Python Programming

Expressions

Basic data types:

- **Boolean**: `bool`
- **Integer**: `int`
- **Floating point**: `float`
- **String**: `str`

Escape characters

- Newline character `\n` (i.e., Enter)
- Tab character `\t` (i.e., Tab)
- Single-quote `\'` (i.e., ')
- Double-quote `\"` (i.e., ")
- Backslash `\\`

`type()` function: check the type of a value

Conversion between data types

- Conversion to boolean: check if the value is empty or not (this depends on the type).
  E.g. an empty number is 0 and 0.0; an empty string is '' or ""
- Conversion to integer:
  True - 1, False - 0
  Truncate floats (remove numbers after decimal)

Arithmetic operations, PEMDAS

- Addition `+`
- Subtraction `-`
- Multiplication `*`
- Division `/`
- Integer division `//`
- Modulo `%`
- Exponentiation `**`

Floating points are only an approximation of real numbers. The most famous example is shown below: `0.1+0.2` returns `0.30000000000000004`

Relational operations

- Compare numbers ==, !=, <, <=, >, >=
- Compare string using ASCII table: numeric < uppercase < lowercase

Logical operations:

- Negation: `not`
- Conjunction: `and`
- Disjunction: `or`

String operations

- Substring check: `in`, `notin`
- Indexing: `s[<index>]`
  start from 0, negative indexing -1 is last letter
- Slicing:
  `s[start : stop]` (the string is from index `start` to index `stop` −1)
  `s[start : stop : step]`
  In particular, `s[::-1]` reverses `s`; `s[:-1:]` removes the last element.
  If `start` > `stop`, it is unreachable so output is empty string.
  If `stop` exceeds length of `s` (beyond the edge), the output is only until the end of `s`.
- String length: `len()`
- Concatenate strings: `+`
- Replicate string: `*`
- Uppercase: `upper()`
- Replace a segment of the string: `replace()`
- Find substring: `find()` (outputs index of first letter of sub-string)

Strings are *immutable*, e.g. cannot modify any letter.

Assignments

Variable naming rules

- Start with uppercase (`A-Z`), lowercase (`a-z`), or underscore (_).
- Only contain uppercase (`A-Z`), lowercase (`a-z`), numeric (`0-9`), or underscore (_).
- Cannot be one of the reserved keywords.

Use = for an assignment.

Input and output: `input()`, `print()`

Selection

If-statement:

```
if cond1:
    block1
elif cond2:
    block2
else:
    block3
```

Iteration

While-loop:

```
while cond:
    body
```

Complex Loop

Nested loop: a loop inside a loop

Use `break` to exit the nearest loop that it is inside.

Use `continue` to jump back to the beginning of the loop (i.e., the loop condition).

Function

Importing modules

- `import X`: use `X.name` to refer to objects in `X`
- `from X import *`: creates references to all public objects in `X`, can use plain name
- `from X import a, b, c`: creates references to specified objects, can now use `a`, `b`, `c` in your program

```
from math import *      # At beginning of code
```

Define **function**:

```
def <name>(<parameters>):
    <block>
    return ...
```

Debugging

- `IndexError`: when the wrong index of a list is retrieved.
- `ImportError`: an imported module is not found.
- `NameError`: the variable is not defined.
- `SyntaxError`, `IndentationError`
- `TypeError`: when a function and operation are applied in an incorrect type.
- `ZeroDivisionError`
- Infinite loop

Divide and Conquer

Divide-and-conquer

1. **Abstraction**: think in terms of high-level operations/procedures, instead of implementation details.
2. **Decomposition**: split a problem into smaller subproblems and solve them.
3. **Integration**: combine the solutions to the smaller subproblems to solve the original problem.

Recursion: solving a simpler self-similar subproblem (a function that calls itself)

1. Figure out the base case (typically $n = 0$ or $n = 1$)
2. Assume you know how to solve $n - 1$, how to solve for n?

```
# Factorial
def factorial(n):
    if n == 0:                    # base case
        return 1                  # base value
    else:                         # recursive case
        return n * factorial(n - 1) # recursion
```

Sequence

For-loop:

```
for var in seq:
    body
```

where `seq` is any sequence, e.g. string, list, range.

`range` contains a sequence of integers.

1. `range(stop)`
2. `range(start, stop)`
3. `range(start, stop, step)`

We exclude the value of `stop`.

Tuple: `(expr1, expr2, expr3)`

Tuple operations

- Indexing
- Length: `len()`
- Slicing
- Repetition `*`
- Concatenation `+`

Tuples are *immutable* (once the data is created, the content cannot be modified).

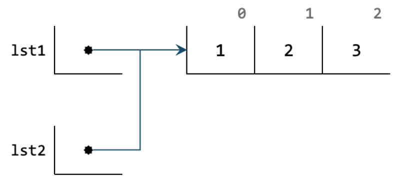
Mutability

List: `[expr1,expr2,expr3]`

List operations:

- Indexing
- Slicing
- Iteration
- Append `+` (cannot concatenate a list with a tuple or vice versa)
- Repetition `*`

Lists are *mutable*, but lists can be *aliased* (two different arrows pointing to the same location).



List methods:

- `lst.append(obj)`: Adds the element `obj` to the end of the list
- `lst.extend(seq)`: Adds the elements of the sequence `seq` to the end of `lst`
- `lst.remove(obj)`: Removes the first occurrence of `obj` from `lst`
- `lst.insert(idx, obj)`: Inserts `obj` into the list at index `idx`
- `lst.pop()`: Removes and returns the last element
- `lst.pop(idx)`: Removes and returns element at index `idx`
- `lst.copy()`: Returns a shallow copy of `lst`
- `lst.clear()`: Clears the list

List comprehension: `[F(x) for x in S if P(x)]`

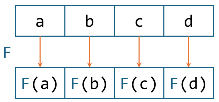
Working with Sequence

Lambda function (anonymous function):
`lambda par: expr`
where `par` can be empty.
We can assign name to lambda functions:

```
add = lambda x, y: x + y
```

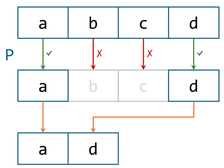
Limitation: only has a single return statement. Thus we use lambda to create *pure functions* (only depend on the input parameter, does not modify anything).

Map: `map(function, iterable)`



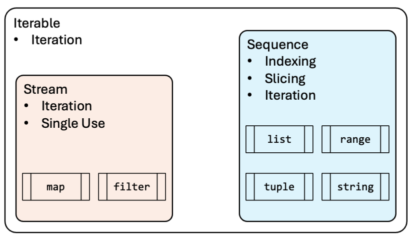

```
# Multiply a list of integers by n
def map_n(lst, n):
    return map(lambda item: item * n, lst)
# The result of map is a map.
```

Filter: `map(P, iterable)`
where `P` is a predicate (function) that returns a boolean value.
If the result of the `P(elem)` is `True`, we keep the `elem`; otherwise it is removed.




```
# Keep only multiples of n in a list
def filter_n(lst, n):
    return filter(lambda item: item % n == 0, lst)
# The result of filter is filter.
```

The result of `map` and `filter` is an iterable.



A **stream** is an iterable that is *single-use* (once the data is used up (e.g., in a for-loop), then the next time we try to use it again, it will be empty).

To preserve the values of stream, convert `map` or `filter` into `list` (or `tuple`), using `list()`.

`enumerate()`: takes an iterable, returns an enumerate object; adds a counter (index)

Sorting and Searching

Sorting

Selection Sort repeatedly selects the smallest element from the unsorted portion, then swaps it with the first unsorted element.

```
def selection_sort(lst):
  front = 0
  for _ in range(len(lst) - 1): # loop variable not
      used
    # find smallest in the rest
    smallest = front
    for i in range(front, len(lst)):
      if lst[i] < lst[smallest]:
        smallest = i
    # swap smallest with front
    lst[smallest], lst[front] = lst[front], lst[
      smallest]
    # extend the sorted sublist
    front = front + 1
  return lst
```

Bubble Sort repeatedly swaps adjacent elements if they are in the wrong order.

```
def bubble_sort(lst):
  is_sorted = False
  while not is_sorted:
    is_sorted = True     # rest flag
    for i in range(len(lst) - 1):
      if lst[i] > lst[i+1]:    # violation condition
        is_sorted = False     # not sorted
        lst[i], lst[i+1] = lst[i+1], lst[i]   # swap
  return lst
```

Built-in `sort()` method to sort lists:

- `lst.sort()`: sort in non-decreasing order
- `lst.sort(key=lambda arg: expr)`: sort in non-decreasing order of `key`
- `lst.sort(reverse=True)`: sort in non-increasing order.

`sorted()` accepts list, tuple or other sequence, and returns a sorted *new* list.

Searching

Linear Search iterates over the entire list.

```
def linear_search(lst, val):
  for idx in range(len(lst)):
    if lst[idx] == val:
      return idx
  return -1
```

If the list is sorted, **Binary Search** repeatedly divides the search interval in half, reducing time complexity to $O(\log n)$.

```
def binary_search(lst, val):
  left, right = 0, len(lst) - 1     # starting hands
  while left <= right:              # hands crossing
    middle = (left + right) // 2    # compute middle
    if lst[middle] == val:          # lucky!
      return middle                 #   found
    if lst[middle] < val:           # smaller
      left = middle + 1             #   move left to
      middle (exclude middle)
    else:                           # larger
      right = middle - 1            #   move right to
      middle (exclude middle)
  return -1                         # not found
```

Multi Dimensional

Array: nested list
Matrix: 2-dimensional array
To access i-th row and j-th column, `tbl[i][j]`.
Matrix operations:

- Addition: `c[i][j] = a[i][j] + b[i][j]`
- Multiplication: `c[i][j] = c[i][j] + (a[i][k] * b[k][j])` where $k = 0, \ldots, n-1$
  A better algorithm is the *Strassen Algorithm*.
- Transpose: `t[i][j] = m[j][i]`

Table: 2-dimensional array
`map` operation on a table is applied to every row.

- Type conversion
- Remove column
- Rearrange column
- Update column
- Add column

`filter` applied to a table: keep/remove rows (row selection operation)

Unordered Collection

Set is useful when we need our data to be distinct.
Set: `{expr1, expr2, expr3}`
To construct an empty set, use `set()`.
Set operations:

- Union |
- Intersection &
- Set difference −

A set is *unordered*, so indexing and slicing do not work.
Deduplication is performed automatically, so `{1, 2, 3} == {3, 2, 1, 2, 3}`.
Relational operators:

- Subset <=
- Proper subset <
- Superset >=
- Proper superset >

Check for pangram:

```
def pangram(alphabet, word):
    return set(alphabet) == set(word)
```

Dictionary: `{key: val, ...}`, or `dict([(key, val), ...])`.
The empty dictionary is `{}` or `dict()`.
Unique keys:

```
>>> dct = {1: 'A', 1: 'B'}
>>> dct   # only {1: 'B'} is kept
{1: 'B'}
```

Keys of a dictionary must be immutable. As such, we cannot use mutable values (e.g. list) as a key.

Dictionary is a *mutable* collection.

- Use `in` operator to check if a key exists.
- Retrieve value: `dct[key]`
- Update/insert value: `dct[key] = val`
- Delete key-value pair: `del dct[key]`
- Equality: ==

Dictionary methods:

- `dct.copy()`: Copy dictionary
- `dct.clear()`: Clear dictionary
- `dct.keys()`: Output keys
- `dct.values()`: Output values
- `dct.items()`: Output key-value pairs

Memoisation: store the results of expensive function calls in a dictionary, and reuse them when the same inputs are encountered again, in order to speed up calculations

Order of Growth

Rough measure of resources used:

1. *Space* (memory) required to run the program
2. *Time* taken to run the program

Note that order of growth is NOT the *absolute* time/space; it is the *proportion* of growth of the time/space of a program w.r.t. the growth of the input.
Formally, a given function $f(n)$ has **order of growth** $O(g(n))$ if there exist $k > 0$ sch that

$$f(n) \leq k \cdot g(n) \quad (n \geq n_0).$$

In Big-O notation, denote $f(n) \in O(g(n))$.
Given $f(n)$, we can find the tightest $g(n)$ as follows:

- Identify the dominant term(s).
- Ignore additive constants.
- Ignore multiplicative constants.

Common $g(n)$: $1, n, n^2, n^3, \log n, n \log n, 2^n$.