Python: miscellaneous topics

Intro. to computing

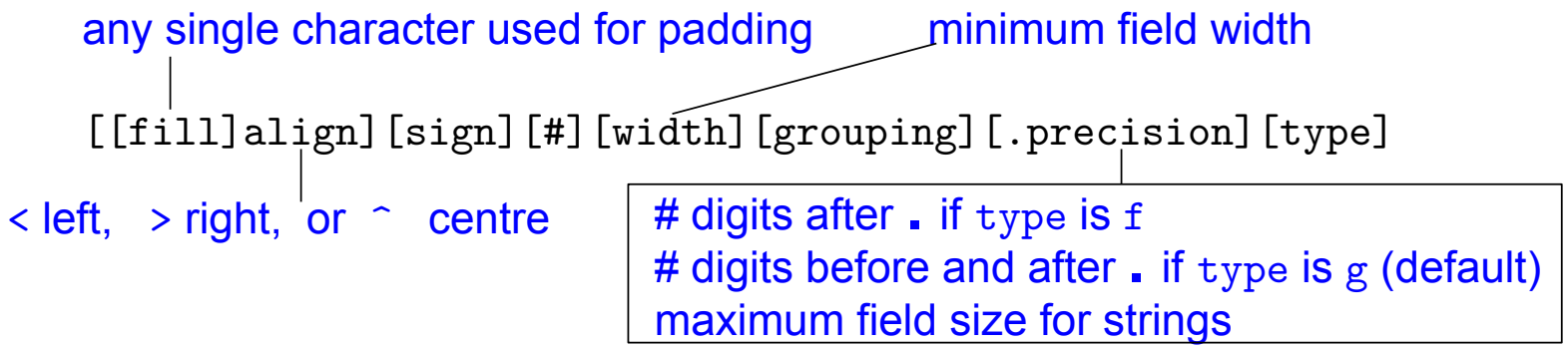
`https://mandarm.github.io/computing.html`

Topics

- Formatted strings
- Measuring time taken
- Simple file input/output
- Pickling
- Command line arguments

f-strings I

- Prefix string with `f` or `F`.
- Enclose variables / expressions in braces `{ }`.
  - expressions evaluated from left to right wherever the formatted string literal appears
  - use `{{` or `}}` to print actual curly bracket
- Add `:format-string` after expression to customise output.
- Format of `format-string`(*):



* For more details, see
http://docs.python.org/3/library/string.html#format-specification-mini-language

http://learnpython.com/blog/python-string-formatting/ (simpler)

f-strings II

Examples:

```
x = 2026
y = 6147
print(f"""Value of x is {x}
Value of y is {y}""")
print(f'Sum = {x+y}, product = {x*y}')

s = "Hello"
pie = 3.1415296
field_width = 10
print(f"""Print s, field width = 20, left aligned, padded by !:
{s.lower():!<20}
Print s centred, field width specified by a variable:
{s.lower():-^{field_width}}
The value of pi, printed right aligned in a field of width 10,
    correct to 3 decimal places is {pie:>10.3f}.
Also see {pie:>10.3g}, {pie:>10.3}, {100*pie:>10.3g}, {100*pie:>10.3}.
""")
```

f-strings III

Measuring time

Measuring time: using `time` I

`import time` OR `from time import *`

`time.time()` **AVOID**

usually 01.01.1970

- return time in **seconds** since *epoch* as floating point number
- underlying system may not measure time with better precision than 1 second
  $\Rightarrow$ fractional part of returned time not reliable

`time.perf_counter()`, `time.perf_counter_ns()` (`ns` $\equiv$ nanoseconds)

- as above, but uses system clock with highest available resolution
- appropriate for measuring short durations
- measures *elapsed time* or *wall-clock time*
  $\Rightarrow$ dependent on system load

`time.process_time()`, `time.process_time_ns()` ✓

- as above, but measures time taken by current process
  $\Rightarrow$ independent of system load

Measuring time: using `time` II

Example:

```
from time import *

start_time = time() # try perf_counter(), process_time()
L = [ i**3 for i in range(100000) ]
sleep(2)            # does this affect the measured time?
end_time = time()
print("Time taken: %.4f seconds\n" % (end_time - start_time))
```

Measuring time: using `timeit` I

`import timeit` OR `from timeit import *`

```
timeit(stmt='pass', setup='pass', timer=time.perf_counter,
number=1000000, globals=None)
```

- returns time in seconds required to run `stmt` $\times$ `number` times
- `setup` : initialisation code (if any) to be executed (not included in measured time)
- `timer` : which system timer to use
- `globals` : specifies *namespace* (global variables, function names, etc.)
- all arguments are optional

```
repeat(stmt='pass', setup='pass', timer=default, repeat=5,
number=1000000, globals=None)
```

- as above, returns a list containing `repeat` timing values

Measuring time: using `timeit` II

Examples

```
def f(x): return x*x
def g(x): return x**2
def h(x): return x**8

timeit(number=100)
timeit(number=100000000)
for func in (f,g,h):
    timeit('func(42)', globals=globals()) # try timer=process_time}
```

https://docs.python.org/3/library/timeit.html

It's tempting to calculate mean and standard deviation from the result vector and report these. However, this is not very useful. In a typical case, the lowest value gives a lower bound for how fast your machine can run the given code snippet; higher values in the result vector are typically not caused by variability in Python's speed, but by other processes interfering with your timing accuracy. So the `min()` of the result is probably the only number you should be interested in. After that, you should look at the entire vector and apply common sense rather than statistics.

Measuring time: using `timeit` from command line

```
python -m timeit [-n N] [-r N] [-u U] [-s S] [-h] [statement ...]
```

- `n` : how many times to execute `statement`
- `r` : how many times to repeat (default 5)
- `s` : initialisation
- `p` : use `time.process_time()` instead of `time.perf_counter()`
  (i.e., measure process time, not wallclock time)
- `u` : unit to be used (nsec, usec, msec, or sec)

Example:

```
python3 -mtimeit -s'import your_filename' 'your_filename.f()'
```

File input/output

Opening / closing a file I

Opening/closing a file

```
f = open("file.txt", "w")
print("Some string", file=f)
f.close()

# RECOMMENDED
with open('filename.txt') as f:
     do_something()

# Line by line handling
with open('filename.txt') as f:
    i = 1
    for line in f:
        print(f'Line {i}: {line}')
    i += 1
```

Opening / closing a file II

Opening modes:

- `r`, `w`, `a`, `x`: read, write, append, create
- `r+`, `w+`, `a+`: read and write, append and write
- `rb`, `rb+`, `wb`, `wb+`, `ab`, `ab+`: as above for binary files

Difference between `r+` and `w+`

- `r+` preserves existing content;
  `w+` truncates it immediately.
- `r+`: file must exist
  `w+`: file created if necessary (need not exist)
- Initial position for both is at the beginning of the file

Thumbrule: use `r+` to (partially) modify existing file; use `w+` if you need to read back what was written.

Reading / writing

Assumption: `f` handle for file opened in appropriate mode

```
# Read size bytes, or entire file if size is omitted
input = f.read(size)

# Read + return the next line
inputline = f.readline()

# Read + return next n (or all, if n omitted) lines as list
inputlines = f.readlines(n)

print('test', file=f)

s = ["namashkar", "vanakkam", "hello", "hi", "aloha"]
f.write(s[0])
f.write(s[1] + '\n')                # include \n if you need it
f.writelines(s)                     # l is a list of strings
f.writelines('\n'.join(s) + '\n') # include \n if you need it
```

Homework: other functions

```
f.flush()
f.tell()
f.seek()
```

Saving / loading data using **pickle**

Reference: `https://docs.python.org/3/library/pickle.html`

- `pickle.dump(obj, file, protocol=None, *, fix_imports=True)`
  - write 'pickled' representation of `obj` to `file`
  - `file` must correspond to a file opened in binary mode
- `pickle.load(file, *, fix_imports=True, encoding="ASCII", errors="strict")`
  - read 'pickled' object representation from `file` and return the reconstituted object hierarchy specified therein
  - `file` must correspond to a file opened in binary mode

Simple command line arguments

```
import sys
print("argc = ", len(sys.argv))
for i, a in enumerate(sys.argv) :
    print(str(i) + "-th argument:", a)
```

For more advanced command line argument processing, see the `argparse` library

Some useful modules

itertools

Reference: https://docs.python.org/3/library/itertools.html

```
import itertools
# OR import itertools as it
# OR from itertools import *


# return r length subsequences of elements from the input iterable
# combinations(list, 2) is particularly useful
itertools.combinations(iterable, r)
itertools.permutations(iterable, r)
itertools.combinations_with_replacement('ABCD', 2)


# cartesian product of input iterables
itertools.product(A, B, C)      # A x B x C
itertools.product(A, repeat=2) # A x A x A
```

copy

Reference: `https://docs.python.org/3/library/copy.html`

From the documentation:

- Assignment statements in Python do not copy objects, they create bindings between a target and an object. For collections that are mutable or contain mutable items, a copy is sometimes needed so one can change one copy without changing the other. This module provides generic shallow and deep copy operations (explained below).
- A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.
- A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

- `copy.copy(x)` : return a shallow copy of `x`
- `copy.deepcopy(x)` : return a deep copy of `x`