Computing Laboratory

Basics of Python - II

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

December, 2020

Standard Input/Output functions

I/O from the terminal:

- `print()` # Value can be printed without mentioning the type
- `input()` # Value is taken in a string and can be converted to appropriate type using int(), float(), bool(), etc.

I/O from files:

- `open()`, `close()` # Files are opened in r/w/a mode and the address is returned to a file pointer
- `read()`, `write()` # With a file pointer
- `readline()` # With a file pointer
- `readlines()`, `writelines()` # With a file pointer

Standard Input/Output functions

`end` **delimiter in** `print`**:**

```
print("Python", end = ";") # It can end with any string
print("Language", end = ";") # The default one is '\n'
```

Output: Python;Language;

Standard Input/Output functions

`end` delimiter in `print`:

```
print("Python", end = ";") # It can end with any string
print("Language", end = ";") # The default one is '\n'
```

Output: Python;Language;

More about `print`:

```
a = 7
print(a, 'is prime') # ',' includes a space in between
b = 'prime'
print('7 is ' + b) # '+' works only on strings
```

Output: 7 is prime
7 is prime

Standard Input/Output functions

Reading data from file:

```
def read(file):
    f = open(file, 'r')
    output = f.read()
    f.close()
    return output
output = read('Data.txt')
```

Standard Input/Output functions

Reading data from file:

```
def read(file):
    f = open(file, 'r')
    output = f.read()
    f.close()
    return output
output = read('Data.txt')
```

Reading data from file (alternative approach):

```
with open('Data.txt', 'r') as f: output = f.read();
```

Special Input/Output functions

Reading data from a CSV file:

```
import pandas as pd # Import pandas
pd.read_csv("file.csv") # reading CSV file (same path)
```

Special Input/Output functions

Reading data from a CSV file:

```
import pandas as pd # Import pandas
pd.read_csv("file.csv") # reading CSV file (same path)
```

Reading data from an XLS file:

```
import pandas as pd # Import pandas
pd.read_excel("file.xls") # reading XLS file (same path)
```

Functions

```
def <function-name>(<argument 1>, ..., <argument n>):
    Statement 1
    Statement 2
    Statement 3
```

Functions

```
def <function-name>(<argument 1>, ..., <argument n>):
    Statement 1
    Statement 2
    Statement 3
```

Note: You may either return a `<value>` or return nothing based on your requirement.

Finding prefixes of a string

```
def prefix(str):
    start, end = 0, 0
    while start < len(str):
        if str[end] == str[end-start]:
            print(str[start:end + 1], end = " ")
            end += 1
            if end == len(str):
                start += 1
                end = start
        else:
            start += 1
            end = start # Index of substring
prefix("Python")
```

Depth-First Search

The DFS algorithm works as follows:

1. Keep any one of the graph's vertices on top of a stack.
2. Pop out the top data item from the stack and add it to the `Visited` list.
3. Create a list of that vertex's adjacent nodes. Add the ones which are not in the `Visited` list to the top of stack.
4. Repeat steps 2-3 until the stack is empty.

Depth-First Search

```
# Adjacency list defined as a dictionary
Graph = {
    '0' : ['1','2'], '1' : ['3', '4'], '2' : ['5'],
    '3' : [], '4' : ['5'], '5' : []
}
Visited = [] # List to keep track of visited vertices
def DFS(Visited, Graph, Vertex):
    if Vertex not in Visited:
        Visited.append(Vertex)
        print(Visited[len(Visited) - 1]) # Top of stack
        for Adjacent in Graph[Vertex]:
            DFS(Visited, Graph, Adjacent)
DFS(Visited, Graph, '0') # Function call with vertex '0'
```

Lambda functions

Python provides an anonymous function `lambda` that can take any number of arguments, but can only have one expression.

```
x = lambda a: a + 10
print(x(5))
```

Here, the output will be 15.

Late binding closures

Python's closures are late binding. Hence, the values of variables used in closures are looked up when the inner function is called.

```
def increase():
    return [lambda x : i + x for i in range(5)]
for add in increase():
    print(add(10))
```

Here, whenever any of the returned functions are called, the value of i is looked up in the surrounding scope at call time. By then, the loop has completed and i is left with its final value of 4. So, it will print {14, 14, 14, 14, 14} instead of {10, 11, 12, 13, 14}.

Importing modules and functions

Importing a module:

```
import <module_name>
```

OR

```
import <module_name> as <custom_name>
```

Importing modules and functions

Importing a module:

```
import <module_name>
```

OR

```
import <module_name> as <custom_name>
```

Using a function:

```
import <module_name>
<module_name>.<function_name>()
```

OR

```
import <module_name> as <custom_name>
<custom_name>.<function_name>()
```

OR

```
from <module_name> import <function_name>
<function_name>()
```

Using built-in methods

There are some functions in Python that does not require to import a module because they work on specific data structures.

- Built-in methods for strings (e.g., `capitalize()`, `strip()`, `zfill()`, etc.)
- Built-in methods for lists/arrays (e.g., `sort()`, `reverse()`, `clear()`, etc.)
- Built-in methods for dictionaries (e.g., `keys()`, `values()`, `update()`, etc.)
- Built-in methods for tuples (e.g., `count()`)

Note: The `sort()` method in Python implements the hybrid algorithm *Timsort*, derived from merge sort and insertion sort.

Mathematical functions for real numbers

Using the `math` module:

```
import math
math.<function_name>()
```

Mathematical functions for real numbers

Using the `math` module:

```
import math
math.<function_name>()
```

ceil(x)	– Ceiling of x
comb(n, r)	– Number of ways to choose r from n (${}^nC_r$)
copysign(x, y)	– Float with magnitude of x but sign of y
fabs(x)	– Absolute value of x
factorial(x)	– Factorial of x
floor(x)	– Floor of x

Table: Functions in `math` module

Mathematical functions for real numbers

fmod(x, y)	– x % y (preferable for integers)
frexp(x)	– Mantissa and exponent of x as a pair (m, e)
fsum(iterable)	– Accurate floating point sum of values in iterable
gcd(x, y)	– GCD of the integers x and y
isclose(x, y, *, rel-tol, abs-tol)	– Whether x is close to y
isfinite(x)	– Whether x is finite (or ∞/NaN)
isinf(x)	– Whether x is infinite
isnan(x)	– Whether x is NaN ("not a number")
isqrt(x)	– Integer square root of x
ldexp(x, i)	– $x * 2^i$

Table: Functions in `math` module

Mathematical functions for real numbers

modf(x)	– Fractional and integer parts of x
perm(n, r=None)	– Number of ways to choose k from n without repetition (${}^nP_r$)
prod(iterable, *, start=1)	– Product of values in iterable
remainder(x, y)	– Remainder of x when divided by y
trunc(x)	– Value of x truncated to an integral
exp(x)	– e^x
expm1(x)	– $e^x - 1$ (provides a better precision)
log(x[, base])	– Natural logarithm of x (base e)
log1p(x)	– Natural logarithm of 1+x (base e)
log2(x)	– Base-2 logarithm of x
log10(x)	– Base-10 logarithm of x
pow(x, y)	– x^y

Table: Functions in `math` module

Mathematical functions for real numbers

sqrt(x)	– Square root of x
acos(x)	– Arc cosine of x (result in radians)
asin(x)	– Arc sine of x (result in radians)
atan(x)	– Arc tangent of x (result in radians)
atan2(x, y)	– Arc tangent of x/y (in radians)
cos(x)	– Cosine of x
dist(iterable1, iterable1)	– Euclidean distance between iterable 1 and iterable2
hypot(*coordinates)	– Euclidean norm sqrt(x*x + y*y) for the point (x, y), sqrt(sum(x**2 for x in coordinates))
sin(x)	– Sine of x
tan(x)	– Tangent of x

Table: Functions in `math` module

Mathematical functions for real numbers

degrees(x)	– Convert angle x from radians to degrees
radians(x)	– Convert angle x from degrees to radians.
acosh(x)	– Inverse hyperbolic cosine of x
asinh(x)	– Inverse hyperbolic sine of x
atanh(x)	– Inverse hyperbolic tangent of x
cosh(x)	– Hyperbolic cosine of x
sinh(x)	– Hyperbolic sine of x
tanh(x)	– Hyperbolic tangent of x
erf(x)	– Error function at x
erfc(x)	– Complementary error function at x
gamma(x)	– Gamma function at x
lgamma(x)	– Natural logarithm of absolute value of Gamma function at x

Table: Functions in `math` module

Mathematical functions for real numbers

pi	– $\pi = 3.14...$, to available precision
e	– $e = 2.71...$, to available precision
tau	– $\tau = 6.28...$, to available precision
inf	– Floating-point positive infinity
nan	– Floating-point NaN

Table: Functions in `math` module

Mathematical functions for complex numbers

Using the `cmath` module:

```
import cmath
cmath.<function_name>()
```

OR

```
from cmath import <function_name>
<function_name>()
```

Mathematical functions for complex numbers

Using the `cmath` module:

```
import cmath
cmath.<function_name>()
```

OR

```
from cmath import <function_name>
<function_name>()
```

phase(x)	– Phase of x (in float)
polar(x)	– Representation of x in polar coordinates as (r, phi)
rect(r, phi)	– Complex number x with polar coordinates r and phi
exp(x)	– e^x
log(x[, base])	– Natural logarithm of x (base e)
log10(x)	– Base-10 logarithm of x

Table: Functions in `cmath` module

Mathematical functions for complex numbers

sqrt(x)	– Square root of x
acos(x)	– Arc cosine of x
asin(x)	– Arc sine of x
atan(x)	– Arc tangent of x
cos(x)	– Cosine of x
sin(x)	– Sine of x
tan(x)	– Tangent of x
acosh(x)	– Inverse hyperbolic cosine of x
asinh(x)	– Inverse hyperbolic sine of x
atanh(x)	– Inverse hyperbolic tangent of x
cosh(x)	– Hyperbolic cosine of x
sinh(x)	– Hyperbolic sine of x
tanh(x)	– Hyperbolic tangent of x

Table: Functions in `cmath` module

Mathematical functions for complex numbers

isclose(x, y, *, rel-tol, abs-tol)	– Whether x is close to y
isfinite(x)	– Whether x is finite (or ∞/NaN)
isinf(x)	– Whether x is infinite
isnan(x)	– Whether x is NaN
pi	– $\pi = 3.14...$, to available precision
e	– $e = 2.71...$, to available precision
tau	– $\tau = 6.28...$, to available precision
inf	– Floating-point positive infinity
infj	– Complex number with zero real and positive infinity imaginary parts
nan	– Floating-point NaN
nanj	– Complex number with zero real part and NaN imaginary part

Table: Functions in `cmath` module

Statistical functions

Using the `statistics` module:

```
import statistics
statistics.<function_name>()
```

Statistical functions

Using the `statistics` module:

```
import statistics
statistics.<function_name>()
```

mean(X)	– Arithmetic mean of the data in X
fmean(X)	– Arithmetic mean of the data (converted to float) in X
geometric_mean(X)	– Geometric mean of the data (converted to float) in X
harmonic_mean(X)	– Harmonic mean of the data in X

Table: Functions in `statistics` module

Statistical functions

median(X)	– Median of the data in X
median_low(X)	– Low median of the data in X
median_high(X)	– High median of the data in X
median_grouped(X, interval)	– Median of the grouped data in X
mode(X)	– Most frequent data item in X
multimode(X)	– Most frequent data items in the order they appear in X

Table: Functions in `statistics` module

Statistical functions

pstdev(X, mu=None)	– Population standard deviation of the data in X
pvariance(X, mu=None)	– Population variance of the data in X
stdev(X, xbar=None)	– Sample standard deviation of the data in X
variance(X, xbar=None)	– Sample variance of the data in X
quantiles(X, *, n, method)	– Divide data in X into n continuous intervals with equal probability

Table: Functions in `statistics` module

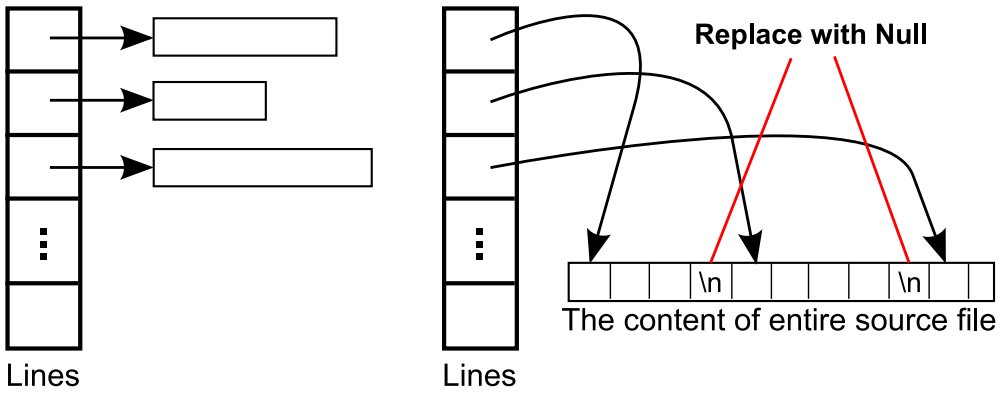
Problems – Day 3

1. Suppose a list of strings are given as user input. Write a program to verify whether the ordering of distinct characters in the last string preserves the ordering of characters in all the preceding strings. Note that, conflicts of ordering might exist in the preceding strings.
   Note: The inputs `apple, apollo, apl` preserves the ordering but not the inputs `capacity, pacific, cap`.
2. Suppose there are two separate files containing sufficiently large integer values. Write a program that will take those two filenames as user inputs and return the addition result.
   Note: An efficient implementation will not depend on the primary memory of the system.

Problems – Day 3

3. Consider 2 sets of integers, A and B, are stored in arrays. Write a program to find the number of (possibly overlapping) occurrences of the sequence B in A.

4. You have to write a program that reads its own source file (i.e., mtc20xx-day3-prog4.py), and prints the lines in that file in lexicographically sorted order.
   Note: An efficient implementation is highlighted below.



Naive approach Efficient approach