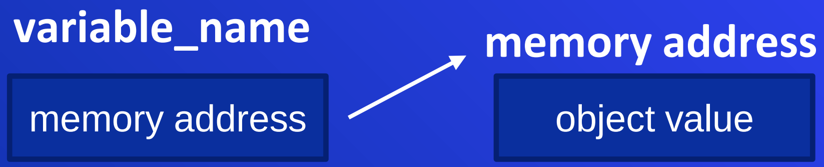
Introduction to Computer Science Using Python 3 Exam Review

Vincent Zhang

Topic 1: Fundamentals

Variable

- Named location in computer memory
- Reference to object value
- Variable contains memory address
- Value has memory address



variable_name

memory address

memory address

object value

```
variable = expression
```

Executing Assignment Statement

Evaluate expression → result is value → produce memory address with value → store in variable

Object

Immutable

- Cannot modify subsection of value after assignment
- int, float, bool, NoneType, str, tuple, range object

Mutable

- Can modify subsection of value after assignment
- list, dict

Variable & Constants Naming Convention

Constants: variables whose value cannot be changed

- **Start with: letter, _**
- **Contain: letters, digits, _**

```
variable_name_is_pothole_case_123 = expression
CONSTANT_NAME_IS_ALL_CAPS_POTHOLE_CASE_123 = expression
```

How many 10-character variable names are possible in Python 3?

$$(26+26+1)^1 * (26+26+10+1)^9 = 53 * 63^9$$

Operators
Order of Precedence

Highest Precedence

() changes order of precedence
func(args,...)
x[index:index]
x[index]
**
-x (negation)
* / // %
+ -
in, not in, <, <=, >, >=, !=, ==
not x
and
or

Comparison Operators (Compares Propositions)

Logical Operators (Connectives)

Lowest Precedence

Augmented Assignment Operators

```
x **= y
x = x ** y
```

Domination Laws

$$T \vee Proposition \Leftrightarrow T$$
$$F \wedge Proposition \Leftrightarrow F$$

Lazy Evaluation

Lazy evaluation on “and”, “or”

`if 1 == 1 or 2 / 0 == 2:` no ZeroDivisionError

Practical application: while loops

List Operators

```
>>> [1, 2, 3] + [3]
[1, 2, 3, 3]

>>> [2] * 3
[2, 2, 2]
```

String Operators

```
string + string (concatenation of Strings)
string * integer (concatenate integer copies of string)
other operand types and operators raises TypeError
```

Errors

Broad Categories

- Syntax: set of rules for valid combination of Python symbols
- Semantics: meaning of a combination of Python symbols

Specific Python Error Names

- SyntaxError: `82 = x`
- SyntaxError:

```
x = 2 +
2
```

- NameError: `x = y`
- ZeroDivisionError: `x = 2 / 0`

Type Conversion Errors: ValueError

☑ `str('string')`
✕ `int('string')`
✕ `float('string')`
☑ `bool('string')`
☑ `list('string')`
☑ `tuple('string')`
✕ `dict('string')`

How to solve second SyntaxError without removing anything?

```
x = (2 +
2)
```

Type ‘str’

Comments

```
# Single line comments
"""Unofficial multi
line comments: interpreted
"""
```

All of these forms mean the same to Python 3 interpreter: string literal

```
s = 'text'
s = "text"
s = """text"""

s = ('text')


s = ('''text''')
```

```
s = 't"e'xt'
s = "t'e"xt"
s = """te'
"xt"""
s = ('text' +
"123" + """2222""")

s = ('''te
xt'''
)
```

```
s = 't"e\'xt'
s = "t'e\"xt"
```

Why so many forms? What are the benefits?

How to solve the SyntaxError?

Escape Sequences: Special Characters

`'\''`: single quote string literal (')

`'\"'`: double quote string literal (")

`'\n'`: new line string literal (ASCII linefeed – LF)

`'\t'`: tab (ASCII horizontal tab – TAB)

`'\\'`: backslash string literal (\)

Notable String Methods

Method: function inside an object

```
object.method(arguments[, optional arguments])
```

Let’s write our own function alternatives for these string methods for practice!

```
str.rfind
str.lower
str.swapcase
str.isalpha
str.count
```

Topic 2: Importing

Importing Modules

Module: file containing functions

```
import module_name
module_name.function(arguments[, optional arguments])

import math
math.sqrt(4)
# 2.0
```

Importing Functions

```
from math import sqrt
print(sqrt(4))
# 2.0
```

Import Objects

```
import typing
def f(x) -> typing.List[str]:
    pass

from typing import List, Dict
def f(x: Dict[str, str]) -> List[float]:
    pass
```

Name Guard (Out of Scope)

```
if __name__ == '__main__':
    pass
if __name__ == 'module_name':
    pass
```

Let’s use our string functions by importing them!

Topic 3: Functions

Reduces Repetition

Improves Clarity

Eases Testing

Function Calls

```
function_name(arguments[, optional arguments])
```

Executing Function Calls

Evaluate arguments → results object values → produce memory addresses with values → store addresses in parameters → execute function body

```
abs(x)
round(number[, ndigits])
dir([object]), e.g., dir(__builtins__), dir(str), dir('string')
help([object])
pow(base, exp[, mod])
len(s)
min(iterable, *[, key, default])
min(arg1, arg2, *args[, key])
max(iterable, *[, key, default])
max(arg1, arg2, *args[, key])
```

Steps to Writing a Function

```
# STEP 2. Header: function/parameter name, type contract (param/return type)
def function_name(zero_or_more_parameters: param_type) -> return_type:
    """docstring section
    STEP 3. Description: describe what function does by describing return value,
    mention parameter by its name

    STEP 4. Preconditions if necessary
    Preconditions are restrictions on the domain of expected input

    STEP 1. Examples (at least 2): test input/expect output
    >>> function_name(arguments[, optional arguments])
    expected_result
    """

    # STEP 5. Body: write function body (algorithm)
    ...

    return expression # (optional)
```

Return: pass back a value

Return Statements

returns None by default

Evaluate expression → obtain value → store value in memory address → pass address of value to caller → exist function

Topic 4: Testing

Testing Template

```python
# Create new file to use unittest
import unittest
from module_name import function_name
class TestFunctionName:
    def test_function_name_edge_case_return_value(self):
        """Describe the type of edge case.
        """
        actual = function_name(something)
        expected = something
        msg = "some error message, we want " + actual + " but got " + expected
        self.assertEqual(actual, expected, msg)
    def test_function_name_mutation(self):
        """Testing if input mutated
        """
        some_input = [1, 2, 3]
        expected = some_input.copy()
        function_name(some_input)
        msg = generate_error_message(some_input, expected)
        self.assertEqual(some_input, expected, msg)

def generate_error_message(actual, expected):
    return 'Wnat ' + expected +' but got ' + actual

if __name__ == '__main__':
    unittest.main(exit=False)
```

Case 1: Test return value

Case 2: Test mutation

Parameter: variables used in scope of function

```python
# STEP 6. Test: run examples
# Manual testing
function_name(some_input)

# Automatic testing of docstring examples
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Argument: value given to function

Pass: to provide to a function

Call: ask Python to evaluate a function

Function calls are expressions because an expression is returned, so function calls can be assigned to variables

doctest vs. unittest

prompt vs. .py file

unittest Notation

. ← pass

E ← error, e.g., divide by zero

F ← failed, assertEqual finds actual, expected no match

Choosing Test Cases

Size

- Collections (str, list, tuple, dict): 0, 1, many elements

Dichotomy (Pairwise Opposites)

- E.g., odd/even, vowel/consonant, pos/neg, empty/full

Boundary (Neighborhood of Thresholds)

- E.g., bus fare by age thresholds: child/youth/adult/senior

Order

- E.g., bubble sort algorithm, test order

Topic 5:
Input Output

Standard Input/Output

Input (strips newline) `input([prompt])`

```
input()
input('Enter some value.')
# returns string
```

Output (adds newline) `print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)`

```
print()
print(123)
print('123', end='')
print('123', str(456))
print('123', str(456), sep='$')
print('123', str(456), end='_', sep='')

123
123123 456
123$456
123456_
```

Files

Read (Input)

```
open('file_path\\file.txt')
open('file_path\\file.txt', 'r')
```

Manual Close

```
file_read = open('file_path\\file1.txt', 'r')
file_write = open('file_path\\file2.txt', 'w')
file_read.close()
file_write.close()
```

Write (Output)

```
open('file_path\\file.txt', 'w')
open('file_path\\file.txt', 'a')
```

Automatic Close (Out of Scope)

```
with open('file_path\\file1.txt', 'r') as file_read:
    pass
with open('file_path\\file2.txt', 'w') as file_write:
    pass
```

File Object Methods

`file.readline()` read from current position to next `'\n'`

`file.readlines()` read from current position to `''`, store as list where each list element is a sequence of characters ending in `'\n'`

`file.read()` read from current position to `''`, store as string

`file.write()` file version of `print(end='')`

`file.close()` closes the file

4 Ways to Read Through a File

1. readline()

```
file = open('file.txt', 'r')
line = file.readline()
while line != '':
    print(line, end='')
    line = file.readline()
file.close()
```

2. for line in file:

```
file = open('file.txt', 'r')
for line in file:
    print(line, end='')
file.close()
```

3. read()

```
file = open('file.txt', 'r')
data = file.read().split('\n')
for line_without_newline in data:
    print(line_without_newline, end='')
file.close()
```

4. readlines()

```
file = open('file.txt', 'r')
data = file.readlines()
for line_with_newline in data:
    print(line_with_newline, end='')
file.close()
```

Initial File State

```
linee1
linee2
linee3
linee4
linee5
linee6
```

Standard Output

```
'linee1\n'
'linee2\n'
'linee3\n'
'linee4\n'
['linee5\n', 'linee6\n']
'linee1\nlinee2\nlinee3\nlinee4\nlinee5\nlinee6\nadded'
```

4 methods can be mixed

```
file = open('file.txt', 'r')
print(repr(file.readline()))
count = 0
for line in file:
    print(repr(line))
    if count == 2:
        break
    count += 1

print(file.readlines())
file.close()

with open('file.txt', 'a') as file:
    file.write('added')
with open('file.txt') as file:
    print(repr(file.read()))
```

repr() Out of Scope

Topic 5: Conditional Statements (if, elif, else)

Group by if Statements (Single Layer)

- 1 if
  - 0 elif
    - 0 else
    - 1 else
    - Cannot have multiple else
  - 1 elif
    - 0 or 1 else
  - Multiple elif
    - 0 or 1 else
- 0 if
  - 0 elif, cannot have elif without if
  - 0 else, cannot have else without if

Outside Functions

Each group of if statements is evaluated separately.

1 group of if statements

Group 1: 1 if, 1 elif, 0 else

```
if boolean_expression:
    pass
elif boolean_expression:
    pass
else:
    pass
```

2 groups of if statements

Group 1: 1 if, 0 elif, 0 else

Group 2: 1 if, 0 elif, 1 else

```
if boolean_expression:
    pass
if boolean_expression:
    pass
else:
    pass
```

Inside Functions

Use of `return` in functions allow for simplifications.

Simplifying Conditional Branches

```
def f():
    if boolean_expression:
        return
    if boolean_expression:
        return
    return
```

Simplifying Boolean Return

```
def f():
    if boolean_expression:
        return False
    else:
        return True
```

```
def f():
    return not boolean_expression
```

Nested Conditional Satements (Multiple Layers)

Original

```
if bool_a:
    if bool_b:
        print('a b')
    elif bool_c:
        print('a c')
    else:
        print('a d')
```

Same as

```
if bool_a and bool_b:
    print('a b')
elif bool_a and bool_c:
    print('a c')
elif bool_a:
    print('a d')
```

Not same as

```
if bool_a and bool_b:
    print('a b')
if bool_a and bool_c:
    print('a c')
elif bool_a:
    print('a d')
```

Topic 6: Subscripting: Indexing & Slicing

Overview

Applies to: `str`, `list`, `tuple`, `range object`

Does not apply to: `int`, `float`, `dict`, `bool`, `NoneType`

raises TypeError for invalid cases

Index: position within a valid collection

`collection[index]`

`index` $\in \mathbb{Z}\,\{-\infty, \ldots, -1, 0, 1, \ldots \infty\}$

raises IndexError when index out of range of collection

Slicing

```
collection[:]
collection[::]
collection[start(inclusive):stop(exclusive):step]
```

Slicing $\mathbb{Z}$: Function range()

`range([start, ]stop[, step])`

Returns range object containing a collection of integers specified by start, stop, step

Parallel Collection: two or more collections of the same object related by index

Slice: extraction of elements at specified positions to form a new valid collection

Topic 7: Loops (for, while)

Keyword in & Loops

Known number of iterations: for

Unknown number of iterations: while

in: element in collection

```
char in string
elem in lst
elem in tup
key in dictionary
integer in range(arg)
```

Checking Membership

```
if char in string:
if elem in lst:
if elem in tup:
if key in dictionary:
if integer in range(arg):
```

Iterate Over Collection

```
for char in string:
for elem in lst:
for elem in tup:
for key in dictionary:
for integer in range(arg):
```

while Loop

```
while boolean_condition == True:
    # Iterate
```

Accumulator Variable:

accumulates value

```
str, list, int, float
```

Application: write built-in sum()

```
def our_sum(collection):
    accumulator = 0
    for val in collection:
        accumulator += val
    return accumulator
print(our_sum({1:2, 3:4}))
# Output: 4
# raises TypeError in non-
# numeric collection
```

Lazy Evaluation Application

```
def contains(collection, value):
    index = len(collection) - 1
    while index >= 0 and collection[index] != value:
        index = index - 1
    return index != -1
print(contains([1, 2, 3], 3))
```

Loops in Functions: Early Return

```
def find_linear_search(collection, value):
    for i in range(len(collection)):
        if collection[i] == value:
            return i
    return -1
print(find_linear_search([1, 2, 3], 2))
print(find_linear_search([1, 2, 3], -1))
# Output: 1\n-1\n
```

Nest Loops Application: Pascal’s Triangle

1

1 1

1 2 1

1 3 3 1

…

$$\binom{0}{0}$$
$$\binom{1}{0} \binom{1}{1}$$
$$\binom{2}{0} \binom{2}{1} \binom{2}{2}$$
$$\binom{3}{0} \binom{3}{1} \binom{3}{2} \binom{3}{3}$$

…

```python
from math import factorial

def output(rows):
    for i in range(rows):
        row = ''
        for j in range(i + 1):
            row += str(int(factorial(i)/factorial(j)/factorial(i-j))) + " "
        print(row)
print(output(4))
```

```
"""Output
1
1 1
1 2 1
1 3 3 1
None

"""
```

Global Variables

factorial	id1	**Id1: function** factorial(...)
output	id2	**Id2: function** output(rows)

$$\binom{n}{r} = \frac{n!}{r!\,(n-r)!}$$

$$x! = x(x-1)\dots 1$$

$$0! = 1$$

In scope of output(rows), Note: object memory address simplified

rows	4													
i	0		1			2				3				
row	“”	“1 ”	“”	“1 ”	“1 1 “	“”	“1 ”	“1 2 ”	“1 2 1 ”	“”	“1 ”	“1 3 ”	“1 3 3 ”	“1 3 3 1 ”
j	0		0	1		0	1	2		0	1	2	3	
print		1\n			1 1 \n				1 2 1 \n					1 3 3 1 \n

Return value id19 **Id19: NoneType** None

print: None\n

All variables in scope of output(rows) are automatically deleted along with unused memory addresses

Topic 8:
Data Structure
list[], tuple(), dict{}

list & tuple & dict

list (Mutable; Elements: no restrictions)

```
lst = [expression_1, expression_2,..., expression_n]
lst = [2]
lst = []

lst[index] = value
# raises IndexError if index out of range
```

List Methods that Modify the List

```
lst.append(element)
lst.insert(index, element)
lst.extend(lst)
lst.sort()
lst.reverse()
lst.pop(optional_index)
lst.remove(element)
```

List Methods that Do Not Modify the List

```
lst.count(obj)
lst.index(obj)
# find() is not a list method
```

Sorting With sorted()

```
lst = [12, 4, 5]
lst = sorted(lst)
```

Aliasing of Mutable Objects

Aliased objects allow changes to be applied when a section of the object is modified.

tuple (Immutable; Elements: no restrictions)

```
tup = (expression_1, expression_2,..., expression_n)
tup = (2,)
tup = ()

tup.count(obj)
tup.index(obj)
```

dict (Mutable; Keys: Immutable, Values: no restrictions)

```
d = {
  key: value,
  key: value,
  ...,
  key: value
}
```

```
# if key exists, overrides value
d = {key: value_1}
d[key] = value

# to change value at existing key, key must exist
# in dictionary
d[key] = []
d[key].append(123)

# Dictionary Delete Key
del dict[key]

# Heterogeneous Dictionaries
# Dictionary with keys of different object types
d = {'string': 2, 123: 2}
```

Let’s practice dictionary iterating!

Topic 9:
Algorithm Analysis

Algorithm: sequence of steps to accomplish a task

Categories (variable # of iterations): worst case, best case, (average case, out of scope)

Criteria: # of comparisons, # of iterations, # of assignments

Runtime: quadratic, linear, logarithmic

Searching Algorithm: search for index of value

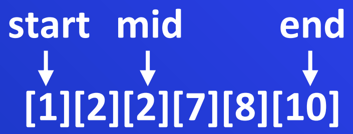
- Linear Search (linear runtime): check every element one by one in list
- Binary Search (logarithmic runtime): recursively check middle of SORTED sub-lists

Sorting Algorithm (quadratic runtime)

- Bubble Sort: bubble smallest/largest to one end of list in each pass
- Selection Sort: swap min/max in unsorted part with first/last element in unsorted part
- Insertion Sort: place first/last value in unsorted part in correct position in sorted part by shifting

Binary Search

Find 2:

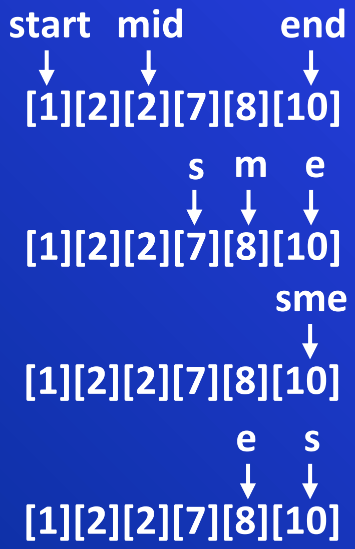


start mid end

[1][2][2][7][8][10]

Found: 2 at index 2

Find 9:



start mid end

[1][2][2][7][8][10]

s m e

[1][2][2][7][8][10]

sme

[1][2][2][7][8][10]

e s

[1][2][2][7][8][10]

Did not find 9

```python
def bin_search(data, val):
    s = 0
    e = len(data) - 1

    while s <= e:
        m = s + (e - s) // 2
        if data[m] == val:
            return m
        if data[m] < val:
            s = m + 1
        else:
            e = m - 1
    return None
print(bin_search(sorted([10, 2, 2, 7, 8, 1]), 2))
# [1, 2, 2, 7, 8, 10], 2
# Output: 2
# [1, 2, 2, 7, 8, 10], 9
# Output: None
```

Bubble Sort

0 1 2 3 4 5
[10][2][2][7][8][1]

```
def bubble_sort(data):
    print("Original:", data)
    for i in range(len(data) - 1, 0, -1): # Sorted boundary
        for j in range(i): # Swap up to boundary
            print(f"i:{i}, j:{j}, j+1:{j+1}")
            if data[j] > data[j+1]: # Swap if necessary
                data[j], data[j+1] = data[j+1], data[j]
                # Concurrent swap alternative: 3 variables
        print(f'Pass {len(data)-i}: {data}')
bubble_sort([10, 2, 2, 7, 8, 1])
```

f-strings out of scope

```
"""Output
Original: [10, 2, 2, 7, 8, 1]
i:5, j:0, j+1:1
i:5, j:1, j+1:2
i:5, j:2, j+1:3
i:5, j:3, j+1:4
i:5, j:4, j+1:5
Pass 1: [2, 2, 7, 8, 1, 10]
i:4, j:0, j+1:1
i:4, j:1, j+1:2
i:4, j:2, j+1:3
i:4, j:3, j+1:4
Pass 2: [2, 2, 7, 1, 8, 10]
i:3, j:0, j+1:1
i:3, j:1, j+1:2
i:3, j:2, j+1:3
Pass 3: [2, 2, 1, 7, 8, 10]
i:2, j:0, j+1:1
i:2, j:1, j+1:2
Pass 4: [2, 1, 2, 7, 8, 10]
i:1, j:0, j+1:1
Pass 5: [1, 2, 2, 7, 8, 10]
"""
```

Selection Sort

```
def selection_sort(data):
    print("Original:", data)
    for i in range(len(data) - 1): # < i: sorted part
        min_index = i
        for j in range(i + 1, len(data)): # Find min
            print(f"i:{i}, j:{j}")
            if data[j] < data[min_index]:
                min_index = j
        data[i], data[min_index] = data[min_index],
data[i]
        print(f'Pass {i+1}: {data}')
selection_sort([10, 8, 7, 2, 2, 1])
```

```
"""Output
Original: [10, 8, 7, 2, 2, 1]
i:0, j:1
i:0, j:2
i:0, j:3
i:0, j:4
i:0, j:5
Pass 1: [1, 8, 7, 2, 2, 10]
i:1, j:2
i:1, j:3
i:1, j:4
i:1, j:5
Pass 2: [1, 2, 7, 8, 2, 10]
i:2, j:3
i:2, j:4
i:2, j:5
Pass 3: [1, 2, 2, 8, 7, 10]
i:3, j:4
i:3, j:5
Pass 4: [1, 2, 2, 7, 8, 10]
i:4, j:5
Pass 5: [1, 2, 2, 7, 8, 10]
"""
```

Insertion Sort

```python
def insertion_sort(data):
    print("Original:", data)
    for i in range(1, len(data)): # < i sorted
        j = i
        value = data[j]
        # Shift to correct position
        while j > 0 and data[j-1] > value:
            print(f"i:{i}, j:{j}, j-1:{j-1}")
            data[j] = data[j-1]
            j -= 1
        data[j] = value
        print(f'Pass {i}: {data}')
insertion_sort([10, 2, 2, 7, 8, 1])
```

```
"""Output
Original: [10, 2, 2, 7, 8, 1]
i:1, j:1, j-1:0
Pass 1: [2, 10, 2, 7, 8, 1]
i:2, j:2, j-1:1
Pass 2: [2, 2, 10, 7, 8, 1]
i:3, j:3, j-1:2
Pass 3: [2, 2, 7, 10, 8, 1]
i:4, j:4, j-1:3
Pass 4: [2, 2, 7, 8, 10, 1]
i:5, j:5, j-1:4
i:5, j:4, j-1:3
i:5, j:3, j-1:2
i:5, j:2, j-1:1
i:5, j:1, j-1:0
Pass 5: [1, 2, 2, 7, 8, 10]
"""
```

Topic 10:
cGPA Calculator Practice Project!

Storing Data

Input.txt

$$FORMAT$$ (% grade) (% weight)

```
Course:Intro to CS
80 8 Midterm_1
95 12 Midterm_2
50 8 Assignment_1
15 10 Assignment_2
100 14 Assignment_3
100 48 Final_Exam
END
```

read_data()

```
data = {
    course: [
        (grade, weight, description),
        (grade, weight, description)],
    course: [
        (grade, weight, description),
        (grade, weight, description)],
}
```

Functionality

```
get_cGPA(gpa_by_course)
    gpa_by_course = calculate_gpa(data)

    gpa_by_course = {
        course: gpa,
        course: gpa
    }

get_max_gpa_course_by_insertion_sort(course_by_gpa)
    course_by_gpa = flatten_inverted_dictionary(gpa_by_course)
    course_by_gpa = [
        (gpa, course, course),
        (gpa, course, course)
    ]
contains_gpa_course_by_binary_search(gpa_by_course, gpa_exists)

    from sorting_algorithms import insertion_sort

    from binary_search import bin_search
```