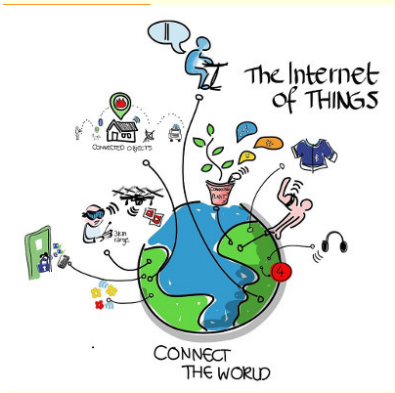
SE-IOT: Internet of Things



Python and Linux on the Raspberry PI

Derek Kiong
dkiong@nus.edu.sg

Python/Linux

- Both Linux and Python chosen as key software for Raspberry PI
- Open Source is riding on code reuse, enhancement and portability
- Large user-base for support facilitates active code repository
- Tooling mindset
- Agile and productive

What/Why Linux?

- Open source/modular principles -- operating system with elegant abstraction
  - Flexibility from abstraction (eg, device vs files)
- Extensive tool set available
- Comprehensive GNU libraries available
- Stable software repositories
- Re-targetable open source code (eg Python interpreter)
  - MacOS, Windows, Linux, ARM etc

Successful software reuse strategy

- Linux reuse strategy from
  - Program as a reusable component
    - **`fork()`** and **`exec()`** system calls
  - I/O abstraction and redirection
    - **`dup()`** system call
  - Pipes for convenient std I/O communication
    - **`pipe()`** system call

What/Why Python?

- Interpreter model following from shell **`/bin/sh`**
  - since shell and PERL does not scale well
- small and consistent programming language
- supports multiple programming paradigms
  - object-oriented, imperative and functional
- design philosophy emphasizes code readability □has clear syntax
- □old□language, but growing popularity in year 2017/2018

What/Why Python?

- latent (dynamic) type system
- run-time model similar to Scheme/Lisp
- installed with standard library, but extensible

Python Features

- Dynamic (latent) typing □does not require declaration
  - Contrast C and Java which have strong typing (implemented through compile-time checks)
- Dynamic objects with built-in memory management
  - Similar with Java and C#
  - Contrast C and C++
- Even function definition and closures are dynamic

Comments, Keywords & Structure

- Line comments follow **#**
- Python keywords are as follows

`and`	`del`	`from`	`not`	`while`
`as`	`elif`	`global`	`or`	`with`
`assert`	`else`	`if`	`pass`	`yield`
`break`	`except`	`import`	`print`	
`class`	`exec`	`in`	`raise`	
`continue`	`finally`	`is`	`return`	
`def`	`for`	`lambda`	`try`	

- Other identifiers may be variable names
- Structure implied by indentation

Variables and Data

- Variables/types need not be declared before use
- Variable to object reference are created on assignment
- Scoping rules differ for assignment/access
  - Search enclosing scope for access
  - Local scope for assignment
- Predefined types are **`int`**, **`float`**, **`str`**

Variables Example

```
$ python
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license"
for more information.
>>> a=1
>>> b=4
>>> print a+b
5
>>>
```


Functions

- Functions are also created at runtime and on the fly
- Nested functions retain their closure (enclosing environment)
- Function parameters may have default values
- Function invoked with positional or named parameters
  - error if remaining parameter does not have value

Function Example

```
>>> def factorial(n):
...     if (n>1):
...         return n*factorial(n-1)
...     else:
...         return 1
...
>>> factorial(b)
24
```


Function with default values

```
>>> def make_increment1(n):
...     def increment(x):
...         return x+n
...     return increment
...
>>> def make_increment2(n):
...     def increment(x, n=n):
...         return x+n
...     return increment
...
>>>
```


Function with default values

```
>>> def make_increment3(n):
...     def increment(x, mul=1, n=n):
...         return x*mul+n
...     return increment
...
>>> add1 = make_increment1(10)
>>> add2 = make_increment2(2)
>>> add3 = make_increment3(5)
>>>
```


Function with default values

```
>>> add3(6)
11
>>> add3(6, mul=2)
17
```


Class and objects

- Python classes allow for defining new structures and multiple inheritance
- Class defines object creation with enclosing scope for instance variables (created by constructor)

Class definition

base class

constructor

instance variable

call constructor of base class

explicit **self**

```
class Product(object):
    def __init__(self, code, stock, price):
        object.__init__(self)
        self.code = code
        self.stock = stock
        self.price = price
    def getPrice(self):
        return self.price
    def setPrice(self, price):
        self.price = price
    def getStock(self):
        return self.stock
    def setStock(self, stock):
        self.stock = stock
    def getCode(self): return self.code
    def totalvalue(self): return self.getStock()*self.getPrice()
```


Class instantiation

```
>>> p1=Product("10010", 180, 2)
>>> p2=Product("10013", 240, 1)
>>> p1.getCode()
'10010'
>>> p1.totalvalue()
360
```


Summary

- Scalability from language constructs and modularity
- Flexibility from latent-typed variables and objects
- Large variety of libraries for regular expressions, networking, protocols such as **`http`**/**`imap`**/**`ftp`** etc

