Python Essentials 2015

Benny Khoo

16/10/2015

Reachable via

- Linkedin
- Github

Programming languages and model

- C – expose low level details to coders: memory, pointers, native data type, structure, strongly typed etc. Still valid for kernel, drivers, BIOS and compiler.
- C++ – data as object in addition to low level details.
  - Preferred languages for system programming.
  - Examples Chrome, LLVM, Android, Google and many more.
- Objective-C – like C++ offer different opinion about object oriented programming.
  - ARC liberated coders from manual memory management.
  - Popular in iOS.
- Java – automatic memory management (Garbage collection), use reference instead of pointer, object oriented, strongly typed .
  - Run on uniform model of machine called virtual machines.
  - Usually run cross-platform (WORA).

… programming languages and model

- Haskell, Erlang – full blown functional languages. Functions should not have side effects.
  - Parallel execution become easy again.
- Scripting languages (PHP, Perl, Python, Tcl)
  - dynamic evaluation, usually type-less.
  - no memory management are required.
  - low level details are usually omitted or simplified become cumbersome - many applications do not need to know low-level anyway.
  - Guided by certain philosophies by the creator –common programming patterns are simplified as features in the language.

Python and computing model

- Every kind of values including the primitive ones are objects.
- Python does not have concept about native data type. Integer in Python can be longer than the computer can support!
- Python language design naturalizes human thinking to a computer. In contrast low level languages expose the intricate details in a machine.

Understanding Python syntax

- *Demo*

How to invoke Python

% python –i <script>
% python –mpdb <script>
% python

Debugging

- *Demo*

Using IDE

- WingIDE, PyCharm
- Hyperlinks
- Code folding
- GUI debugger
- Quick help
- Automatic code refactoring
- Automatic code completion
- Auto such and such …

Errors in your program

- Syntax errors
- Runtime errors
- Semantic errors

Values and Types

- Supported Literals: “Hello world”, 123, 1.23, 1e-6, 0xff, 0b1011, 0o10
- Constant value: True, False and None
- Using type function

Variable names and keywords

```
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
```

```
try = a
    ^
SyntaxError: invalid syntax
```

Operators

- Arithmetic: +, -, *, /, // (floor division), ** (power) and % (modulus)
  - Unary: +, - and ~
- Comparison: <, >, ==, >=, <=, <>, !=, in, not in, is, is not
- Boolean: and, or
  - Unary: not
- Bitwise: |, &, ^
  - Shift operators: <<, >>

Orders of operations

- Honors operator precedence
- 2**1+1 is 3, not 4
- 6 + 4/2 is 8 not 5
- Operators with same precedence is evaluated from left to right:
  - degrees / 2 * pi – division happens first then multiplied
  - To divide to 2π: degrees / 2 / pi

Expression

- Expression always yields a value
- Depending on values evaluated in expression an expression may compute to any supported value type including sequence.
  - 0x5f3759df - (i >> 1) => numerical value
  - a and b is not c => True/False
  - a in locals
  - [i for i in range(10) if i % 2 == 0] => list

Types of expression

- Arithmetic expression - expression that yield a value: e = m * c**2
- Boolean expression:
  - Comparison: x > y
  - Containment test: a in something
  - Identity test:

```
a = Foo()
b = a
a is b # True
a == b # True/False?
```

- Obviously you can compose them: a % 2 == 0

Assignments

- Conventional: x = y
- In-place assignments: +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, **=, //=

Statement

- A program is make of a sequence of statements
- Statement can be considered a unit of execution
- A statement process some data
- Technically an expression is also a statement

Grammar

- Python BNF grammar

Conditions

```
If x is True:
        pass
```

```
if x < y:
        print('x is lesser than y')
else:
        print('y is greater')
```

```
if x == y:
        print('x is equal to y')
elif x > y:
        print('but x is greater')
else:
        print("y is greater')
```

```
# conditional expression
True n if % 2 == 0 else False
```

Iteration

```
while n > 0:
    print(n)
    n=n-1
```

```
for n in range(10)
    print(n)
```

```
for x in sieve(10000):
   if testn % x == 0:
      sum *= x
      print(x, sum)
      if sum >= testn:
         print("largest prime is", x)
         break
```

```
for s, t in genCoPrime():
    if s == 0 or t == 0:
        continue
```

Functions

- Basic concepts: -
  - Function stack, frame
  - Local and global variables
  - Stack trace

```
import math

def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print(dx, dy)
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    return result

print(distance(0, 0, 3, 4)) # 5.0
```

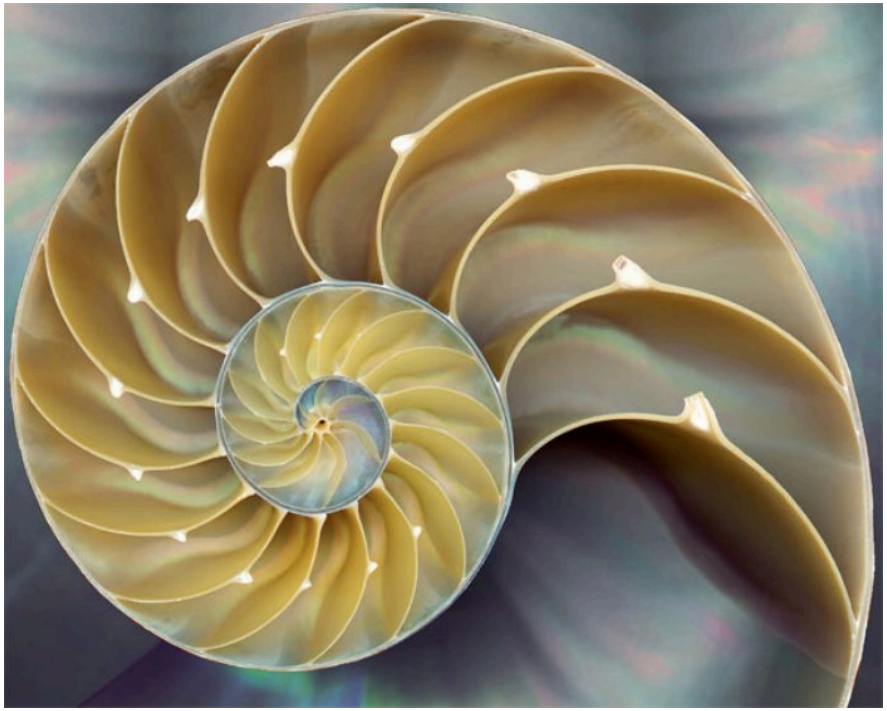
```
def divide(a,b):
    q = a // b
    r = a - q*b
    return (q, r)

q, r = divide(10, 3)
print(q, r) # 3 1
```

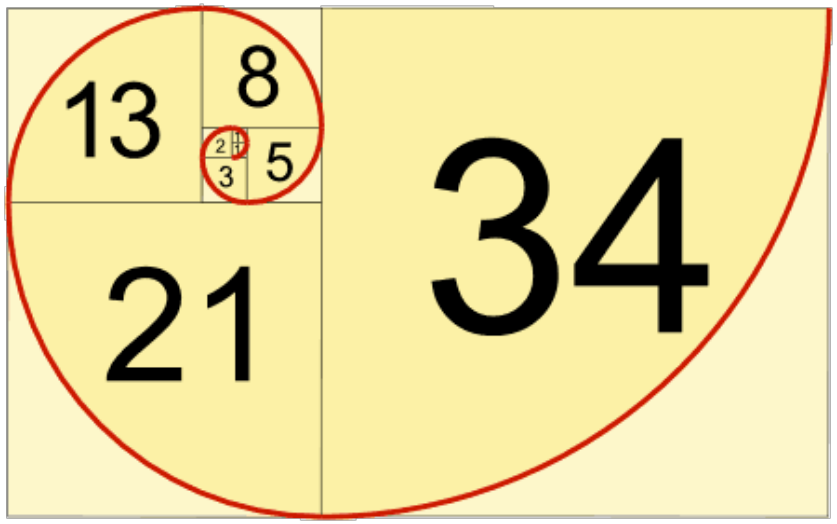
Recursive functions

- Basic concepts: -
  - Function that call itself
  - Use stack to reduce the problem until it becomes solvable
  - Termination criteria
  - Mutually recursive function

```
def fibonacci (n):
    if n == 0:
        return 0
    elif  n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Recursive nature





The Rule is $F_n = F_{n-1} + F_{n-2}$

Strings

a = “Hello world”
b = ‘Python is cool’
c = “””Computer says ‘no’”””

d = ‘’’Content-type: text/html
<h1>hello world</h1>
Click here.
‘’’

String as a sequence

A string is a sequence

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

Length of a string

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

Out of bound error

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

Traversing a string

Traversing a string

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

Or even simpler

```
for char in fruit:
    print(char)
```

String slices

```
>>> s = 'Monty Python'
>>> print(s[0:5])
Monty
>>> print(s[6:12])
Python

>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'

>>> fruit = 'banana'
>>> fruit[3:3]
''
```

String is immutable

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: object does not support item assignment

>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> print(new_greeting)
Jello, world!
```

String methods

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print(new_word)
BANANA

>>> word = 'banana'
>>> index = word.find('a')
>>> print(index)
1
```

String and the “in” operator

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False

def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print letter

>>> in_both('apples', 'oranges')
a
e
S
```

String comparison

```
if word == 'banana':
    print 'All right, bananas.'

if word < 'banana':
    print 'Your word,' + word + ', comes before banana.'
elif word > 'banana':
    print 'Your word,' + word + ', comes after banana.'
else:
    print 'All right, bananas.
```

List is a sequence

[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']

Mixed data types and nested list
['spam', 2.0, 5, [10, 20]]

Empty list
a = []

The “in” operator also works on lists
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False

Lists are mutable

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print numbers
[17, 5]
```

Traversing a list

```
for cheese in cheeses:
    print(cheese)
```

Access list via index

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

What happen to nested list?

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

Lists and operators

‘+’ to concatenate two lists together

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
```

‘*’ multiply to duplicate list number of times

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Lists slices

Use start index and stop index to slice

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

Omitting index will default the index to 0 or end

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Replacing slice

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print(t)
['a', 'x', 'y', 'd', 'e', 'f']
```

Lists methods

Appending an item to list

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print(t)
['a', 'b', 'c', 'd']”
```

Appending or extending a list

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print(t1)
['a', 'b', 'c', 'd', 'e']”
```

Sorting a list

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print t
['a', 'b', 'c', 'd', 'e’]
```

Map, Filter and Reduce

```
def add_all(t):
    total = 0
    for x in t:
        total += x
    return total

>>> t = [1, 2, 3]
>>> sum(t)
6
```

Deleting elements in lists

Remove the element at end of a list

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print(t)
['a', 'c']
>>> print(x)
b
```

Deleting an index

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print t
['a', 'c']
```

Removing a value from list

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
```

Removing a range from list

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
```

Aliasing

```
>> a = [1, 2, 3]
>>> b = a
>>> b is a
True

>>> b[0] = 17
>>> print a
[17, 2, 3]
```

Lists arguments

```
def delete_head(t):
    del t[0]

>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print letters
['b', 'c']
```

It is important to distinguish between operations that modify a list and operations that creates a new list

```
def mistaken_delete_head(t):
    t = t[1:]
```

Dictionaries

```
>>> eng2sp = dict()
>>> print eng2sp
{}

>>> eng2sp['one'] = 'uno'
>>> print eng2sp
{'one': 'uno'}

>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
>>> print eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}

>>> print eng2sp['two']
'dos'

>>> print eng2sp['four']
KeyError: 'four'
```

The ‘in’ operator and dictionaries

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False

>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

Counting using dictionaries

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d

>>> h = histogram('brontosaurus')
>>> print h
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

Traversing dictionaries

```
>>> h = histogram('parrot')
for c in h:
        print c, h[c]
a 1
p 1
r 2
t 1
o 1
>>> h.values()
dict_values([1, 1, 2, 1, 1])
```

Tuples

```
>>> t = 'a', 'b', 'c', 'd', 'e'
>>> t = ('a', 'b', 'c', 'd', 'e')
```

Tuple with one element use comma or enclosed

```
>>> t1 = 'a',
>>> type(t1)
<type 'tuple’>

>>> t2 = ('a')
>>> type(t2)
<type 'str'>

>>> t = tuple()
>>> print t
()

>>> t = tuple('lupins')
>>> print t
('l', 'u', 'p', 'i', 'n', 's’)
```

Tuples works like lists

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print t[0]
'a'

>>> print t[1:3]
('b', 'c')
```

Tuples are immutable

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment

>>> t = ('A',) + t[1:]
>>> print t
('A', 'b', 'c', 'd', 'e')
```

Swap a variable using Tuple

```
>>> temp = a
>>> a = b
>>> b = temp

>>> a, b = b, a

>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

Tuples in return values

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')

>>> t = divmod(7, 3)
>>> print t
(2, 1)

>>> quot, rem = divmod(7, 3)
>>> print quot
2
>>> print rem
1

def min_max(t):
    return min(t), max(t)
```

Variable length arguments

```
def printall(a, b, *args):
    print args

>>> printall(99, 99, 1, 2.0, '3')
(1, 2.0, '3')

>>> t = (7, 3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1

>>> divmod(*t)
(2, 1)
```

Tuples and lists

```
>>> s = 'abc'
>>> t = [0, 1, 2]
>>> zip(s, t)
[('a', 0), ('b', 1), ('c', 2)]

>>> zip('Anne', 'Elk')
[('A', 'E'), ('n', 'l'), ('n', 'k')]

t = [('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
    print number, letter

for index, element in enumerate('abc'):
    print index, element
0 a
1 b
2 c
```

Tuples and dictionaries

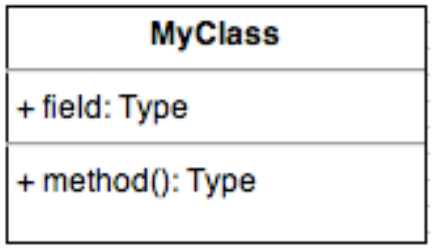
```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> print t
[('a', 0), ('c', 2), ('b', 1)]

>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> print d
{'a': 0, 'c': 2, 'b': 1}

>>> d = dict(zip('abc', range(3)))
>>> print d
{'a': 0, 'c': 2, 'b': 1}

for key, val in d.items():
    print val, key
```

```
directory[last,first] = number

for last, first in directory:
    print first, last, directory[last,first]
```

Defining a class

- Class definition

```
class MyClass:
    pass
```



- Becomes an object when instantiated

```
>>> a = MyClass()
>>> a
<__main__.MyClass object at 0x10cc08090>
```

- You may instantiate multiple objects as needed

```
>>> a = MyClass()
>>> b = MyClass()
```

Structure of class

- Methods: add, remove
- Special methods: __init__, __del__
- Attributes: name, version
- Private attributes: _cell
- Class attribute: numOfCell

```
class Library:
    numOfCell = 0

    def __init__(self, name, version):
        self.name = name
        self.version = version

        # init a dict to keep all store entries
        self._cell = {}

        Library.numOfCell += 1

    def __del__(self):
        Library.numOfCell -= 1

    def add(self, cell, name):
        self._cell[name] = cell

    def remove(self, name):
        del self._cell[name]
```

Class instances

- Use the class name to construct object which in turn invokes __init__ internally
- Use the dot (.) operator to access object attributes and to invoke methods on object

```
# create a few libraries

# this invokes Library.__init__('ckt1.net', 1)
a = Library("ckt1.net", 1)
b = Library("ckt2.net", 3)


# accessing attributes

# prints 'ckt1.net' and 1
print(a.name, a.version)

# invoke some methods
a.add(someCell)
b.remove(cellName)
```

Class versus Object

- Class is where you express the specification or common behaviors for an object
- Object are runtime instances of a class. Every object is unique and different so is the trees in a forest.

Scoping rules

- Use *self* to access attributes in the object
- Self tracks the current object of class under processing

```
(Pdb) p self
<__main__.Library object at 0x1100cd610>
(Pdb) p self.__dict__
{'_cell': {}, 'version': 1, 'name': 'ckt1.net'}
```

```
class Library:
    …
    def hasCell(self, name):
        return name in self._cell

    def add(self, cell, name):
# if hasCell(name) gets error
# NameError: global name 'hasCell' is not defined
        if self.hasCell(name): # This works!
            print("Warning: cell with name %s
already exists! replacing..." % name)
        self._cell[name] = cell

    def remove(self, name):
        if not self.hasCell(name):
            print("Error: can't find any cell name
with %s to remove" % name)
            return
        del self._cell[name]
```

Properties

```
import math

class Circle:
    def __init__(self, radius):
        self.radius = radius

    @property
    def area(self):
        return 2*math.pi*self.radius**2

    @property
    def perimeter(self):
        return 2*math.pi*self.radius
```

```
>>> a = Circle(3)
>>> a.radius
3
>>> a.perimeter
18.84955592153876
>>> a.area
56.548667764616276
```

Enable methods to be accessed like natural attribute

Properties getters and setters

```
class Unnameable:
    def __init__(self, name):
        self.__name = name

    @property
    def name(self):
        return self.__name

    @name.setter
    def name(self, value):
        assert isinstance(value, str)
        self.__name = value

    @name.deleter
    def name(self):
        raise TypeError("can't delete name")
```

```
>>> a = Unnameable('benny')
>>> a.name
'benny'
>>> a.name = "khoo"
>>> a.name
'khoo’
>>> del(a.name)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "property.py", line 32, in name
    raise TypeError("can't delete name")
TypeError: can't delete name
```

Everything is object

- In Python everything that is nameable (including names of classes and functions, literals) can be referred to and processed like regular data in your code.
- *Demo*

Exceptions

```
def foo():
    …
    # error condition
    return -1
```

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

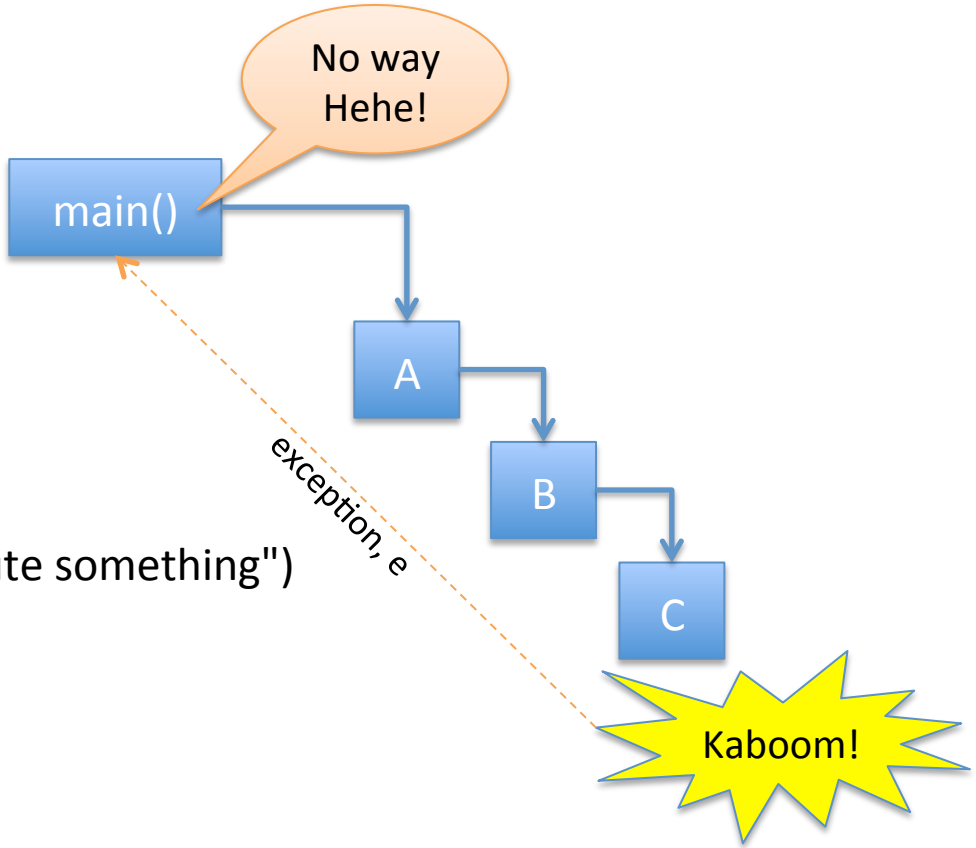
- Traditional error handling via return value
- Exception changes the course of normal execution flow
- Program can be designed to response to exception with alternate program flow or logic typically to recover from error condition

Python and exceptions

- Python itself has been practicing exception throughout: -
  - SyntaxError raised when source code has invalid syntax
  - KeyError raised when attempt to access dictionary with non-existent key. To test a key properly use: *x in container*
  - For complete list
- App may choose to raise built-ins exception without reinventing new one (for clarity)

Handling exception

```
def A():
    return B()

def B():
    return C()

def C():
    # error 1
    raise KeyError('key error')
    # error 2
    raise OverflowError("can't compute something")
try:
    A()
except OverflowError as e:
    # handle overflow error here
# empty or more exception …
except: # leave out the argument for “catch-all”
   # handle every else here
finally:
   # clean-up code
```



Unhandled exception

```
Traceback (most recent call last):
  File "/Users/bennykhoo/Sources/MyPythonClass/except.py", line 15, in <module>
    A()
  File "/Users/bennykhoo/Sources/MyPythonClass/except.py", line 4, in A
    return B()
  File "/Users/bennykhoo/Sources/MyPythonClass/except.py", line 7, in B
    return C()
  File "/Users/bennykhoo/Sources/MyPythonClass/except.py", line 10, in C
    raise KeyError('key error')
KeyError: 'key error'
```

- You will get stack trace which can be useful for debug
- What is your opinion?

Exception is just ordinary object

- Exception is just a regular of class with a constructor
- Always derive from base Exception class or below to create custom exception class just like ordinary OOP

- Good use cases if you want to carry other error info as attributes
- Keep it simple

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print('My exception occurred, value:', e.value)
...
My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

Finally …

- The finally block in try statement
- Must be executed under all circumstances whether exception has occurred or not
- Good use for handling clean-up
  - Closing hardware handles which occupied system resources
  - Critical exit point such transaction exit in security application
  - Releasing locks in multi threaded code

The with statement

```
# old style
f = open('debuglog', 'a')
f.write("debugging\n")
statements
exception occur!!
f.close()
```

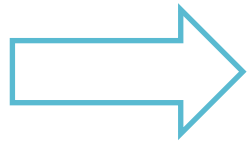
```
with open('debuglog', 'a') as f:
    f.write("debugging\n")
    statements
    f.write("done\n")
# file f is guaranteed closed
```

```
import threading
lock = threading.lock()
with lock:
    # critical section
    statements
    # end critical section
# lock is guaranteed released
```

- Managing system resources can be tricky biz
  - Error condition can cause control flow to bypass statement responsible for proper releasing of resources
- A form of defensive programming pattern [link]

With and without

```
with VAR = EXPR:
    BLOCK
```



roughly equivalents to

```
VAR = EXPR
VAR.__enter__()
try:
    BLOCK
finally:
    VAR.__exit__()
```

- A short hand idiom for try-except-finally block
- Magic methods __enter__ and __exit__ for context management protocol

File IO

- *Demo session*

Modules and packages

- Module provides a namespace for your functions, classes and global vars on per file basis
- Use import to introduce the module to your current module namespace
- Use dot naming convention to reference a module namespace
- Use *from … import …* to exclude child module name
  - Use specific names to cherry pick names to import.
  - Use asterisk (*) to refer to every available names in a importing module.
- Convenient way to test your module using __name__
- Python standard modules and *sys.path*

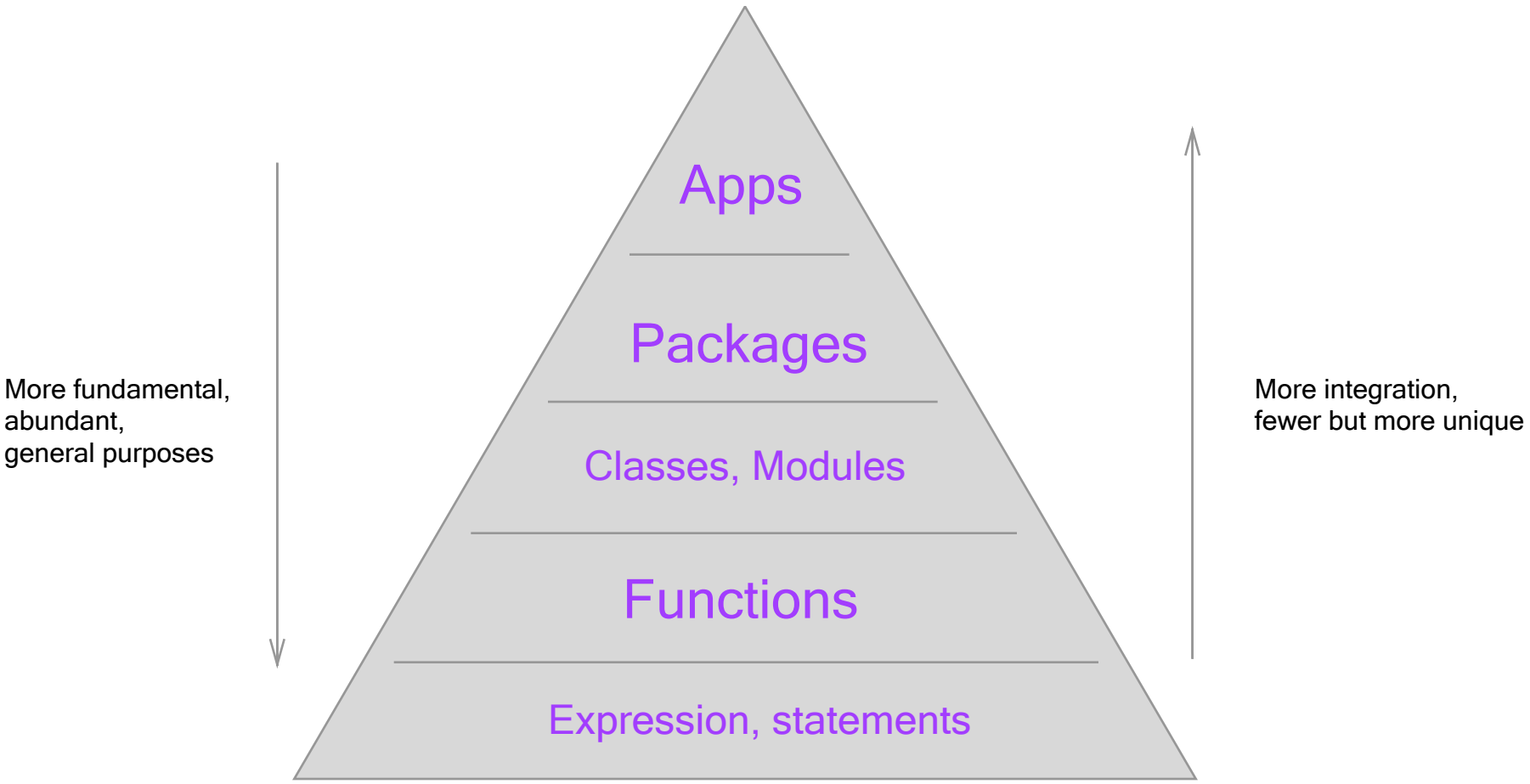
Package

- Package provides a logical grouping of modules.
  - Organize your module files in a folder. The container folder becomes the package name
  - Don’t forget to include a bootstrap file __init__.py which is necessary for Python to locate a package
  - Works like module. Package honors “import …” and “from ... Import ...” rules albeit with minor exceptions

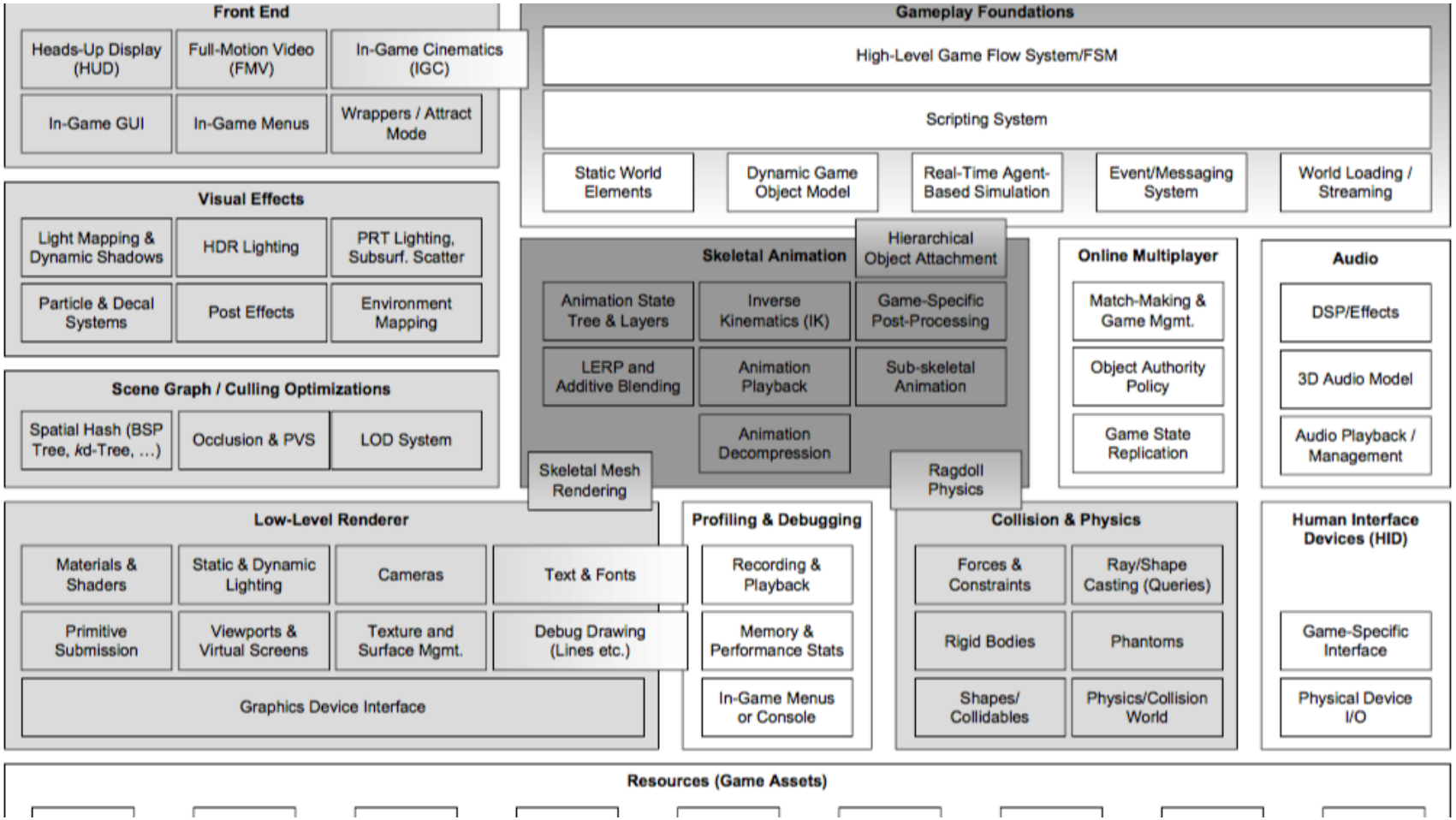
Packages, modules, classes and functions – Where do they fit?

- Functions – to encapsulate a group of logics: - algorithms, procedure, mathematical function. E.g. sort, gcd etc.
- Classes, Modules – a unit of logical grouping of functions that define the behavior of an object. E.g. Characters in COC: damage per second, hit points, movement speed, special abilities.
- Packages– represent a logical subsystem in a larger system (application or program). E.g. in games 2d, 3d renderer, audio, network, physics.

Hierarchy of needs

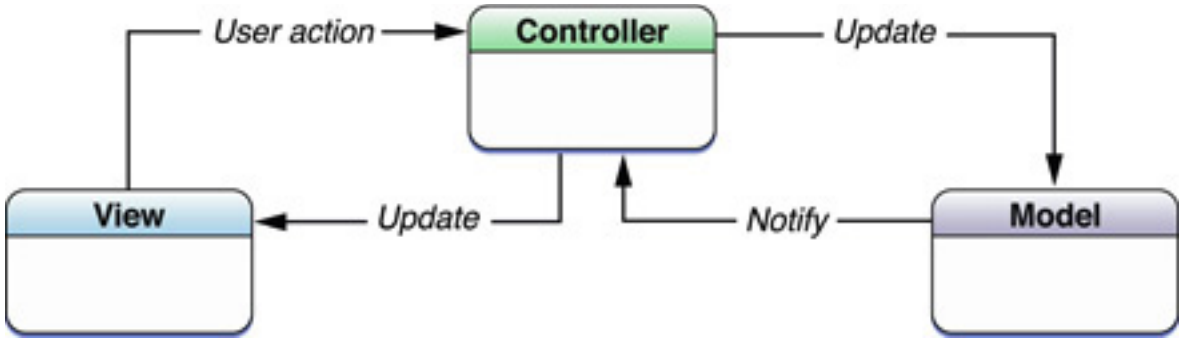


How does a real world system looks like?



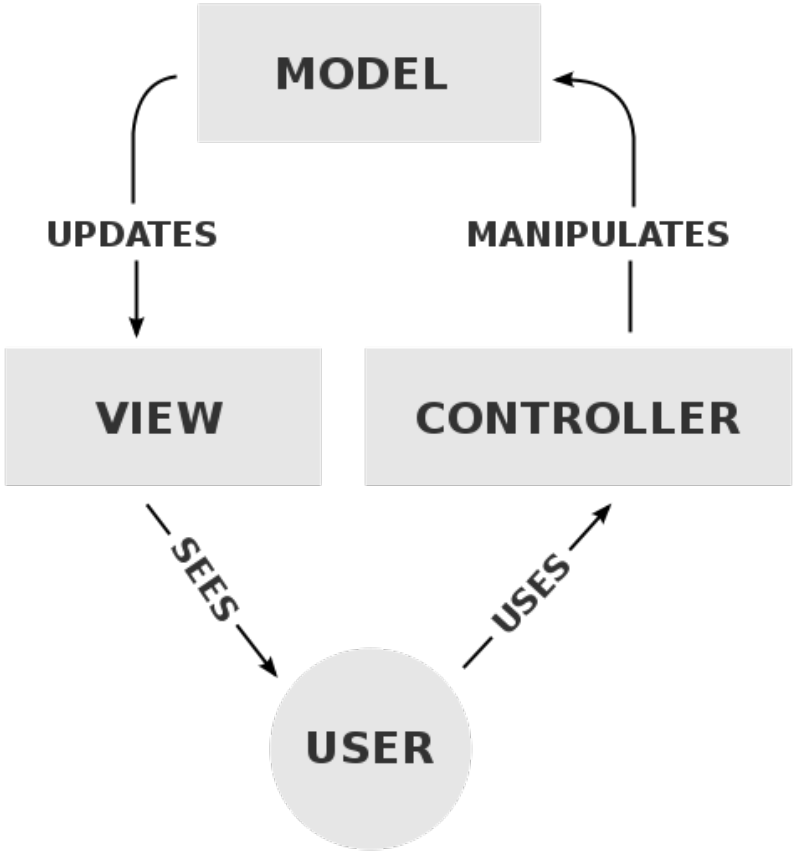
* Courtesy of Game Engine Architecture 2nd Ed. [1]

Model view controller



Apple iOS doc [1]

Model view controller (2)

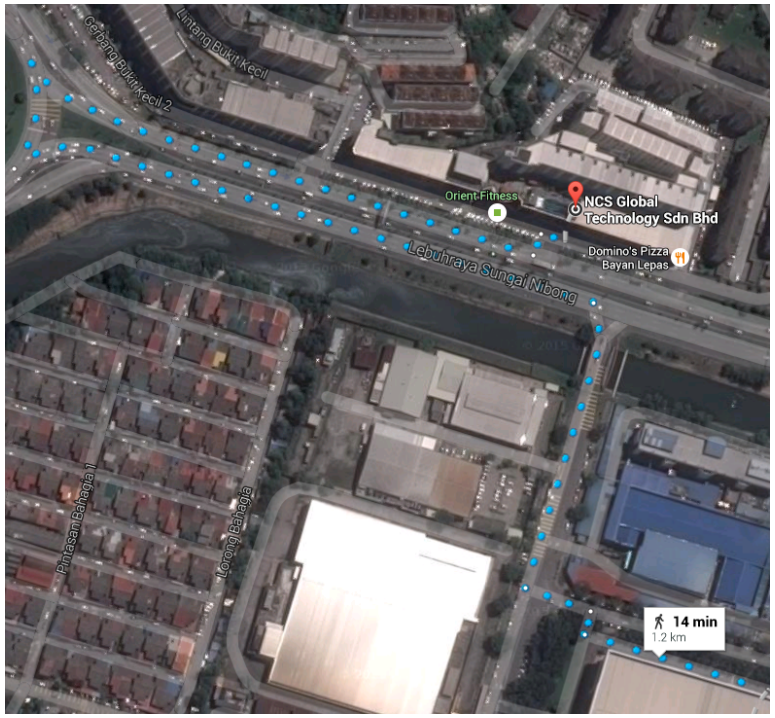


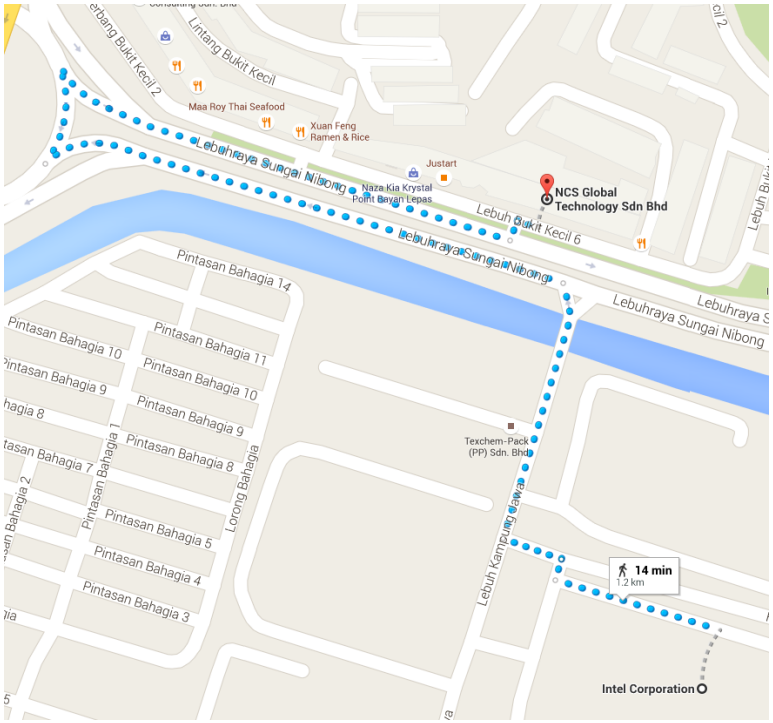
Wikipedia [1]

Model

- Main role is to host data to other components in MVC.
- Usually feature some DB but can be anything that host the data source. Data can be derived, computed via algorithms and mathematical model.

The map is not territory





- Model is a kind of simplification or estimation to complexity (real world).
- Challenges for software engineers is to be able to formulate an abstract problem onto some model that can be computed by computers.
- Models can be creative too like other artistic disciplines.
- In other disciplines: models usage are crucial to help understanding, diagnosis and prediction: - weather forecast system, semiconductors and neurologists[1]

View

- Users (your boss included) of your app may want to view their data differently (list or grid tabulation, charts etc.) .
- There may be further constrained by platform and devices availability (desktop, phones and different OSes).
- For consumer apps – may contains other aesthetical challenges (design, graphics, motions etc.)

Controller

- Intermediary (glue logics) that bind together view and model.
- Translate user responses via View to changes in Model.
- Translate changes from Model to changes in View.
- May response to OS events. For instance in mobile, low battery, orientation change and so on.

MVC - summary

- Model hosts the data. May contains logic of the data, entities relationship.
- View – the view that users see. Projection of model and enable manipulation of model by users.
- Controller – intermediary between view and model.
- Separation of concerns – What needs to be changed when porting an iOS apps to Android?

Planning your project

1. Identify a list of classes needed to complete the requirements. Sketch or list it down on paper.
2. Identify behavior of each class.
   - Try to list out the attributes in each of this class.
   - Try to list out methods in respect to each class.
   - Word it out why do you need them.
   - Identify potential interaction between each of those classes. How is the relationship between each classes like (e.g. 1 to 1, 1 to many)?
3. Dry testing your abstract plan. Break down and itemize it to a list of tasks.
   1. Try to visualize how does your plan (listed in #1 and #2) going to work to accomplish each of those task you have just listed?
   2. Word out the steps.

Putting your plan to work

- Start implementing your plan in Python. See how far you can get.