Names and Types

- Python associates types with values.
  - languages like C, Perl associate types with variables
- A Python variable can refer to a value of any type.
  - optional type annotations can indicate a variable should refer only to a particular type
- The `type` function allows introspection.

```python
>>> a = 42
>>> type(a)
<intype 'int'>
>>> a = "String"
>>> type(a)
<type 'str'>
>>> a = [1,2,3]
>>> type(a)
<type 'list'>
>>> a = {'ps':50,'cr':65,'dn':75}
>>> type(a)
<type 'dict'>
```

Python Sequences

- Python does not have arrays
  - widely used Python library `numpy` does have arrays
- Python has 3 basic sequence types: lists, tuples, and ranges
  - lists are mutable - they can be changed
  - tuples similar to lists but immutable - they can not be changed
    - some important operations require immutable types, e.g. hashing
  - ranges are immutable sequence of numbers
    - commonly used for loops
Some Useful Python Sequence Operations

These can be applied to lists, tuples and ranges

- `x in s`: True if an item of `s` is equal to `x`
- `x not in s`: False if an item of `s` is equal to `x`
- `s + t`: the concatenation of `s` and `t`
- `s * n`: equivalent to adding `s` to itself `n` times
- `s[i]`: `i`th item of `s`
- `s[i:j]`: slice of `s` from `i` to `j`
- `s[i:j:k]`: slice of `s` from `i` to `j` with step `k`
- `len(s)`: length of `s`
- `min(s)`: smallest item of `s`
- `max(s)`: largest item of `s`
- `s.index(x[, i[, j]])`: index of the first occurrence of `x` in `s` (at or after index `i` and before index `j`)
- `s.count(x)`: total number of occurrences of `x` in `s`

Some Useful Python Mutable Sequence Operations

These can be applied to lists, not tuples or ranges

- `s[i] = x`: item `i` of `s` is replaced by `x`
- `s[i:j] = t`: slice of `s` from `i` to `j` is replaced by elements of `t`
- `del s[i]`: same as `s[i] = []`
- `del s[i:j]`: the elements of `s[i:j]` are replaced by those of `t`
- `del s[i:j:k]`: removes the elements of `s[i:j:k]` from the list
- `s.append(x)`: appends `x` to the end of the sequence
- `s.clear()`: removes all items from `s`
- `s.copy()`: creates a shallow copy of `s`
- `s.extend(t)`: extends `s` with the contents of `t`
- `s *= n`: updates `s` with its contents repeated `n` times
- `s.insert(i, x)`: inserts `x` into `s` at the index given by `i`
- `s.pop(i)` or `s.pop()`: retrieves the item at `i` and also removes it from `s`
- `s.remove(x)`: remove the first item from `s` where `s[i]` is equal to `x`
- `s.reverse()`: reverses the items of `s` in place
- `s.sort()`: sort the items of `s` in place
Ranges

>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> tuple(range(10))
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> list(range(5,10))
[5, 6, 7, 8, 9]
>>> list(range(5,10,3))
[5, 8]
>>> list(range(5, -10, -3))
[5, 2, -1, -4, -7]
>>> list(range(5, 3))
[]

Example - /bin/echo using while

# Python implementation of /bin/echo
# using indexing & while, not pythonesque
import sys
i = 1
while i < len(sys.argv):
    if i > 1:
        print(' ', end='')
    print(sys.argv[i], end='')
    i += 1
print()

Example - /bin/echo using for/range

# Python implementation of /bin/echo
# using indexing & range, not pythonesque
import sys
for i in range(1, len(sys.argv)):
    if i > 1:
        print(' ', end='')
    print(sys.argv[i], end='')
print()
# Python implementation of /bin/echo
import sys
if sys.argv[1:]:
    print(sys.argv[1], end='')
for arg in sys.argv[2:]:
    print('', arg, end='')
print()

source code for echo.2.py

# Python implementation of /bin/echo
import sys
print(' '.join(sys.argv[1:]))

source code for echo.3.py

# Python implementation of /bin/echo
import sys
print(*argv[1:])

source code for echo.4.py

Example - /bin/echo using just for

Example - /bin/echo - two other versions

Example - Summing Command-line Arguments

# sum integers supplied as command line arguments
# no check that arguments are integers
import sys
total = 0
for arg in sys.argv[1:]:
    total += int(arg)
print("Sum of the numbers is", total)

source code for sum_arguments.0.py
# sum integers supplied as command line arguments
import sys
total = 0
for arg in sys.argv[1:]:
    try:
        total += int(arg)
    except ValueError:
        print(f"error: '{arg}' is not an integer", file=sys.stderr)
        sys.exit(1)
print("Sum of the numbers is", total)

source code for sum_arguments.1.py

# Count the number of lines on standard input.
import sys
line_count = 0
for line in sys.stdin:
    line_count += 1
print(line_count, "lines")

source code for line_count.0.py

import sys
lines = sys.stdin.readlines()
line_count = len(lines)
print(line_count, "lines")

source code for line_count.1.py

import sys
lines = list(sys.stdin)
line_count = len(lines)
print(line_count, "lines")

source code for line_count.2.py
Opening Files

Similar to C, file objects can be created via the `open` function:

```python
file = open('data')
# read from file 'data'
file = open('data', 'r')
# read from file 'data'
file = open('results', 'w')
# write to file 'results'
file = open('stuff', 'ab')
# append binary data to file 'stuff'
```

Closing Files

File objects can be explicitly closed with `file.close()`

- All file objects closed on exit.
- Original file objects are not closed if opened again, can cause issues in long running programs.
- Data on output streams may be not written (buffered) until close - hence close ASAP.

Reading and Writing a File: Example

```python
file = open("a.txt", "r")
data = file.read()
file.close()

file = open("a.txt", "w")
file.write(data)
file.close()
```
## Exceptions

Opening a file may fail - always check for exceptions:

```python
try:
    file = open('data')
except OSError as e:
    print(e)
```

OSError is a group of errors that can be caused by syscalls, similar to errno in C.

Specific errors can be caught:

```python
try:
    file = open('data')
except PermissionError:
    # handle first error type
...
except PermissionError:
    # handle second error type
...
except IsADirectoryError:
    # handle third error type
...
```

## Context Managers

Closing files is annoying and error-prone. Python can do it for us with a context manager. The file will be closed when execution leaves the code block.

```python
sum = 0
with open("data", "r") as input_file:
    for line in input_file:
        try:
            sum += int(line.strip())
        except ValueError:
            pass
print(sum)
```

## Example - cp

# Simple cp implementation for text files using line-based I/O
# explicit close is used below, a with statement would be better
# no error handling

```python
import sys
if len(sys.argv) != 3:
    print("Usage: ", sys.argv[0], " <infile> <outfile>", file=sys.stderr)
    sys.exit(1)
infile = open(sys.argv[1], "r", encoding="utf-8")
outfile = open(sys.argv[2], "w", encoding="utf-8")
for line in infile:
    print(line, end='', file=outfile)
infile.close()
outfile.close()
```

source code for cp.py
# Simple cp implementation for text files using line-based I/O
# and with statement, but no error handling
import sys
if len(sys.argv) != 3:
    print("Usage:", sys.argv[0], "<infile> <outfile>", file=sys.stderr)
sys.exit(1)
with open(sys.argv[1]) as infile:
    with open(sys.argv[2], "w") as outfile:
        for line in infile:
            outfile.write(line)

source code for cp.1.py

# Simple cp implementation for text files using line-based I/O
# and with statement and error handling
import sys
if len(sys.argv) != 3:
    print("Usage:", sys.argv[0], "<infile> <outfile>", file=sys.stderr)
sys.exit(1)
try:
    with open(sys.argv[1]) as infile:
        with open(sys.argv[2], "w") as outfile:
            for line in infile:
                outfile.write(line)
except OSError as e:
    print(sys.argv[0], "error:", e, file=sys.stderr)
sys.exit(1)

source code for cp.2.py

# Simple cp implementation for text files using line-based I/O
# reading all lines into array (not advisable for large files)
import sys
if len(sys.argv) != 3:
    print("Usage:", sys.argv[0], "<infile> <outfile>", file=sys.stderr)
sys.exit(1)
try:
    with open(sys.argv[1]) as infile:
        with open(sys.argv[2], "w") as outfile:
            lines = infile.readlines()
            outfile.writelines(lines)
except OSError as e:
    print(sys.argv[0], "error:", e, file=sys.stderr)
sys.exit(1)

source code for cp.3.py
# Simple cp implementation using shutil.copyfile
import sys
from shutil import copyfile

if len(sys.argv) != 3:
    print("Usage:", sys.argv[0], "<infile> <outfile>", file=sys.stderr)
    sys.exit(1)

try:
    copyfile(sys.argv[1], sys.argv[2])
except OSError as e:
    print(sys.argv[0], "error:", e, file=sys.stderr)
    sys.exit(1)

source code for cp.4.py

# Simple cp implementation by running /bin/cp
import subprocess
import sys

if len(sys.argv) != 3:
    print("Usage:", sys.argv[0], "<infile> <outfile>", file=sys.stderr)
    sys.exit(1)

p = subprocess.run(['cp', sys.argv[1], sys.argv[2]])
sys.exit(p.returncode)

source code for cp.5.py

## UNIX-filter Behavior

treats all command-line arguments as file names
opens and reads from each of them in turn
no command line arguments, then fileinput == stdin
accepts -- as stdin
so this is cat in Python:

```bash
#!/usr/bin/env python3

import fileinput

for line in fileinput.input():
    print(line)
```

fileinput can be used to get UNIX-filter behavior.

```python
for line in fileinput.input():
    print(line)
```
Python Dicts

- Many languages have arrays accessed with small integer indexes.
  - Can be thought of as a mapping integer -> value
  - Python has lists (see widely used package numpy for arrays)
  - Easy to implement indexing
- Some languages have associative arrays - index doesn't have to be integer
  - Very useful, e.g. being able to use string as index
  - Harder to implement indexing
- Python has dicts - index can be almost any value
  - Index value cannot be mutable, e.g. can not be list or dict
  - Can be thought of as a mapping integer -> value

Example - Remembering Snap - Dict

```python
# Check if we've seen a line read from stdin, # using a dict.
# Print snap! if a line has been seen previously
# Exit if an empty line is entered
line_count = {}

while True:
    try:
        line = input("Enter line: ")
    except EOFError:
        break
    if line in line_count:
        print("Snap!")
    else:
        line_count[line] = 1
```

Source code for snap_memory.0.py

Example - Remembering Snap - Set

```python
# Check if we've seen lines read from stdin, 
# using a set.
# Print snap! if a line has been seen previously.
# Exit if an empty line is entered
lines_seen = set()

while True:
    try:
        line = input("Enter line: ")
    except EOFError:
        break
    if line in lines_seen:
        print("Snap!")
    else:
        lines_seen.add(line)
```

Source code for snap_memory.1.py
Some Useful Python Dict Operations
These can be applied to lists, tuples and ranges

- `d[key]` Return the item of d with key `key`
- `del d[key]` Remove d[key] from d. Raises a KeyError if key is not in the map.
- `key in d` Return True if d has a key key, else False.
- `key not in d` Equivalent to not key in d.
- `keys()` Return a new view of the dictionary's keys
- `items()` Return a new view of the dictionary's items
- `get(key[, default])` Return the value for key if key is in the dictionary, else default
- `values()` Return a new view of the dictionary's values.
- `update(other)` Update the dictionary with the key/value pairs from other
- `setdefault(key[, default])` If key is in the dictionary, return its value. If not, insert and return default.
- `clear()` Remove all items from the dictionary.
- `copy()` Return a shallow copy of the dictionary.

Running External Programs with subprocess
Python requires you to import the subprocess module to run external programs.

- `subprocess.run()` is usually the function used to run external programs.
- `subprocess.Popen()` can be used if lower level control is necessary.

```python
>>> subprocess.run(['date', '--utc'])
Tue 05 Aug 1997 01:11:01 UTC
CompletedProcess(args=['date', '--utc'], returncode=0)
```

By default stdout/stderr from the program goes directly to Python's stdout/stderr.
By default stdin from the program comes directly from Python's stdin.

Capturing the output from an External Programs with subprocess
To capture the output from commands:

```python
>>> p = subprocess.run(['date'], capture_output=True, text=True)
'Mon 18 Jul 2022 10:27:28 AEST\n'
>>> p.returncode
0
>>> q = subprocess.run(['ls', 'no-existent-file'], capture_output=True, text=True)
'ls: cannot access 'no-existent-file': No such file or directory\n'
>>> q.returncode
2
```
- captured output is a byte sequence (binary) by default.
- the option text=True converts it to a string
  - we want this 90% of the time
  - assumes the binary is utf-8 (if that is the local encoding)
Passing input to an External Programs with subprocess

To send input to a program:

```python
>>> message = "I love COMP(2041|9044)\n"
>>> p = subprocess.run(["tr", "a-z", "A-Z"], input=message, capture_output=True, text=True)
>>> p.stdout
'I LOVE COMP(2041|9044)\n'
>>> # note, you don't need an external program for this
>>> message.upper()
'I LOVE COMP(2041|9044)\n'
```

---

Example - Using Subprocess to Capture

```python
import subprocess
p = subprocess.run(["date"], capture_output=True, text=True)
if p.returncode != 0:
    print(p.stderr)
    exit(1)
weekday, day, month, year, time, timezone = p.stdout.split()
print(f"{year} {month} {day}")
```

Source code for parse_date.py

---

Python and External Commands

Optionally subprocess can pass the command to a shell to evaluate, e.g.:

```python
>>> subprocess.run("sort *.csv | cut -d, -f1,7 >output.txt", shell=True)
```

This conveniently allows use of shell features including pipes, I/O re-direction, globbing ...

Beware, this can also produce unexpected behaviour, e.g. if a Shell metacharacter appears in a filename.

Beware, this a common source of security vulnerabilities. It should be avoided when security is important.
Serving Web Pages with Python

Python includes a http server - easy to use for development/testing.

```python
>>> server_address = ('', 2041)
>>> handler = http.server.SimpleHTTPRequestHandler
>>> with http.server.HTTPServer(server_address, handler) as h:
...    h.serve_forever()
```

And there is a convenient command-line short cut:

```sh
$ echo hello from httpd >hello.txt
$ python3 -m http.server 2041
Serving HTTP on 0.0.0.0 port 2041 (http://0.0.0.0:2041/) ...
127.0.0.1 - - [17/Jul/2023 10:19:00] "GET /hello.txt HTTP/1.1" 200 -
in another terminal
$ curl -s http://0.0.0.0:2041/hello.txt
hello from httpd
```

Example - Using Subprocess to Capture Curl Output

```python
# Repeatedly download a specified web page
# until a specified regexp matches its source
# then notify the specified email address.
# implemented using subprocess
import re
import subprocess
import sys
import time
REPEAT_SECONDS = 300  # check every 5 minutes
if len(sys.argv) == 4:
    url = sys.argv[1]
    regexp = sys.argv[2]
    email_address = sys.argv[3]
else:
    print(f"Usage: {sys.argv[0]} <url> <regex> <email-address>", file=sys.stderr)
    sys.exit(1)

while True:
    p = subprocess.run(
        ['curl', '--silent', url], text=True, capture_output=True
    )
    webpage = p.stdout
    if not re.search(regexp, webpage):
        time.sleep(REPEAT_SECONDS)
        continue
    mail_body = f"Generated by {sys.argv[0]}"
    subject = f"website '{url}' now matches regex '{regexp}'"
    # the echo is for testing, remove to really send email
    subprocess.run(['echo', "mail", "-s", subject], text=True, input=mail_body)
    sys.exit(0)
```

source code for watch_website.0.py
### While True
```python
while True:
    response = urllib.request.urlopen(url)
    webpage = response.read().decode()
    if not re.search(regexp, webpage):
        time.sleep(REPEAT_SECONDS)
        continue
    mail_body = f"Generated by {sys.argv[0]}"
    subject = f"website '{url}' now matches regex '{regexp}'"
    # the echo is for testing, remove to really send email
    subprocess.run(['echo', 'mail', '-s', subject, text=True, input=mail_body])
sys.exit(0)
```

### Source code for `watch_website.py`

### Example - Using Beautiful Soup
```python
import bs4 as BeautifulSoup
IGNORE_WEBPAGE_ELEMENTS = set("[document] head meta style script title".split())
for url in sys.argv[1:]:
    response = urllib.request.urlopen(url)
    webpage = response.read().decode()
    soup = BeautifulSoup.BeautifulSoup(webpage, "lxml")
    for element in soup.findAll(text=True):
        parent = element.parent.name.lower()
        if parent in IGNORE_WEBPAGE_ELEMENTS:
            continue
        text = element.getText()
        # remove empty lines and leading whitespace
        text = re.sub(r"\n\s+", "\n", element)
        text = text.strip()
        if text:
            print(text)
```

### Source code for `fetch_website_text.py`

### Example - File Operations
```python
# Change the names of the specified files to lower case.
# (simple version of the Perl utility rename)
import os
import sys
for old_pathname in sys.argv[1:] :
    new_pathname = old_pathname.lower()
    if new_pathname == old_pathname:
        continue
    if os.path.exists(new_pathname):
        print(f"{sys.argv[0]}: '{new_pathname}' exists", file=sys.stderr)
        continue
    try:
        os.rename(old_pathname, new_pathname)
    except OSError as e:
        print(f"{sys.argv[0]}: '{new_pathname}' {e}", file=sys.stderr)
```

### Source code for `rename_lower_case.py`
Type hints

- Python doesn't enforce types even when they are given, thus they are hints
- Static type checkers are common that do enforce types as much as possible
- For best results type enforcement should be including in your code
- Type hints help you and others read your code and are highly recommended

```python
from typing import Optional, Union

da = 5
b = "Hello World"
# a type hint
c: int = 6
# but not enforced
d: int = "this isn't an int"
# composition of types
e: list[int] = [1, 2, 3, 4, 5]
# more composition of types
f: dict[int, list[tuple[str, str]]] = {1: [('a', 'b'), ('a', 'c')], 3: [('c', 's'), ('c', 'g')]}
```

```python
from typing import Optional, Union

g: Optional[float] = None
# `Union` allows for two or more types
h: Union[int, float] = 4
# type hints can also be used on function arguments and return values
def func(a: int, b: str='Hi
')->int:
    return len(b*a)
# type hints can be used on variables used in loops, tuple unpacking, or assignment can be pre-hinted
j: int
for j in range(0, 100):
    pass
k: bool
if k :=validate(data):
    pass
l: bool
m: int
n: str
l, m, n = (True, 99, "Apple")

# a variables type can be changed by first deleting it then redefining it
o: int = 0
del o
o: str = ""
Types

>>> type(())
tuple
>>> type((1))
int
>>> type((1,))
tuple
>>> type((1, 2, 3))
tuple
>>> type(('a', 'b', 'c',))
tuple
>>> type(tuple())
tuple
# same value as {}
>>> type({})
dict
# ??
>>> type({1})
set
>>> type((1,))
set
>>> type((1,))
set
>>> type({1, 2, 3})
set
>>> type(('a', 'b', 'c',))
set
>>> type(set())
set
>>> type({})
set
# same value as {}
>>> type(dict())
dict
# ??
>>> type((1, 2, 3))
dict
>>> type('a': 1)
set
>>> type('a': 'b': 2, 'c': 3,)
dict
>>> type(dict())
dict
# same value as {}