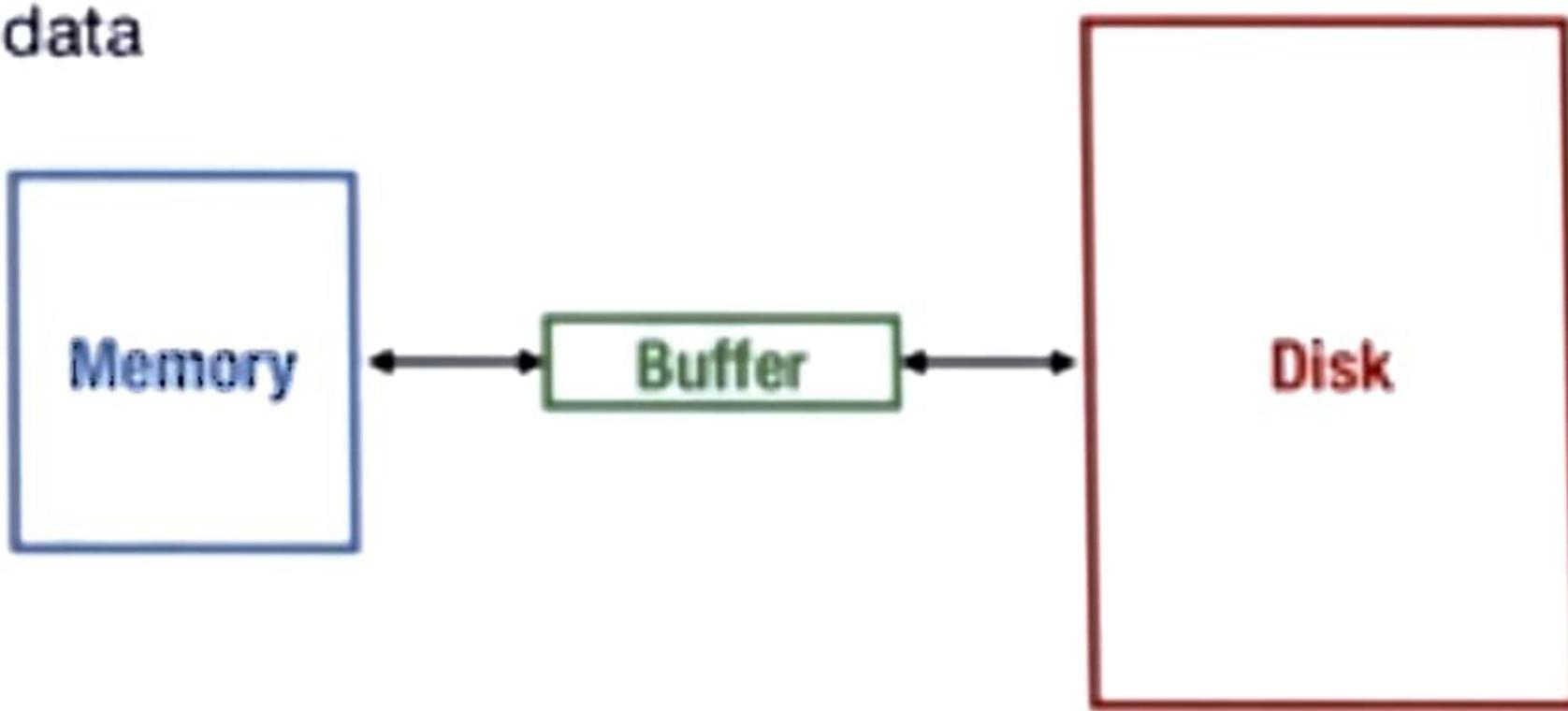
- Read from keyboard using `input()`
  - Can also display a message
- Print to screen using `print()`
  - Caveat: In Python 2, `()` is optional for `print`
- Can control format of `print()` output
  - Optional arguments `end="..."`, `sep="..."`
  - More precise control later

# Dealing with files

- Standard input and output is not convenient for large volumes of data
- Instead, read and write files on the disk
- Disk read/write is much slower than memory

# Disk buffers

- Disk data is read/written in large blocks
- “Buffer” is a temporary parking place for disk data



# 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

# Opening a file

```
fh = open("gcd.py", "r")
```

- First argument to `open` is file name
  - Can give a full path
- Second argument is mode for opening file
  - Read, "r": opens a file for reading only
  - Write, "w": creates an empty file to write to
  - Append, "a": append to an existing file

# Read through file handle

```
contents = fh.read()
```

- Reads entire file into name as a single string

```
contents = fh.readline()
```

- Reads one line into name—lines end with '\n'
  - String includes the '\n', unlike input()

```
contents = fh.readlines()
```

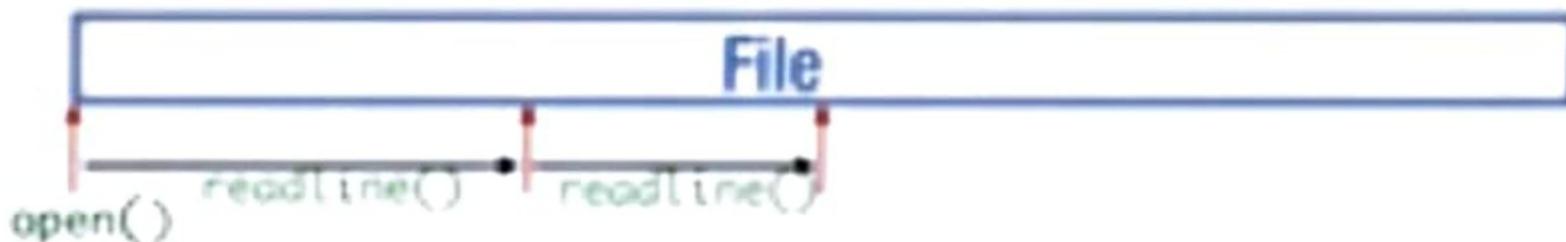
- Reads entire file as list of strings
  - Each string is one line, ending with '\n'

# Reading files



- Reading is a sequential operation
  - When file is opened, point to position 0, the start
  - Each successive `readline()` moves forward
  - `fh.seek(n)` – moves pointer to position n

# Reading files



- Reading is a sequential operation
  - When file is opened, point to position 0, the start
  - Each successive `readline()` moves forward
- `fh.seek(n)` – moves pointer to position `n`
- `block = fh.read(12)` – read a fixed number of characters

## End of file

- When reading incrementally, important to know when file has ended
- The following both signal end of file
  - `fh.read()` returns empty string ""
  - `fh.readline()` returns empty string ""

# Writing to a file

```
fh.write(s)
```

- Write string `s` to file
  - Returns number of characters written
  - Include '`\n`' explicitly to go to a new line

# Writing to a file

```
fh.write(s)
```

- Write string s to file
  - Returns number of characters written
  - Include '\n' explicitly to go to a new line

```
fh.writelines(l)
```

- Write a list of lines l to file
  - Must includes '\n' explicitly for each string

# Closing a file

`fh.close()`

- Flushes output buffer and decouples file handle

- All pending writes copied to disk

`fh.flush()`

- Manually forces write to disk

# Processing file line by line

```
contents = fh.readlines()  
for l in contents:  
    ...
```

- Even better

```
for l in fh.readlines():  
    ...
```

# Copying a file

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

# Copying a file

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

## Strip new line character

- Get rid of trailing '\n'

```
contents = fh.readlines()
for line in contents:
    s = line[:-1]
```

# Strip new line character

- Get rid of trailing '\n'

```
contents = fh.readlines()
for line in contents:
    s = line[:-1]
```

- Instead, use rstrip() to remove trailing whitespace

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

# Strip new line character

- Get rid of trailing '\n'

```
contents = fh.readlines()
for line in contents:
    s = line[:-1]
```

- Instead, use rstrip() to remove trailing whitespace

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

- Also strip() – both sides, lstrip() – from left

- String manipulation functions – coming up

```
mathew@mathewdolphin:...@16-jul/week5/python/files$ ls  
input.txt  
mathew@mathewdolphin:...@16-jul/week5/python/files$ more input.txt  
The quick brown  
fox  
jumps over the lazy  
dog.  
mathew@mathewdolphin:...@16-jul/week5/python/files$ python3.5  
Python 3.5.2 (v3.5.2:4def2a29cc45, Jun 26 2016, 16:47:25)  
[GCC 4.2.1 (Apple Inc. build 5846) (dot 3)] on darwin  
Type "help", "copyright", "credits" or "license" for more information.  
>>> f = open("input.txt","r")  
>>> for line in f.readlines():  
...     print(line)  
...  
The quick brown  
fox  
jumps over the lazy  
dog.  
>>> 
```

```
mathew@mathewdolphin:...@16-jul/week5/python/files$ ls  
input.txt  
mathew@mathewdolphin:...@16-jul/week5/python/files$ more input.txt  
The quick brown  
fox  
jumps over the lazy  
dog.  
mathew@mathewdolphin:...@16-jul/week5/python/files$ python3.5  
Python 3.5.2 (v3.5.2:4def2a29cc6, Jun 26 2016, 16:47:25)  
[GCC 4.2.1 (Apple Inc. build 5846) (dot 3)] on darwin  
Type "help", "copyright", "credits" or "license" for more information.  
>>> f = open("input.txt","r")  
>>> for line in f.readlines():  
...     print(line)  
...  
The quick brown  
fox  
jumps over the lazy  
dog.  
>>> for line in f.readlines():  
...     print(line)  
...  
>>> █
```





```
>>> test = f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'f' is not defined
>>> test = f.read()
>>> test
 ''
>>> test = f.read(100)
>>> test
 ''
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: I/O operation on closed file.
>>> |
```

```
>>> test = f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'f' is not defined
>>> test = f.read()
>>> test
 ''
>>> test = f.readlines()
>>> test
 ''
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
>>> f = open("input.txt","r")
>>> for line in f.readlines():
...     print(line,end="")
...
The quick brown
fox
jumps over the lazy
dog.
>>> 
```



```
>>> f = open("input.txt", "r")
>>> g = open("output.txt", "w")
>>> for line in f.readlines():
...     g.write(line)
...
15
4
29
5
>>> f.close()
>>> g.close()
>>> cat
cat@cat-MacBook-Pro:...000-jul-mod0/python/f1$ cat > output.txt
The quick brown
fox
jumps over the lazy
dog.
cat@cat-MacBook-Pro:...000-jul-mod0/python/f1$ cat |
```

# Summary

- Interact with files through file handles
- Open a file in one of three modes — read, write, append
- Read entire file as a string, or line by line
- Write a string, or a list of strings to a file
- Close handle, flush buffer
- String operations to strip white space

# String processing

- Easy to read and write text files
- String processing functions make it easy to analyse and transform contents
  - Search and replace text
  - Export spreadsheet as text file (csv) and process columns
  - ...

# Strip whitespace

- `s.rstrip()` removes trailing whitespace

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

# Strip whitespace

- `s.rstrip()` removes trailing whitespace

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

- `s.lstrip()` removes leading whitespace

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







# Searching for text

`s.find(pattern)`

- Returns first position in `s` where `pattern` occurs, -1 if no occurrence of `pattern`

# Searching for text

`s.find(pattern)`

- Returns first position in `s` where `pattern` occurs, -1 if no occurrence of `pattern`

`s.find(pattern, start, end)`

- Search for `pattern` in slice `s[start:end]`

`s.index(pattern)`, `s.index(pattern, l, r)`

- Like `find`, but raise `ValueError` if pattern not found

# Searching for text

`s.find(pattern)`

- Returns first position in `s` where `pattern` occurs, -1 if no occurrence of `pattern`

`s.find(pattern, start, end)`

- Search for `pattern` in slice `s[start:end]`

`s.index(pattern)`, `s.index(pattern, l, r)`

- Like `find`, but raise `ValueError` if `pattern` not found

```
>>> a = zeros(100,100,100)
>>> a(:,:,1) = ones(100,100)
>>> a(:,:,2) = 2*ones(100,100)
>>> a(:,:,3) = 3*ones(100,100)
>>> a(:,:,4) = 4*ones(100,100)
>>> a(:,:,5) = 5*ones(100,100)
>>> a(:,:,6) = 6*ones(100,100)
>>> a(:,:,7) = 7*ones(100,100)
>>> a(:,:,8) = 8*ones(100,100)
>>> a(:,:,9) = 9*ones(100,100)
>>> a(:,:,10) = 10*ones(100,100)
```

## Search and replace

`s.replace(fromstr,tostr)`

- Returns copy of `s` with each occurrence of `fromstr` replaced by `tostr`

`s.replace(fromstr,tostr,n)`

- Replace at most first `n` copies
- Note that `s` itself is unchanged — strings are immutable



```
>>> t = "Does the guy dig brown hair"
>>> t.replace("Does", "What")
'What the guy dig brown hair'
>>> t.replace("Does", "What", 0)
'What the guy dig brown hair'
>>> t = "Hello"
>>> t.replace("Hello", "Hi")
'Hi'
>>> t = "Hello"
>>> t.replace("Hello", "Hi")
'Hello'
```

# 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)
```





# Joining strings

- Recombine a list of strings using a separator

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

# Joining strings

- 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])
```



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

## Converting case

- Convert lower case to upper case, ...
- `s.capitalize()` – return new string with first letter uppercase, rest lower
- `s.lower()` – convert all uppercase to lowercase
- `s.upper()` – convert all lowercase to uppercase

## Converting case

- Convert lower case to upper case, ...
- `s.capitalize()` – return new string with first letter uppercase, rest lower
- `s.lower()` – convert all uppercase to lowercase
- `s.upper()` – convert all lowercase to uppercase
- `s.title()`, `s.swapcase()`, ...

# Resizing strings

`s.center(n)`

- Returns string of length n



`s centred, rest blank`

`s.center(n, "*")`

- Fill the rest with \* instead of blanks

`s.ljust(n), s.ljust(n, "*"), s.rjust(n), ...`

- Similar, but left/right justify s in returned string

# Resizing strings

`s.center(n)`

- Returns string of length `n` with `s` centred, rest blank

`s.center(n, "*")`

- Fill the rest with `*` instead of blanks

`s.ljust(n), s.ljust(n, "*"), s.rjust(n), ...`

- Similar, but left/right justify `s` in returned string

```
>>> s = 'Hello'
>>> s.center(20)
'          Hello           '
>>> s.center(20, '-')
'-----Hello-----'
>>> s.ljust(20, '-')
Tried to (just recent cell last):
File "<stdin>", line 1, in <module>
AttributeError: 'str' object has no attribute 'ljust'
>>> s.ljust(20, '-')
'Hello-----'
>>> s.rjust(20, '-')
'-----Hello'
>>> 0
```

## Other functions

- Check the nature of characters in a string.  
`s.isalpha(), s.isnumeric(), ...`
- Many other functions
- Check the Python documentation

# Formatted printing

- 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

# String format() method

- By example

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

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

# String format() method

- 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

```
>>> "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:    4'
```

## Now, real formatting

```
>>> "Value:{0:6.2f}".format(47.52)
```

- 6.2f describes how to display the value 47.523
- f is a code specifies that 47 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:{0:6.2f}"  
54321

# Real formatting

- Codes for other types of values
  - String, octal number, hexadecimal ...
- Other positioning information
  - Left justify
  - Add leading zeroes

## 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
```

# 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

## 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

# Formatted printing



- 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

# String format() method

- By example

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

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

## 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

```
>>> "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:    4'
```

## Now, real formatting

```
>>> "Value: {:.2f}".format(47.523)
```

- `.2f` describes how to display the value `47.523`
- `f` is a code specifies that `47` should be treated as a floating point value

## 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 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

## Now, real formatting

```
>>> "Value:{0:6.2f}".format(47.52)
```

- 6.2f describes how to display the value 47.523
- f is a code specifies that 47 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:{0:6.2f}"  
54321

# 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

- 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
```



## Removing a list entry

- Want to remove `l[4]`?

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

- Automatically contracts the list and shifts elements in `l[5:]` left

## Checking undefined name

- Assign a value to `x` only if `x` is undefined

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

# The value None

- `None` is a special value used to denote “nothing”
- Use it to initialise a name and later check if it has been assigned a valid value

```
x = None
```

```
-
```

```
if x is not None:
```

```
    y = x
```

- Exactly one value `None`

## Summary

- Use `pass` for an empty block
- Use `del()` to remove elements from a list or dictionary
- Use the special value `None` to check if a name has been assigned a valid value



# 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
```

## Removing a list entry

- Want to remove `l[4]`?

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

- Automatically contracts the list and shifts elements in `l[5:]` left

## 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
```

# The value None

- `None` is a special value used to denote “nothing”
- Use it to initialise a name and later check if it has been assigned a valid value

```
x = None
```

```
-
```

```
if x is not None:
```

```
    y = x
```

# The value None

- `None` is a special value used to denote “nothing”
- Use it to initialise a name and later check if it has been assigned a valid value

```
x = None
```

-

```
if x is not None:
```

```
    y = x
```

# The value None

- `None` is a special value used to denote "nothing"
- Use it to initialise a name and later check if it has been assigned a valid value

```
x = None
```

```
-
```

```
if x is not None:
```

```
    y = x
```

- Exactly one value `None`
- `x is None` is same as  
`x == None`

## Summary

- Use `pass` for an empty block
- Use `del()` to remove elements from a list or dictionary
- Use the special value `None` to check if a name has been assigned a valid value