Handout 3: Functions, Conditions, Iteration

Functions

1. Calling a function

a. The value of the name is the function itself

```
>>> len
<built-in function len>
```

b. One **calls** the function by giving it **arguments** (inputs)

```
>>> len('hi')
2
```

c. Some functions take more than one argument: `percentage(4,5)`

d. Some can be called with no arguments:

```
>>> list()
[]
```

e. There are two kinds of parentheses in the following. Explain. Draw the parse tree.

```
>>> 2 * (len('cat' + 's') + 1)
```

2. Defining functions

a. Example: computing percentage

```
>>> def percent (k, n):
...     return (k / n) * 100
...
>>> n = 2
>>> percent(n, len('abcde'))
40.0
```

b. General form:

`def` *name* `(`*param, param, ...*`):`
    `return` *value*

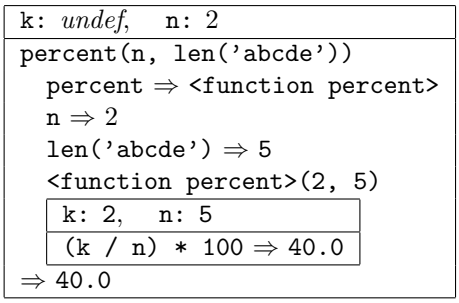
3. *Exercises*

a. Define a function called `wordlist` that takes a list of word tokens and returns an alphabetic list of the distinct words that occur.

b. Define a function called `f` implementing

$$f(x) = 2x + 1$$

4. How Python evaluates a function-call expression.



5. Functions are first-class objects

   a. Defining a function creates the object and sets the value of the name

```
>>> percent
<function percent at 0x1003bebf8>
>>> f = percent
>>> f(4, 8)
0.5
```

   b. Can have complex expression in function position

```
>>> lst = [len, percent, 42]
>>> lst[1](1,5)
0.2
```

Objects and methods

6. An object contains named **members**

   a. Syntax: *object*.*member*

   b. For example, a module is an object. Create a file `foo.py` containing "`x = 2 + 2`."

```
>>> import foo
>>> foo.x
4
```

   c. A **method** is a member whose value is a function.

```
>>> text1.count
<bound method Text.count of <Text: Moby Dick by Herman Melville 1851>>
>>> f = text1.count
>>> f('whale')
906
```

   d. Note that the method "knows" which object it belongs to: we get the count in *Moby Dick,* not in some other text.

e. *Everything* in python is an object

7. Classes and methods

a. Every object has a **class**. Also called a *type.*

```
>>> x = ['Mary', 'Beth']
>>> type(x)
<class 'list'>
>>> s = 'Mary Beth'
>>> type(s)
<class 'str'>
```

b. Which methods an object has depends on its class.

```
>>> s.lower
<built-in method lower of str object at 0x10063ddb0>
>>> x.lower
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'lower'
```

c. To find out what members an object has:

```
>>> dir(s)
['__add__', ..., 'lower', ..., 'upper', 'zfill']
```

d. To get information about a method:

```
>>> help(s.lower)
Help on built-in function lower:

lower(...) method of builtins.str instance
    S.lower() -> str

    Return a copy of the string S converted to lowercase.
```

(Press 'q' to get out.)

String and list methods

8. String methods: see Table 4.2 in Ch1(4.1) and Table 3.2 in Ch3(3.2).

```
s.islower()      t = s.lower()       i = s.find(t)
s.isupper()      t = s.upper()       i = s.index(t)
s.istitle()      t = s.title()       i = s.rfind(t)
s.isalpha()      t = s.strip(c)      i = s.rindex(t)
s.isalnum()      t = s.rstrip(c)
s.isdigit()      t = s.replace(c,d)
s.startswith(t)  lst = s.split(sep)
s.endswith(t)    s = sep.join(lst)
```

9. List methods: `find`, `index`, `rfind`, `rindex`.

Iteration

10. Lowercasing a string:

```
>>> 'Mary Beth'.lower()
'mary beth'
```

11. **Iteration:** do it *for each word* in a list:

```
>>> for w in ['Just', 'Testing']:
...     print(w.lower())
...
just
testing
```

12. **Blocks.**

a. Example

```
>>> for w in ['Just', 'Testing']:
...     print('---', end='')
...     print(w.lower(), end='')
...     print('---')
...
---just---
---testing---
```

b. Python CARES ABOUT WHITESPACE. Indentation being off will cause an error!

c. Don't forget the colon

13. `for ... in` works with anything that contains elements

a. `for n in [10, 20, 30]:`

b. `for elt in {'hi', 'there'}:`

c. `for c in 'test':`

14. What if we want to iterate over the numbers 1, 2, 3, etc.?

a. A **range** generates a sequence of numbers:

```
>>> for n in range(1,4):
...     print(n)
...
1
2
3
```

b. Unlike a list, it generates the numbers as they are called for, does not allocate memory.

c. Example: iterating through two lists in parallel. (Omitted first number taken to be 0.)

```
for i in range(len(lst1)):
    elt1 = lst1[i]
    elt2 = lst2[i]
    ...
```

Conditional

15. Introduced by `if`

a. Example. Set `n = 1`

```
>>> if n == 0:
...     print('none')
... elif n == 1:
...     print('one')
... else:
...     print('multiple')
...
one
```

b. The `elif` and `else` parts are optional.

c. Any expression with a value can be used after `if`. The expression counts as false if it evaluates to `False`, 0, or an empty object. Otherwise it counts as true.

16. Combined with iteration

```
>>> for w in ['dogs', 'and', 'cats']:
...     if w.endswith('s'):
...         print(w)
...
dogs
cats
```

List comprehension

17. Example

a. Set `words = ['Just', 'Testing', '!']`

b. Suppose we want to lowercase our words. One way:

```
>>> out = []
>>> for w in words:
...     out.append(w.lower())
>>> out
['just', 'testing', '!']
```

c. A cleaner alternative:

```
>>> [w.lower() for w in words]
['just', 'testing', '!']
```

18. Getting rid of the exclamation point in `words`

a. Collecting into list:

```
>>> out = []
>>> for w in words:
...     if w.isalpha():
...         out.append(w.lower())
>>> out
['just', 'testing']
```

b. The list comprehension:

```
>>> [w.lower() for w in words if w.isalpha()]
['just', 'testing']
```

19. List comprehension producing generator

a. Produces a list: `[w.lower() for w in words]`

b. Produces a generator: `(w.lower() for w in words)`

c. If a generator is the only argument to a function, you can drop the extra parentheses

```
>>> v = set((w.lower() for w in words))
>>> v = set(w.lower() for w in words)
```

20. Differences between a list and a generator:

a. The *only* thing you can do with a generator is iterate through it

```
>>> g = (w.lower() for w in words)
>>> g[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'generator' object is not subscriptable
```

b. The list allocates memory for all elements. The generator does not.

c. You can only iterate through a generator once; then it is empty.

```
>>> for w in g:
...     print(w)
...
just
testing
!
>>> for w in g:
...     print(w)
...
>>>
```