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)
<type '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'>
```
>>> type("Hello")
str
>>> type('Hello')
str
>>> type("""Hello"""")
str
>>> type('''Hello''')
str
>>> type(str())
str # same value as "" (empty string)
>>> type(1)
int
>>> type(int())
int # same value as 0
>>> type(4.4)
float
>>> type(float())
float # same value as 0.0

>>> type(5j)
complex
>>> type(3 + 1j)
complex
>>> type(complex())
complex # same value as 0j (and 0+0j)
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
>>> l = [1, 2, 3, 4, 5]
>>> t = (1, 2, 3, 4, 5)
>>> r = range(1, 6)
>>> l[2]
3
>>> t[2]
3
>>> r[2]
3
>>> l[2] = 42
>>> l
[1, 2, 42, 4, 5]
>>> t[2] = 42
Traceback (most recent call last):
  File "<stdin>"", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
Some Useful Python Sequence Operations

These can be applied to lists, tuples and ranges

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x in s</code></td>
<td>True if an item of s is equal to x</td>
</tr>
<tr>
<td><code>x not in s</code></td>
<td>False if an item of s is equal to x</td>
</tr>
<tr>
<td><code>s + t</code></td>
<td>the concatenation of s and t, also <code>s += t</code></td>
</tr>
<tr>
<td><code>s * n</code></td>
<td>equivalent to adding s to itself n times, also <code>s *= n</code></td>
</tr>
<tr>
<td><code>s[i]</code></td>
<td>ith item of s</td>
</tr>
<tr>
<td><code>s[i:j]</code></td>
<td>slice of s from i to j</td>
</tr>
<tr>
<td><code>s[i:j:k]</code></td>
<td>slice of s from i to j with step k</td>
</tr>
<tr>
<td><code>len(s)</code></td>
<td>length of s</td>
</tr>
<tr>
<td><code>min(s)</code></td>
<td>smallest item of s</td>
</tr>
<tr>
<td><code>max(s)</code></td>
<td>largest item of s</td>
</tr>
<tr>
<td><code>s.index(x[, i[, j]])</code></td>
<td>index of the first occurrence of x in s (at or after index i and before index j)</td>
</tr>
<tr>
<td><code>s.count(x)</code></td>
<td>total number of occurrences of x in s</td>
</tr>
</tbody>
</table>
Some Useful Python Mutable Sequence Operations

These can be applied to lists, not tuples or ranges

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s[i] = x</code></td>
<td>item i of s is replaced by x</td>
</tr>
<tr>
<td><code>s[i:j] = t</code></td>
<td>slice of s from i to j is replaced by elements of t</td>
</tr>
<tr>
<td><code>del s[i:j]</code></td>
<td>same as <code>s[i:j] = []</code></td>
</tr>
<tr>
<td><code>s[i:j:k] = t</code></td>
<td>the elements of <code>s[i:j:k]</code> are replaced by those of t</td>
</tr>
<tr>
<td><code>del s[i:j:k]</code></td>
<td>removes the elements of <code>s[i:j:k]</code> from the list</td>
</tr>
<tr>
<td><code>s.append(x)</code></td>
<td>appends x to the end of the sequence</td>
</tr>
<tr>
<td><code>s.clear()</code></td>
<td>removes all items from s</td>
</tr>
<tr>
<td><code>s.copy()</code></td>
<td>creates a shallow copy of s</td>
</tr>
<tr>
<td><code>s.insert(i, x)</code></td>
<td>inserts x into s at the index given by i</td>
</tr>
<tr>
<td><code>s.pop()</code> or <code>s.pop(i)</code></td>
<td>retrieves the item at i and also removes it from s</td>
</tr>
<tr>
<td><code>s.remove(x)</code></td>
<td>remove the first item from s where <code>s[i]</code> is equal to x</td>
</tr>
<tr>
<td><code>s.reverse()</code></td>
<td>reverses the items of s in place</td>
</tr>
<tr>
<td><code>s.sort()</code></td>
<td>sort the items of s in place</td>
</tr>
</tbody>
</table>
Ranges

```python
>>> 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))
[]
```
Even More Types

```python
>>> type([])
list
>>> type([1])
list
>>> type([1,])
list
>>> type([1, 2, 3])
list
>>> type(['a', 'b', 'c',])
list
>>> type(list())
list # same value as []
>>> type(())
tuple
>>> type((1))
int # ??
>>> type((1,))
tuple
>>> type((1, 2, 3))
tuple
>>> type({})
dict
>>> type({1})
set
>>> type({1,})
set
>>> type({1, 2, 3})
set
>>> type({'a', 'b', 'c',})
set
>>> type(set())
set
>>> type({'a': 1})
dict
>>> type({'a': 1, 'b': 2, 'c': 3,})
dict
>>> type(dict())
dict # same value as {}
```
# 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()
# 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()
Example - /bin/echo using just for

```python
# 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
Example - /bin/echo - two other versions

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

source code for echo.3.py

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

source code for echo.4.py
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

https://www.cse.unsw.edu.au/~cs2041/24T1/
### Example - Summing Command-line Arguments with Checking

```python
# 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_arguments1.py](https://www.cse.unsw.edu.au/~cs2041/24T1/)
# Count the number of lines on standard input.

```python
import sys
line_count = 0
for line in sys.stdin:
    line_count += 1
print(line_count, "lines")
```

[source code for line_count.0.py](https://www.cse.unsw.edu.au/~cs2041/24T1/COMP(2041|9044) 24T1 — More on Python](https://www.cse.unsw.edu.au/~cs2041/24T1/COMP(2041|9044) 24T1 — More on Python)
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
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.
file = open("a.txt", "r")
data = file.read()
file.close()

file = open("a.txt", "w")
file.write(data)
file.close()
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 cased by syscalls, similar to errno in C

Specific errors can be caught

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

```
sum = 0
with open("data", "r") as input_file:
    for line in input_file:
        try:
            sum += int(line.strip())
        except ValueError:
            pass
print(sum)
```
# 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
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()
# 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)
# Simple cp implementation for text files using line-based I/O
# and with statement and error handling

```python
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)

https://www.cse.unsw.edu.au/~cs2041/24T1/COMP(2041|9044) 24T1 — More on Python
# Simple cp implementation for text files using line-based I/O
# reading all lines into array (not advisable for large files)

```python
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.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)
# Simple cp implementation by running /bin/cp

```python
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
`fileinput` can be used to get 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:

```python
#!/usr/bin/env python3
import fileinput
for line in fileinput.input():
    print(line)
```
Python Dicts

- many languages have arrays accessed with small integer indexes.
  - can be though 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 can not be mutable, e.g. can not be list or dict
  - can be though of as a mapping integer -> value
# 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_memory0.py
# 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_memory1.py
Some Useful Python Dict Operations

These can be applied to lists, tuples and ranges

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>d[key]</code></td>
<td>Return the item of <code>d</code> with key <code>key</code></td>
</tr>
<tr>
<td><code>del d[key]</code></td>
<td>Remove <code>d[key]</code> from <code>d</code>. Raises a KeyError if <code>key</code> is not in the map.</td>
</tr>
<tr>
<td><code>key in d</code></td>
<td>Return True if <code>d</code> has a key <code>key</code>, else False.</td>
</tr>
<tr>
<td><code>key not in d</code></td>
<td>Equivalent to not <code>key in d</code>.</td>
</tr>
<tr>
<td><code>keys()</code></td>
<td>Return a new view of the dictionary’s keys</td>
</tr>
<tr>
<td><code>items()</code></td>
<td>Return a new view of the dictionary’s items</td>
</tr>
<tr>
<td><code>get(key[, default])</code></td>
<td>Return the value for <code>key</code> if <code>key</code> is in the dictionary, else default</td>
</tr>
<tr>
<td><code>values()</code></td>
<td>Return a new view of the dictionary’s values.</td>
</tr>
<tr>
<td><code>update([other])</code></td>
<td>Update the dictionary with the key/value pairs from <code>other</code></td>
</tr>
<tr>
<td><code>setdefault(key[, default])</code></td>
<td>If <code>key</code> is in the dictionary, return its value. If not, insert and return default.</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>Remove all items from the dictionary.</td>
</tr>
<tr>
<td><code>copy()</code></td>
<td>Return a shallow copy of the dictionary.</td>
</tr>
</tbody>
</table>
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)
>>> p.stdout
'Mon 18 Jul 2022 10:27:28 AEST\n'
>>> p.returncode
0
>>> q = subprocess.run(['ls', 'no-existent-file'], capture_output=True, text=True)
>>> q.stderr
'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 time
  - assumes the binary is utf-8 (if that is the local encoding)
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,
                    >>> 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'
```
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
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.
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:

```bash
$ 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

```bash
$ curl -s http://0.0.0.0:2041/hello.txt
hello from httpd
```
Example - Using Subprocess to Capture Curl Output

# 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)
```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_website1.py
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, "html5lib")

    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)
# 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)
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

a = 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')]}  
```
from typing import Optional, Union

# `Optional` allows for None values
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)

# for variables used in loops, tuple unpacking, or assignment can be pre-hinted
# pre-hinting does not define the variable as it has not assigned a value and python variables must always be initialised
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 = ""

https://www.cse.unsw.edu.au/~cs2041/24T1/