Feb 10th, 2018
Kelly Ryu

# Week 3 Computer Programming in Python (Accelerated)

**Today:**

I. Functions
   A. Defining and Using Functions
   B. Concept of encapsulation
   C. Variable length parameters
   D. Recursion
II. More datatypes
   A. Dictionary
   B. Set
   C. Tuple

## I. Defining functions

```
def functionName(any parameters):
    what to do in function

    return something
```

### A. Module

- A set of text files available to use that contains definitions of functions
- Does nothing demonstrable to us in the console when run by itself
- In order to use the module, we write a file with a main function that imports modules

e.g. `math` module has constants and functions → pi, e, etc…

- A function always has () at end, whether or not they need the input
- eg. `random.random()`
- Constants do not have input.. no `()`
- Without importing `math` module in the beginning, python can't read `math.pi`

- We can write our own modules and import them (important to have them in the same folder!)
- main function file should be the coarsest iteration of your algorithm
  eg. for coding a game, [ 1. Display greeting, 2. Play game, 3. Say goodbye]
- And make sure main function file imports all modules needed
- In main function file, you can change a long module name to a short one by declaring "`import ___ as __`" when importing

e.g.

```
import circle as c
```

```
radius = 13
area = c.area(radius)
```

B. **Why do we use functions so much**
   1) Division of labor
   2) Debugging
      - Easier to check where it went wrong
   3) Readability
   4) Design
   5) Reusability
      - Main function is the part that is not reusable, that's why we want it to be as small as possible

- Does not matter which order definitions of functions come in the code when functions call each other
- **encapsulation:** Packing the details into one module or a function
    - In the main function, you just know it's getting done without knowing how; Details are in the function or module

**C. Writing functions with variable parameters**

**1)** Up to now, we only used **positional parameters**: if I expect two arguments, the user must give me two

**2) Keyword parameters (default argument)**: When you define a function, you can set a default value if there is no argument in the parameter position

e.g.
```
def f(a, b=3):
    …
```
- The parameter at b has a default value of 3 so it is okay to call f(2) or f(4) instead of using two arguments such as f(2,4)
- So when a second argument exists in a function call, it overwrites the default parameter b = 3
- But when there is no second argument, b parameter is set to default 3

**3) Variable length positional arguments** (`*`):
the user can give as many arguments as he/she wants to pack into a tuple

- eg. `f(a,b,*c)`: if the user puts in `f(3,6,12,15,'happy', 87), 3` gets put into `a, 6` gets put into `b,` and tuple (`12,15,'happy',87)` goes into c (not *c, just c)
- * signals for the compiler to automatically pack anything after that position (inclusive) into a tuple when user inputs the arguments

**4) Variable length keyword arguments** (`**`)**:** the user can specify as many keyword-value pairs as he/she wants to pack into a dictionary

- keyword should be written with only alphabets *without any quotation marks*

e.g.
```
def cool(a,b=3,*args,**Kwargs):
    print a
```

```
    print b
    print args
    print Kwargs

def main():
    cool (4,5,'hi','Kelly','python',bob=4,jenny=2)
          #overwrote b=3 to 5
          #every positional argument after that is in args
          #from first keyword argument to last is in Kwargs
main()
```

→ Results:

```
4
5
('hi', 'Kelly', 'python')
{'bob': 4, 'jenny': 2}
```

- **Important tips:**
    - the order of parameters is crucial! (positional, keyword, *, and then **) → do not switch order (at least in python2.7)
    - We were able to set default values for keyword arguments when we only had positional and/or keyword arguments in the function definition
    - When a **variable length positional** parameter follows, however, the **keyword arguments must be specified** in the function invocation (if not, it will take the first argument intended for the tuple as the keyword argument)
    - When a **variable length keyword** parameter follows directly, the situation is the same as before (the default value will be set without argument specification)
    - Why do you think this is the case? The start of the variable length keyword arguments is pretty clear-cut. It has a *string without quotes = value* format. So there's no way something like that is invoked into a keyword parameter. But what about arguments intended for the variable length positional arguments? They can be anything (string, another tuple, list, int, etc.), so the compiler can't tell if the first element is supposed to override the default keyword parameter before that, or if it's actually the first element of the tuple. Thus, you must make sure you have the keyword argument specified in this case.

for eg.

```
def cool(a,b=3,**Kwargs):
                                #variable length keyword parameter
                                #directly follows keyword argument
    print a
    print b
    print Kwargs

def main():
    cool (4,bob=4,jenny=2) # note parameter b is not specified
```

```
main()
```

Results:

```
4
3  # b default value is printed
{'bob': 4, 'jenny': 2}
```

**D. Recursive functions**

1) RULES OF RECURSION!
    1. Must have a simple base case (always write this first when coding recursive functions)
    2. Subproblems should be similar to big problem, but smaller
    3. Operations should not overlap

2) e.g. Recursive function to reverse strings

```
def reverse(s):
    if len(s) == 1:
        return s        # base case
    else:
        new_string = s[len(s)-1] + reverse(s[0:len(s)-1])
        return new_string
```

## II. More datatypes

### A. Dictionaries (`dict`)

*var1* = {*key1*: *value1, key2*: *value2, …*}

- **KEY** must be an immutable type (`str, tuple, int, float, double`)
- Therefore, key can**NOT be a list**
- values can be anything (no need to match types between keys and values)
- It *is* possible to alternate between different types within keys and values, but difficult to do anything with them if their types differ randomly
- What is this good for?
    - associating sets of values together (names with ages, students with grades, etc.)
    - no need to index in order... index by any order I want
    - look things up in a collection by key (not by index)
- Can set/modify values in dictionaries (dictionaries themselves are mutable)

e.g.

```
In: grades = {'frank':'c', 'cindy':'a'}
```

```
In: grades['frank']
Out: 'c'
In: grades['frank'] = 12 #modify value to a different type
In: grades['frank']
Out: 12
```
** note we modified the type of the value without any problems

```
In: grades[0] = 'IDK'
In: grades
Out: {0: 'IDK', 'frank': 12, 'cindy': 'a'}
```

** We've added a new pair of key and value; The order of keys when printed is pretty random; doesn't matter because we access values by key anyway

- Can have multiple keys with same value
- But not one value with multiple keys
- Can iterate through both keys and values of dictionaries

e.g.
```
In: for key in grades:
          print(grades[key])

Out: 'IDK'
     12
     'a'
```

e.g.
```
d = {'johnny': 6, 'sally': 7, 'lisa': 5}

In: d.items()
Out: dict_items([('sally', 7), ('lisa', 5), ('johnny', 6)])


In: d.keys()
Out: dict_keys(['sally', 'lisa', 'johnny'])
In: d.values()
Out: dict_values([7, 5, 6])
```

e.g. error-proof way to retrieve values from keys

```
a = {'johnny': 6, 'sally': 7, 'lisa': 5}
In: a.get('bobby','notfound')
Out: 'notfound' # returns second argument if first argument
                    is not a key in dict a
In: a.get('lisa', 'not found')
Out: 5   # correctly returns value of key 'lisa' since it exists in a
```

**B. Sets (`set`)**

- cannot start with an empty set because it recognizes it as an empty dictionary
- instead use `set()` for empty set

- sets have **no repeat and no order**
- the set has some random order and erases repetitions, while list keeps everything the way we wrote it
- So we cannot order sets

e.g.
```
In: s = {3,4,5,6,3}
In: s
Out: {3, 4, 5, 6} #automatically got rid of the second 3
```

e.g. can also use `in` operator
```
In: 3 in s
Out: True
```

- **Union**: `s.union(s1), s|s1`

e.g.
```
In: s = {5,4,4,10, 'happy', 'sad'}
In: s1 = {4,2,1,'joy'}
In: s | s1
Out: {1, 2, 4, 5, 'happy', 10, 'joy', 'sad'} ##note no sense of order
```

- **Intersection**: `s.intersection(s1) , s&s1`

e.g.
```
In: s.intersection(s1)
Out: {4}
```

- **Difference**: `s-s1 , s.difference(s1)` (here the order matters because s-s1 != s1-s)
  - Takes out the s&s1 elements from s

e.g.
```
In: s-s1
Out: {'sad', 10, 5, 'happy'}
```

- **Symmetric difference**: `s^s1`, `s1.symmetric_difference(s)`
  - union of the differences = everything in the union that is not an intersection
  - e.g.

```
In: s1^s
Out: {1, 2, 5, 'happy', 10, 'joy', 'sad'}
```

- **Subset/Superset**: `s1 <= s` (this is testing whether s1 is a subset of s)
- **Strict subset/superset**: `< , >` (it's testing whether it's a subset but not the set itself so it's actually testing two booleon statements)

- `s.discard(something)`: remove an element from a set (doesn't return an error when the element's not in there)
- Useful tip: if we make set of a list and check the length, we get the number of distinct elements in the list

C. **Tuple (`tuple`)**

- an ordered sequence of immutable objects
- tuples are immutable unlike lists

**e.g. creating tuples**

```
In: tup1 = ("1","3",45,1,"4")
In: tup1
Out: ('1', '3', 45, 1, '4')
```

e.g. Accessing elements in tuples

```
In: tup1[3]
Out: 1
```

e.g. Tuples are immutable, so you can't change them but you can create a new tuple from one

```
In: tup1[1:3]
Out: ('3', 45)
```

So you can't do something like `tup1[0] = 100`
or delete certain elements of a tuple → you should always make a new one with the unwanted ones discarded

e.g.

```
tup1 = (1,2,3,4,5)
tup2 = (tup1[0],tup1[2],tup1[4])
print tup2

tup3 = tup1[0:5:2] #get every 2nd element from index 0 to 5(noninclusive)
print tup3

#both print statements lead to same result:
(1, 3, 5)
(1, 3, 5)
```

**e.g.** Can do things we did in lists

```
len((1,2,3)) → 3
(1,2,3)+(4,5,6) → (1,2,3,4,5,6)
('hello')*4 → ('hello','hello','hello','hello')
3 in (4,5,6)→ false
for x in (4,5,6)
```

e.g. to make tuple with one element, must have comma after it

```
x = ("1",)
```

Feb 10th, 2018
Kelly Ryu

**<Built-in functions for tuples>**

`cmp(tup1,tup2)` → returns 0 if all elements of tup1 are in the exact same order as tup2
→ returns -1 if not

`max(tuple)` → finds min of tuple
`min(tuple)` → finds max of tuple
`len(tuple)` → returns length of tuple
`tuple(list)` → converts list to tuple