DM550/DM857
Introduction to Programming

Peter Schneider-Kamp

petersk@imada.sdu.dk

http://imada.sdu.dk/~petersk/DM550/

http://imada.sdu.dk/~petersk/DM857/

HANDLING TEXT FILES

Reading Files

- open files for reading using the open(name) built-in function
  - Example: f = open("anna_karenina.txt")
- return value is file object in reading mode (mode 'r')
- we can read all content into string using the read() method
  - Example:

```
content = f.read()
print(content[:60])
print(content[3000:3137])
```

- contains line endings (here "\r\n")

Reading Lines from a File

- instead of reading all content, we can use method readline()
  - Example:
    ```
    print(f.readline())
    next = f.readline().strip()
    print(next)
    ```
- the method strip() removes all leading and trailing whitespace
- whitespace = \n, \r, or \t (new line, carriage return, tab)

- we can also iterate through all lines using a for loop
  - Example:
    ```
    for line in f:
        line = line.strip()
        print(line)
    ```

Reading Words from a File

- often a line consists of many words
- no direct support to read words
- string method split() can be used with for loop
  - Example:

```
def print_all_words(f):
    for line in f:
        for word in line.split():
            print(word)
```

- variant split(sep) using sep instead of whitespace
  - Example:

```
for part in "Slartibartfast".split("a"):
    print(part)
```

Analyzing Words

- Example 1: words beginning with capital letter ending in "a"

```
def cap_end_a(word):
    return word[0].upper() == word[0]
```

Analyzing Words

- Example 1: words beginning with capital letter ending in "a"

```
def cap_end_a(word):
    return word[0].upper() == word[0] and word[-1] == "a"
```

Analyzing Words

- Example 1: words beginning with capital letter ending in "a"

```
def cap_end_a(word):
    return word[0].isupper() and word[-1] == "a"
```

- Example 2: words that contain a double letter

```
def contains_double_letter(word):
    last = word[0]
    for letter in word[1:]
        if last == letter:
            return True
        last = letter
    return False
```

Analyzing Words

- Example 1: words beginning with capital letter ending in "a"

```
def cap_end_a(word):
    return word[0].isupper() and word[-1] == "a"
```

- Example 2: words that contain a double letter

```
def contains_double_letter(word):
    for i in range(len(word)-1):
        if word[i] == word[i+1]:
            return True
    return False
```

Adding Statistics

- Example: let’s count our special words

```
def count_words(f):
    count = count_cap_end_a = count_double_letter = 0
    for line in f:
        for word in line.split():
            count = count + 1
            if cap_end_a(word):
                count_cap_end_a = count_cap_end_a + 1
            if contains_double_letter(word):
                count_double_letter = count_double_letter + 1
    print(count, count_cap_end_a, count_double_letter)
    print(count_double_letter * 100 / count, "%")
```

Adding Statistics

- Example: let's count our special words

```
def count_words(f):
    count = count_cap_end_a = count_double_letter = 0
    for line in f:
        for word in line.split():
            count += 1
            if cap_end_a(word):
                count_cap_end_a += 1
            if contains_double_letter(word):
                count_double_letter += 1
    print(count, count_cap_end_a, count_double_letter)
    print(count_double_letter * 100 / count, "%")
```

Debugging by Testing Functions

- correct selection of tests important
- check obviously different cases for correct return value
- check corner cases (here: first letter, last letter etc.)
- Example:

```
def contains_double_letter(word):
    for i in range(len(word)-1):
        if word[i] == word[i+1]:
            return True
    return False
```

- test "mallorca" and "ibiza"
- test "llamada" and "bell"

LIST PROCESSING

Lists as Sequences

- lists are sequences of values
- lists can be constructed using "[" and "]"
- Example:

```
[42, 23]
["Hello", "World", "!"]
["strings and", int, "mix", 2]
[]
```

- lists can be nested, i.e., a list can contain other lists
- Example:

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

- lists are normal values, i.e., they can be printed, assigned etc.
- Example:

```
x = [1, 2, 3]
print(x, [x, x], [[x, x], x])
```

Mutable Lists

- lists can be accessed using indices
- lists are mutable, i.e., they can be changed destructively
- Example:

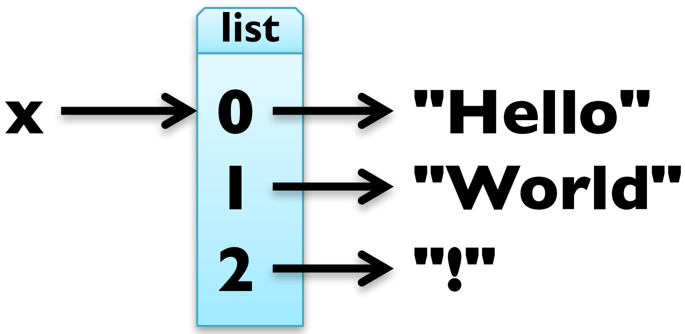
```
x = [1, 2, 3]
print(x[1])
x[1] = 4
print(x, x[1])
```

- `len(object)` and negative values work like for strings
- Example:

```
x[2] == x[-1]
x[1] == x[len(x)-2]
```

Stack Diagrams with Lists

- lists can be viewed as mappings from indices to elements
- Example 1: x = ["Hello", "World", "!"]



- Example 2: x = [[23, 42, -3.0], "Bye!"]

Traversing Lists

- for loop consecutively assigns variable to elements of list
- Example: print squares of numbers from 1 to 10

```
for x in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    print(x**2)
```

- arithmetic sequences can be generated using range function:
  - range([start,] stop[, step])
- Example:

```
list(range(4)) == [0, 1, 2, 3]
list(range(1, 11)) == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
list(range(9, 1, -2)) == [9, 7, 5, 3]
list(range(1,10, 2)) == [1, 3, 5, 7, 9]
```

Traversing Lists

- for loop consecutively assigns variable to elements of list
- general form

```
for element in my_list:
    print(element)
```

- iteration through list with indices:

```
for index in range(len(my_list)):
    element = my_list[index]
    print(element)
```

- Example: in-situ update of list

```
x = [8388608, 43980465111104, 0.125]
for i in range(len(x)):
    x[i] = math.log(x[i], 2)
```

List Operations

- like for strings, "+" concatenates two lists
- Example:

```
[1, 2, 3] + [4, 5, 6] == list(range(1, 7))
[[23, 42] + [-3.0]] + ["Bye!"] == [[23, 42, -3.0], "Bye!"]
```

- like for strings, "* n" with integer n produces n copies
- Example:

```
len(["I", "love", "penguins!"] * 100) == 300
(list(range(1, 3)) + list(range(3, 1, -1))) * 2 ==
        [1, 2, 3, 2, 1, 2, 3, 2]
```

List Slices

- slices work just like for strings
- Example:

```
x = ["Hello", 2, "u", 2, "!"]
x[2:4] == ["u", 2]
x[2:] == x[-3:len(x)]
y = x[:]        # make a copy (lists are mutable!)
```

- BUT: we can also assign to slices!
- Example:

```
x[1:4] = ["to", "you", "too"]
x == ["Hello", "to", "you", "too", "!"]
x[1:3] = ["to me"]
x == ["Hello", "to me", "too", "!"]
x[2:3] = []
x == ["Hello", "to me", "!"]
```

List Methods

- appending elements to the end of the list (destructive)
- Example: x = [5, 3, 1]
  y = [2, 4, 6]
  for e in y: x.append(e)
- Note: x += [e] would create new list in each step!
- also available as method: x.extend(y)

- sorting elements in ascending order (destructive)
- Example: x.sort()
  x == range(1, 7)

- careful with destructive updates: x = x.sort()

Higher-Order Functions (map)

- Example 1: new list with squares of all elements of a list

```
def square_all(x):
    res = []
    for e in x:    res.append(e**2)
    return res
```

- Example 2: new list with all elements increased by one

```
def increment_all(x):
    res = []
    for e in x:    res.append(e+1)
    return res
```

Higher-Order Functions (map)

- these *map* operations have an identical structure:

```
res = []
for e in x:   res.append(e**2)
return res
```

```
res = []
for e in x:   res.append(e+1)
return res
```

- Python has generic function map(function, sequence)
- Implementation idea:

```
def map(function, sequence):
    res = []
    for e in sequence:
        res.append(function(e))
    return res
```

Higher-Order Functions (map)

- these *map* operations have an identical structure:

```
res = []
for e in x:   res.append(e**2)
return res
```

```
res = []
for e in x:   res.append(e+1)
return res
```

- Python has generic function map(function, sequence)
- Example:

```
def square(x):          return x**2
def increment(x):       return x+1
def square_all(x):
    return map(square, x)
def increment_all(x):
    return map(increment, x)
```

Higher-Order Functions (filter)

- Example 1: new list with elements greater than 42

```
def filter_greater42(x):
    res = []
    for e in x:
        if e > 42:    res.append(e)
    return res
```

- Example 2: new list with elements whose length is smaller 3

```
def filter_len_smaller3(x):
    res = []
    for e in x:
        if len(e) < 3:   res.append(e)
    return res
```

Higher-Order Functions (filter)

- these *filter* operations have an identical structure:

```
res = []
for e in x:
    if e > 42:   res.append(e)
return res
```

```
res = []
for e in x:
    if len(e) < 3:   res.append(e)
return res
```

- Python has generic function filter(function, iterable)
- Implementation idea:

```
def filter(function, iterable):
    res = []
    for e in iterable:
        if function(e):     res.append(e)
    return res
```

Higher-Order Functions (filter)

- these *filter* operations have an identical structure:

```
res = []
for e in x:
    if e > 42:   res.append(e)
return res
```

```
res = []
for e in x:
    if len(e) < 3:   res.append(e)
return res
```

- Python has generic function filter(function, iterable)
- Example:

```
def greater42(x):              return x > 42
def len_smaller3(x):           return len(x) < 3
def filter_greater42(x):       return filter(greater42,  x)
def filter_len_smaller3(x):    return filter(len_smaller3, x)
```

Higher-Order Functions (reduce)

- Example 1: computing factorial using range

```
def mul_all(x):
    prod = 1
    for e in x:    prod *= e            # prod = prod * e
    return prod
def factorial(n):
    return mul_all(range(1,n+1))
```

- Example 2: summing all elements in a list

```
def add_all(x):
    sum = 0
    for e in x:    sum += e             # sum = sum + e
    return sum
```

Higher-Order Functions (reduce)

- these *reduce* operations have an identical structure:

```
prod = 1                                sum = 0
for e in x:    prod *= e                for e in x:    sum += e
return prod                             return sum
```

- Python has generic function functools.reduce(func, seq, init)
- Implementation idea:

```
def reduce(func, seq, init):
    result = init
    for e in seq:
        result = func(result, e)
    return result
```

Higher-Order Functions (reduce)

- these *reduce* operations have an identical structure:

```
prod = 1                          sum = 0
for e in x:   prod *= e           for e in x:     sum += e
return prod                       return sum
```

- Python has generic function functools.reduce(funct, seq, init)
- Example:

```
def add(x,y):   return x+y
def mul(x,y):   return x*y
def add_all(x):
    return reduce(add, x, 0)
def mul_all(x):
    return reduce(mul, x, 1)
```

Deleting Elements

- there are three different ways to delete elements from list

- if you know index and want the element, use pop(index)
- Example:
```
my_list = [23, 42, -3.0, 4711]
my_list.pop(1) == 42
my_list == [23, -3.0, 4711]
```
- if you do not know index, but the element, use remove(value)
- Example:
```
my_list.remove(-3.0)
my_list == [23, 4711]
```
- if you know the index, you can use the del statement
- Example:
```
del my_list[0]
my_list == [4711]
```

Deleting Elements

- there are three different ways to delete elements from list

- as we have seen, you can also use slices to delete elements
- Example:

```
my_list = [23, 42, -3.0, 4711]
my_list[2:] = []
my_list == [23, 42]
```

- alternatively, you can use del together with slices
- Example:

```
my_list = my_list * 3
del my_list[:3]
my_list == [42, 23, 42]
```