Functional programming and I/O

Instructor: Jorge Mendez
args and kwargs
Positional and keyword arguments

• Python allows you to call functions with and without specifying the argument’s names in the function call
• If you don’t specify the names, they are inferred from the order of the arguments in the call
  • These are positional arguments
• If you do specify the name, then the order in which you pass them does not matter
  • These are keyword arguments
• After a keyword argument, no positional arguments are allowed
Starred expressions

• Python uses the * operator to pack/unpack iterables
• It turns a iterable object (e.g., list) into separate elements
• It also turns separate elements into a tuple
• Useful for passing arbitrarily many parameters to functions
*args

- Pass in arbitrarily many parameters and convert to a tuple
  - Packing
- *args is not a keyword, just a convention
- Can only have one starred argument
- Opposite direction: pass an iterable as multiple parameters
  - Unpacking
*args

def function(*args):
    for a in args:
        print(a)

function(1, 2, 3)

- 1, 2, 3 packed to tuple args
*args

def function(x, *args):
    for a in args:
        print(a)

function(1, 2, 3)

2
3

• Can use both argument types
• 1 is passed to x
• 2, 3 packed to args
Unpacking iterables

```python
def function(x, y, z):
    print(x, y, z)

my_list = [1, 2, 3]
function(*my_list)
```

- Unpack `my_list`
- Pass as each element as separate argument

1 2 3
Unpacking iterables

```
def transpose(nested_list):
    return [list(row)
            for row in zip(*nested_list)]

my_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
transposed_list = transpose(my_list)
print(transposed_list)

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

- * unpacks to three different lists
- `zip()` iterates over the three lists jointly
def function(*args):
    for a in args:
        print(a)

function(x=1, y=2)

TypeError: function() got an unexpected keyword argument 'a'

my_dict = {'x': 1, 'y': 2}
def function(x=0, y=0):
    print(x, y)

function(*my_dict)

x y
Double starred expressions

• Similar to starred expressions, but for dictionaries
• Use the ** operator to pack/unpack dictionaries
• Turns dictionary into separate named elements
  • Names are the keys
• Also turns separate named elements into a dictionary
**kwargs

- Pass in arbitrarily many keyword arguments and convert to dictionary
  - Packing
- Names are not pre-defined
- Opposite direction: pass dictionary as multiple keyword arguments
  - Unpacking
- Same caveats as args apply: just a convention, no more than one
**kwargs

def function(**kwargs):
    print(kwargs['x'], kwargs['y'])

function(x=1, y=2)

1 2
**kwargs

def function(*args, **kwargs):
    print(args)
    print(kwargs)

function(1, 2, 3, x=1, y=2)

(1, 2, 3)
{'x': 1, 'y': 2}

- 1, 2, 3 packed to tuple args
- x=1, y=2 packed to dictionary kwargs
Keyword-only arguments

• Any argument after a starred expression (e.g., *args) must be a keyword argument

• These must be passed in as named arguments (keyword-only)
  • How would the interpreter know where *args ends?

• We can use keyword-only arguments with no default values
Keyword-only arguments

```python
def function(*args, z, w=-1, **kwargs):
    print(args)
    print(z, w)
    print(kwargs)

function(1, 2, 3, z=0, x=1, y=2)
```

- z has no default value, but w does
- Not passing z will give error
- Trying to pass z as positional argument will not work
Order of arguments

• `def function(arg1, ..., argN, *args, kwarg1, ..., kwargM, **kwargs):

• We can pass in keyword-only arguments without `*args`:
  • Use empty starred expression
  • `def function(arg1, ..., argN, *, kwarg1, ..., kwargM, **kwargs):`
Keyword-only arguments

```python
def function(a, *, b):
    print(a, b)

function(1, b=2)
```

- b may not be passed as position argument
  - Must be passed as named
Functional programming
Functional programming

- Programming paradigm that treats computation as evaluation of mathematical functions
- Assumes immutable data
- Assumes stateless execution
- Python is neither of those!
  - Still supports *some* functional programming concepts
Lambda functions

• They are anonymous functions
  • They are not associated to a name

• They are created with the `lambda` keyword
  • This is a reserved keyword!

• They are restricted to a single expression
Lambda functions can be...

• Assigned to a variable
• Returned by a function
• Passed as an argument to a function
Lambda function syntax

• Lambda functions can take on the same types of parameters as regular functions

• `lambda arg1, ..., argN, *args, kwarg1, ..., kwargM, **kwargs: expression`
Lambda function examples

\[
\text{square} = \text{lambda } x: x ** 2 \\
\text{print}(\text{square}(3))
\]

9

- \text{square} is now a function that computes the square of an input
Lambda function examples

```python
def make_incrementor(n):
    return lambda x: x + n

f = make_incrementor(42)
print(f(500))
```

- `make_incrementor` returns an anonymous function
- The anonymous function increments its input by the value of `n`
Lambda function examples

```python
def apply_to_all(x, *, fun):
    for i in range(len(x)):
        x[i] = fun(x[i])

square = lambda x: x ** 2
my_list = [1, 2, 3]
apply_to_all(my_list, fun=square)
print(my_list)

[1, 4, 9]
```

- `apply_to_all` applies `fun` to all elements in a list
- Modifies list in place
Built-in functions

- **map(fun, iterable)** — returns iterator applying `fun` to each element in `iterable`

```python
fun = lambda x: x ** 2
a = [1, 2, 3]
b = list(map(fun, a))
print(b)

[1, 4, 9]
```
Built-in functions

- **filter(fun, iterable)** — returns iterator over values in iterable for which fun evaluates to True

```python
fun = lambda x: x > 2
a = (1, 2, 3, 4)
b = list(filter(fun, a))
print(b)
```

```
[3, 4]
```
Functools

- **functools.partial** *(fun, *args, **kwargs)* — return function *fun* with partial list of *args* and *kwargs*
  - If resulting function is called with additional *args*, they are appended
  - If called with additional *kwargs*, they override previous matching *kwargs*

```python
def add(x, y):
    return x + y

fun = functools.partial(add, 3)
print(fun(5))
```

8
Functools

- **functools.reduce(fun, iterable) —** return function applied sequentially to consecutive pairs in iterable

fun = lambda x, y: x * y
print(functools.reduce(fun, range(1, 5)))

24
Other functions

- `sorted()`, `sort()`, `max()`, and `min()` all take an optional named argument `key`
- Used to extract a key for comparison from each element in iterable
- Default is the element itself (identity function)
sorted() key example

my_list = [4, 1, 3, 5, 2]
sorted_list = sorted(
    enumerate(my_list),
    key=lambda x: x[1])
sorted_idx = [elem[0] for elem in sorted_list]
print(sorted_idx)

[1, 4, 2, 0, 3]

- enumerate creates iterable of tuples (idx, value)
- key accesses the value part for sorting
  - Recall that otherwise comparisons are lexicographic, so that would just sort based on idx (useless!)
- elem[0] accesses the idx
Input / Output
Standard I/O

- `input()` — read from terminal (STD_IN)
  - Stops reading after newline
- `input(str)` — display `str` and then read from terminal
  - Useful for prompting user
- `print(str)` — display `str` in terminal (STD_OUT) with newline
- `print(str, end=' ')` — display `str` without newline
File objects

- Object that can be operated on with a file-oriented API
- File objects are iterable
- They can be raw binary, buffered binary, or text
- Constructor:
  - `f = open(filename, mode)` — creates a file object
- Destructor:
  - `f.close()` — destroys the file object (not the most common)
# File modes

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>Open for reading (default)</td>
</tr>
<tr>
<td>w</td>
<td>Open for writing (overwrite if exists)</td>
</tr>
<tr>
<td>x</td>
<td>Open for exclusive creation (fail if exists)</td>
</tr>
<tr>
<td>a</td>
<td>Open for writing at the end of the file, if exists</td>
</tr>
<tr>
<td>b</td>
<td>Open in binary mode</td>
</tr>
<tr>
<td>t</td>
<td>Open in text mode (default)</td>
</tr>
<tr>
<td>+</td>
<td>Open for reading and writing</td>
</tr>
</tbody>
</table>

- One of r, w, x, a: dictate whether the file is created, appended to, or overwritten
- One of b, t: dictate whether the file is binary or text
- +: dictate whether the file can be both written to and read from
Iterating over file

- File objects are iterable
- Iterate by reading lines
- Does not return to the top of the file after reading
- May not have more than one iterator at different file positions
File iteration example

```python
def_file iteration example
f = open('my_file.txt')
for line in f:
    print(line)
f.close()
```

line 1
line 2
...

```python
```
## File implementation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>f.fileno()</code></td>
<td>Return file descriptor number (integer)</td>
<td></td>
</tr>
<tr>
<td><code>f.read(size=-1)</code></td>
<td>Read at most size characters from file (default: read to EOF)</td>
<td></td>
</tr>
<tr>
<td><code>f.readline()</code></td>
<td>Read until newline or EOF</td>
<td></td>
</tr>
<tr>
<td><code>f.readlines(hint=-1)</code></td>
<td>Read at most hint lines and return as list (default: all lines)</td>
<td></td>
</tr>
<tr>
<td><code>f.seek(pos)</code></td>
<td>Move stream to desired position</td>
<td></td>
</tr>
<tr>
<td><code>f.tell()</code></td>
<td>Print the stream position</td>
<td></td>
</tr>
<tr>
<td><code>f.write(str)</code></td>
<td>Write str to file</td>
<td></td>
</tr>
<tr>
<td><code>f.writelines(list)</code></td>
<td>Write list of strings to file</td>
<td>Newlines not added</td>
</tr>
<tr>
<td><code>f.readable()</code></td>
<td>True if the file supports reading</td>
<td>Related: writable(), seekable()</td>
</tr>
</tbody>
</table>
Context management: `with`

- Python’s `with` statement allows you to manage resources in enclosed context
- The main use we will give this is to open files:

```python
with open('my_file.txt') as f:
    file operations
...
# file is closed upon exiting other operations
```

- Handles exceptions internally as well
Takeaways

- Positional and keyword arguments give us flexibility for passing parameters
- *args and **kwargs for arbitrarily many parameters
- Lambda for creating anonymous functions (used as return values, arguments, variables...)
- File object API provides several useful functionalities