Names and Types

- Python associates types with values.
  - languages like C, Perl associate types with variables
- A Python variables 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.

```
>>> 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'>
```

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

Python Sequences - Examples

```
>>> 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
```

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]	ith 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

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:j]	same as s[i:j] = []
s[i:j:k] = t	the elements of s[i:j:k] 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 += 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() or s.pop(i)	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()
```

source code for echo.0.py

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()
```

source code for echo.1.py

Example - /bin/echo using just for

```
# 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 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 - 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

Example - Summing Command-line Arguments with Checking

```
# 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

Example - Counting Lines on stdin

```
# 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

Example - Counting Lines on stdin - two more versions

```
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:

```
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

```
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:

```
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

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

Context Managers

Closing files is annoying python can do it for us with a context manager The file will be closed for us when we exit 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)
```

Example - cp

```python
# 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()
```

source code for cp.0.py

Example - cp

```python
# 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

Example - cp

```python
# 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

Example - cp

```
# 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

Example - cp

```python
# 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

Example - cp

```
# 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

`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:

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

import fileinput

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

Python and External Commands

Python requires you to import the `subprocess` module to run external commands.

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

Python and External Commands

`subprocess.run()` can:

run one command:

```
subprocess.run(["ls", "-Al"])
```

run one command line:

```
subprocess.run("cat data.csv | head -n5 | cut -d, -f1,7", shell=True)
```

The output is sent to `stdout` by default

Python and External Commands

To capture the output from commands:

```
proc = subprocess.run(["date"], capture_output=True, text=True)

output = proc.stdout
errors = proc.stderr
```

The output is a byte sequence (binary) by default. `text=True` gives us a Unicode string.

Python and External Commands

To send input to the command:

```
text = "hello world"
subprocess.run(["tr", "a-z", "A-Z"], input=text, text=True)
```

Python and External Commands

External command examples:

```
import subprocess

proc = subprocess.run(["date"], capture_output=True, text=True)

if proc.returncode:
  print(proc.stderr)
  exit(1)

line: str
for line in proc.stdout.splitlines():
  weekday: st; month: str; day: str time: str tz; year: str
  weekday, month, day, time, tz, year = line.split()
  print(f"{year} {month} {day} - {time} {tz}")
```

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

```
from typing import Optional, Union

a = 5
b = "Hello World"
c: int = 6 # a type hint
d: int = "this isn't an int" # but not enforced
e: list[int] = [1, 2, 3, 4, 5] # composition of types
f: dict[int, list[tuple[str, str]]] = {1: [('a', 'b'), ('a', 'c')], 3: [('c', 's'
g: Optional[float] = None # `Optional` allows for None values
h: Union[int, float] = 4 # `Union` allows for two or more types

# type hints can also be used on function arguments and return values
def func(a: int, b: str = 'Hi\n') -> int:
```

Types

```
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)

type([]) # list
type([1]) # list
type([1,]) # list
```

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, 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 {}
```