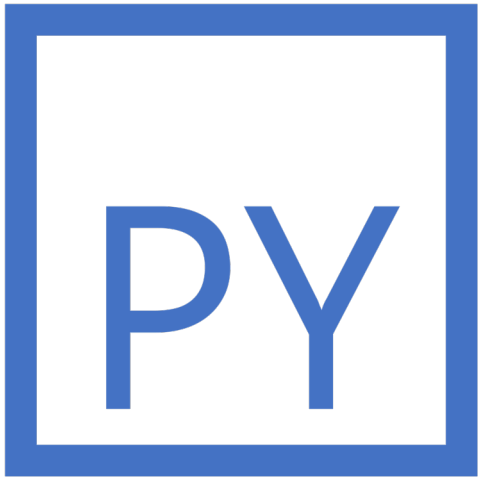
Functional programming and I/O

`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
- Unpack: It turns an iterable object (e.g., list) into separate elements
- Pack: 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

How about for keyword arguments?

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

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

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

```
def function(x=0, y=0):
    print(x, y)

my_dict = {'x': 1, 'y': 2}
function(*my_dict)
```

```
x y
```

Double starred expressions

- Similar to starred expressions, but for dictionaries
- Use the `**` operator to pack/unpack dictionaries
- Unpack: Turns dictionary into separate *named* elements
  - Names are the keys
- Pack: 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

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

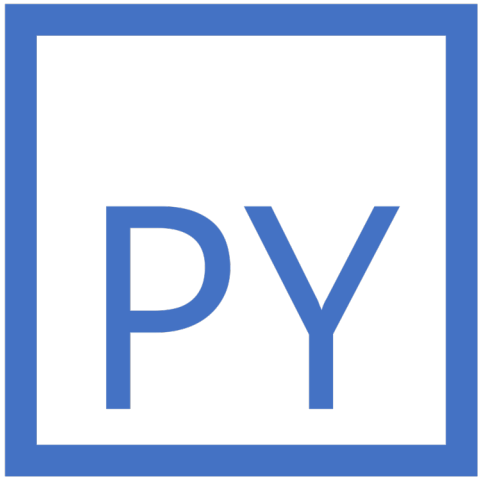
Order of arguments

- `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):`

Built-in functions

- `map(fun, iterable)` — returns iterator applying `fun` to each element in `iterable`
- `filter(fun, iterable)` — returns iterator over values in `iterable` for which `fun` evaluates to `True`

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

Character	Meaning
`r`	Open for reading (default)
`w`	Open for writing (overwrite if exists, else create)
`x`	Open for exclusive creation (fail if exists)
`a`	Open for writing at the end of the file if exists, else create
`b`	Open in binary mode
`t`	Open in text mode (default)
`+`	Open for reading and writing

- 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
- Caveats
  - Does not return to the top of the file after reading
  - May not have more than one iterator at different file positions

File implementation

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

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:

```
with open('my_file.txt') as f:
    file operations
    ...

other operations
```

- Handles exceptions internally as well