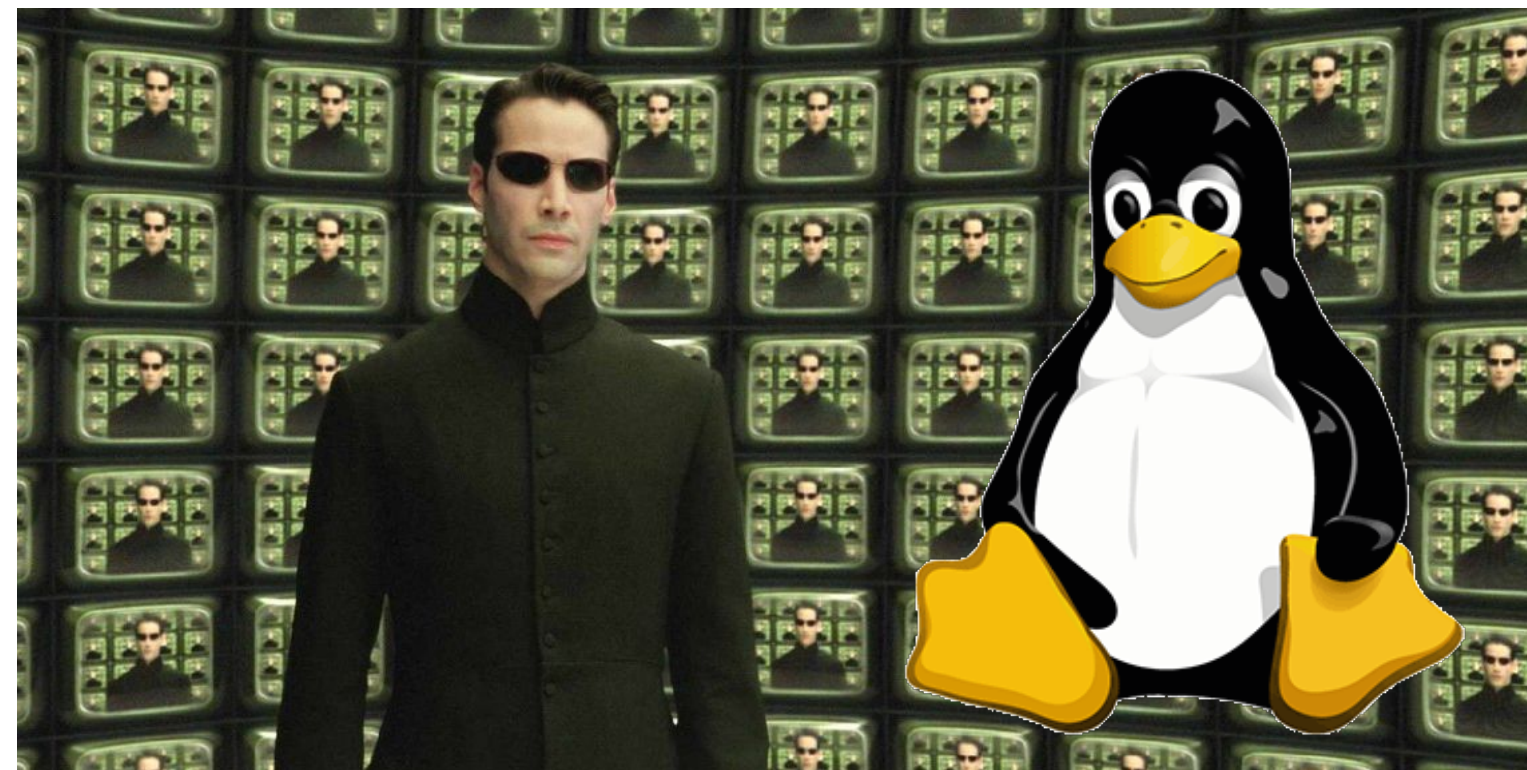




CSCI 120

Introduction to CompSci and Programming I

Review



Jetic Gū

Announcement

- Last Week
 - Today: 1st half review, 2nd half QA
 - Friday: Office Hour



Overview

- Focus: Python Programming
- Architecture: von Neumann
- Core Ideas:
 1. Native Data Structures
 2. Subroutines
 3. Algorithm Analysis

Native Data Structures

Why Python

- Easy, Powerful
 - Loads of useful libraries such as Panda, PyTorch, etc.
- Scientific Computation
 - Autonomous Driving, Machine Translation, Virtual Assistant, etc.
- Data Analysis
 - Big Data, Smart Business, etc.

Python Versions

- Python 2.7
 - Most of the tutorials on the Internet
 - Huge codebases on the Internet
 - Discontinued since 2020. Do NOT use Python 2 now
- Python 3
 - Current version 3.9, but anything after 3.6 should be fine
 - Differences: better optimisation, more features, improved security and stability

Python Variables

- In Python, all variables are objects
 - Numbers: `int`, `float`
 - Boolean: `True`, `False` (yes, booleans are actually numbers)
 - String: `str`
 - Advanced native data structures: `list`, `dict`
 - Object-Oriented Programming: `class`
 - None

Python Variables

- Type Conversion
 - Numbers: `int(x)`, `float(x)`, `str(x)`
Type conversion is automatic between `float` and `int` when doing calculation
 - `list` and `tuple`
 - You can convert a `tuple` to a `list`; this might not work v.v.
 - You can modify elements in a `list`; You cannot use a `list` as a key in `dict`
 - You cannot modify elements in a `tuple`; You can use a `tuple` as a key in `dict`

Python Variables

- Numerical Arithmetics
 - `x + y`
 - `x - y`
 - `x ** y`
 - `x / y`
 - `x // y`
 - `x / float(y)`
 - `x % y`
- String
 - `str(x) + " " + str(y)`

Python Variables

- `strings`
- `strings` are like lists of characters

```
a = 'Hello world\twhatsup'
b = "cheese\tburger\n"
print(len(a), len(b))
a.lower()
b.upper()
a.split()
b.strip()
" ".join(["Hello", "World"])
```

List

- **Collections:** `list`, this is like mutable arrays.

```
names = ['Fry', 'Leela']
```

```
names[0] == 'Fry'
```

```
names.append('Bender')
```

```
len(names) == 3
```

```
print(names)
```

```
names.extend(['Amy', 'Prof'])
```

```
print(names)
```

```
names = [] # Creates an empty list
```

```
names = list() # Also creates an empty list
```

```
stuff = [1, ['hi', 'bye'], -0.12, None] # Can mix types
```

List

- `list` methods. Assuming `a` is a list of numbers, here are some common ones:
 - `a.append(stuff)`
 - `a.insert(index, stuff)`
 - `a.sort()`, `a.reverse()`

List

- Slicing a `list` (not a pizza)
- **Basic format:** `some_list[start_index:end_index]`

```
numbers = [0, 1, 2, 3, 4, 5, 6]
numbers[0:3] == numbers[:3] == [0, 1, 2]
numbers[5:] == numbers[5:7] == [5, 6]
numbers[:] == numbers = [0, 1, 2, 3, 4, 5, 6]
numbers[-1] == 6 # Negative index wraps around
numbers[-3:] == [4, 5, 6]
numbers[3:-2] == [3, 4] # Can mix and match
```

Tuples

- **Collections:** tuple, these are immutable arrays

```
names = ('Charles', 'The Cat') # Note the parentheses
names[0] == 'Charles'
len(names) == 2
print(names)
names[0] = 'Richard' # This will not work
empty = tuple() # Empty tuple
empty = (,) # Empty tuple
single = (10,) # Single-element tuple. Comma matters!
```
- Same slicing like `list`

Dict

- **Collections: dict, essentially key-value map**

```
Age = dict()           # Empty dictionary
Age = {}              # Empty dictionary
Age = {'Jetic': '44'} # Dictionary with one item
Age['Papa'] = '68'
print('Jetic' in Age)
print('Kevin' in Age)
print(Age['Jetic'])
del Age['Papa']
print(Age)
```

Dict

- `dict` methods. Assuming `a` is a `dict` object
 - `a.items()`
Return list of `(key, value)` tuples
 - `a.values()`
Return list of all values
 - `a.keys()`
Return list of all keys

Class

- `Class` are custom data structures
- `Classes` can have methods, and attributes
- `objects` of the same `class` share all methods, but have their own attribute values
- Special method: `def __init__(self, ...)`

Python Routines

Indentation

- Routines are created using indentation
- You can use space (2, 3, 4) or tabs, but be consistent!
- What is the following code doing?

```
def fib(n):  
    # Indent level 1: function body  
    if n <= 1:  
        # Indent level 2: if statement body  
        return 1  
    else:  
        # Indent level 2: else statement body  
        return fib(n-1)+fib(n-2)
```

Loops

```
for name in ['Fry', 'Leela', 'Bender']:
    print('Hi ' + name + '!')
while True:
    print 'We\'re stuck in a loop...'
    break
```

- You can create a new list using `for` loops
`a = [word.upper() for word in words]`
- You can have `for` loops inside `for` loops
`for i in range(n):`
 `for j in range(m):`
 `print(matrix[i][j])`

Loops

- `range()` function returns an iterator, which can be converted to a `list` when not used in a `for` loop
 - `list(range(10))`
 - `range()` has starting indices, `list(range(5, 10))`
 - `range()` has steps, `list(range(5, 10, 2))`

if conditions

```
if condition:  
    do stuff  
else:  
    do other stuff
```

- Conditions
 - See if something is None
if x is None:
 do stuff
 - See if something is not None
if x is not None:
 do stuff

if conditions

- Conditions
 - key **inside** a dict

```
if key in aDict:  
    do stuff
```
 - key **not inside** a dict

```
if key not in aDict:  
    do stuff
```

if conditions

- Conditions
 - Multiple conditions: use boolean operators (and, or, not)

```
if Leela.likes(Fry) and Fry.likes(Leela):  
    break
```


Functions / Methods

- Declaration

```
def func(params):  
    do stuff  
    return values    # optional
```

- Parameters:

- No parameter:

```
def func()
```

- Fixed parameter:

```
def func(a, b, c)
```

- Fixed parameters with defaults: `def func(a, b='1', c=None)`

- parameters with default values are considered optional and you cannot have non-optionals after optionals

Functions / Methods

- Function calls

```
func(param)
```

```
a = func()      # No param or all using defaults
```

```
a = func('Jetic', 44)  # Multiple parameters
```

```
a = func(name='Jetic', age=44)  # Named parameters
```

```
a = func(age=44, name='Jetic')  # Equivalent to above
```

Algorithm Analysis

Understanding an algorithm

- Reading the code

```
def fib(n):  
    if n <= 1:  
        return 1  
    else:  
        return fib(n-1)+fib(n-2)
```

- Line-by-line: what is each line doing? what is the complexity of each line?
- Finally: what is the whole code doing? what is the overall complexity?

Understanding an algorithm

- Reading the pseudocode

```
Function travel(root)
```

```
    root.visited()
```

```
    if root has left child:
```

```
        travel(root.left)
```

```
    if root has right child:
```

```
        travel(root.right)
```

- Pseudocode may not be real programming code, but should be understandable

The Big-O Notation

- Mathematical notation that describes the limiting behaviour of a function when the argument tends towards a particular value or infinity
 - Ignore constants
 - Ignore terms with lesser power
- Time complexity
- Memory complexity

Important Algorithms

- Binary Search
- Binary Tree Preorder Traversal