Imperative programming with Python

Class #2

Facundo Carreiro

ILLC, University of Amsterdam

January 2015

Strings: the basics

- A *string* is a sequence of 'letters'.
- They are specified with single and double quotation marks.

```
>>> type('hey ho')
<type 'str'>
```

```
>>> type("let's go")
<type 'str'>
```

- Let's see some *potential* operations

```
>>> print ('2' - '1')
```

```
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

Strings: the basics

```
>>> print 'bat' + 'man'
```

```
batman
```

```
>>> print 'gabba '*2 + 'hey!'
```

```
gabba gabba hey!
```

- The `len(·)` function returns the length of a string

```
>>> len('gabba '*2 + 'hey!')
16
```

Strings and numbers: conversion

- We may want to convert numbers to strings

```
>>> avg = calculate_average()
>>> type(avg)
<type 'int'>
>>> print 'The average is: ' + avg + ', congratulations!'
TypeError: cannot concatenate 'str' and 'int' objects
```

- The `str(·)` function returns the string representation of a number.

```
>>> print 'The average is: ' + str(avg) + ', congratulations!'
The average is: 9.5, congratulations!
```

- Conversely, `int(·)` and `float(·)` convert strings to numbers

```
>>> type(int('282'))
<type 'int'>
```

```
>>> type(float('5.5'))
<type 'float'>
```

Strings: indexing and slicing

- We saw that strings are sequences of letters

```
s = 'this is a string'
```

- They can be *indexed* by integers with `[·]`

```
>>> s[2]
'i'
```

...starting from `0` and up to `length - 1`

```
>>> len(s)
16
>>> s[16]
IndexError: string index out of range
```

- Although...

```
>>> s[-1]
'g'
```

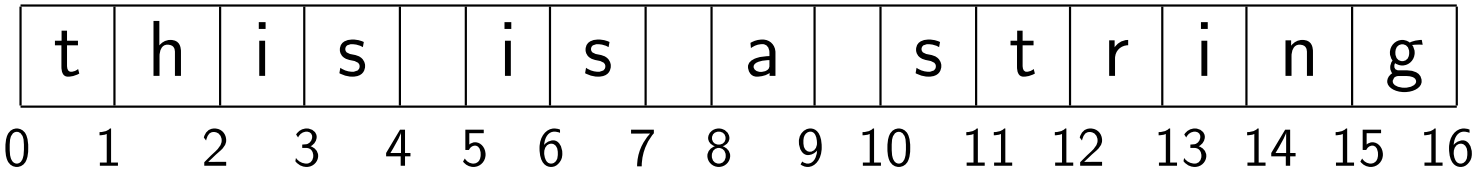
you can count *backwards* using negative numbers!
Warning: this is highly *Python*-specific.

Strings: indexing and slicing

- Strings are immutable: you can't modify them

```
>>> s[4] = 'L'
TypeError: 'str' object does not support item assignment
```

- But you can make new strings out of its *slices*




```
>>> s[0:7] + ' not' + s[7:]
'this is not a string'
```

Strings methods

- *Methods* are functions associated with an object.
- They are called using the 'dot notation'.
- For example, strings have a method called `upper`

```
>>> s.upper()
'THIS␣IS␣A␣STRING'
>>> s
'this␣is␣a␣string'
```

  it returns an uppercased version of the string *without* modifying it.

- The `find` method looks for a substring and returns its index

```
>>> 'Python'.find('thon')
2
>>> 'Python'.find('tuna')
-1
```

- Suggested HW: Check all of them in the Python documentation.

Keyboard input

- `raw_input(·)` lets the user input some text with the keyboard

```
>>> i = raw_input()
Hello, my dear program
>>> type(i)
<type 'str'>
>>> len(i)
22
>>> print i
Hello, my dear program
```

- You can use it with a message

```
>>> i = raw_input('Are you talking to me?')
Are you talking to me?Yes
>>> print i
Yes
```

Flow control: conditional execution

- The simplest form to control the flow of the execution is the *conditional execution* with the `if` statement

```
if boolean_expression: body
```

- A small example

```
name = raw_input('Please insert your name: ')
amount = int(raw_input('How much will you donate? '))

if amount <= 0:
    print 'You should input a positive number!'
    blacklist(name)
    quit()

process_donation(name, amount)
```

- Watch out: The body of the `if` statement is delimited by either tabs or spaces. This is called *indentation*. Do *not* mix tabs and spaces!

Flow control: alternative execution

- Execution of alternatives is controlled with the `else` statement

```
if boolean_expression:
    [some code block]
else:
    [some code block]
```

```
dividend = int(raw_input('Insert the dividend: '))
divisor = int(raw_input('Insert the divisor: '))

# check if the division yields an integer number
if dividend % divisor == 0:
    print 'The result is: ' + str(dividend / divisor)
else:
    print 'I\'m sorry, I can\'t do that.'
```

Flow control: chained conditionals

- You can chain conditionals with the `elif` statement (which stands for 'else if')

```
if boolean_expression:
    [some code block]
elif boolean_expression:
    [some code block]
else:
    [some code block]
```

Let's see an example of all of them...

Flow control: chained conditionals (example)

```
correct_answer = 762057
answer = input("What's the num of inhabitants in Amsterdam? ")

# compute the absolute distance to the correct answer
difference = abs(correct_answer - answer)

# ...
if answer < 0:
    print 'Are you insane?'
elif difference == 0:
    print 'Exactly!'
elif difference < 5000:
    print 'Quite close...'
elif difference < 50000:
    print 'You can do better!'
else:
    print 'Not even close...'
```

Functions

- A *function* is a named sequence of statements that performs a computation.
- We have seen some functions already: `type(·)`, `abs(·)`, `int(·)`.
- A function is 'called' by its name and 'passing' some arguments separated by commas: `name(`$arg_1, \ldots, arg_n$`)`
- Calling a function temporarily *deviates* the flow of execution.
- The arguments can be values, variables, expressions.
- Functions can have a *return value*. For example we say that `abs(·)` takes a number as an argument and returns the absolute value.

Functions

- Functions have to be *defined* before they are used.
- `abs(·)`, `int(·)` are *built-in* functions, they are defined for you with the rest of the Python language.
- Tip: If you know the name of a function you can use the `help(·)` command to get the documentation about it

```
>>> help(abs)
abs(...)
    abs(number) -> number

    Return the absolute value of the argument.
```

Functions: DIY

- Functions are defined with the `def` keyword.

```
def is_even(n):
    if n % 2 == 0:
        return True
    else:
        return False
```

- The argument passed to `is_even(n)` will be assigned to `n`.
- The `return` keyword sets the return value and exits the function immediately. It can also be used without a value (just `return`).
- If no `return` is present, the function automatically returns at the end of its body.
- *Good practice tip*: reduce the number of return points.

```
def is_even(n):
    return (n % 2 == 0)
```

Functions: local variables

- Variables inside function definitions have a *local* scope.

```
def average(n, m):
    thesum = float(n + m)
    return thesum/2
```

- You can only use the function as a *black box*

```
>>> print average(3,4)
3.5
>>> print thesum
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'thesum' is not defined
```

- Design tip: thinking of functions as black boxes performing a certain action is the way to go.

Why functions?

1. Organization
   - Divide and conquer
   - Separation of concerns
2. Code reuse
   - Do not repeat yourself
   - Functions can be shared among different programs
3. Maintainability
   - Easier to debug
   - Easier to read
4. Design for change
   - Define (or at least have in mind) an *interface* for each function
   - Encapsulate things that could change
   - Good practice foundation: Information hiding (David Parnas, "On the Criteria to Be Used in Decomposing Systems Into Modules")

Interfaces

The *interface* of a function is a summary of how it is used:

- What are the parameters?
- What does the function do? as opposed to *how*.
- What is the return value? which are the side-effects?

A popular method is that of *pre-conditions* and *post-conditions*.

- It specifies a contract between the caller and the function.
- The precondition has to be satisfied by the caller.
- The caller can assume the postcondition.
- Written in some formal language.

Interfaces: an example

Suppose we want to specify the `sort` function which takes a list of numbers and orders them.

- `sort(L:[Int])` $\rightarrow$ `res:[Int]`
- `pre:  True`
- `post`
  1. `ordered:` $\forall i, j \in \{0, \ldots, |L| - 1\}, i < j \Rightarrow res_i \leq res_j$
  2. `same list:` $\forall e \in L, e \in res \wedge \forall e \in res, e \in L$ (too weak!)

     $\forall e \in L, \texttt{count}(res, e) = \texttt{count}(L, e) \wedge$
     $\forall e \in res, \texttt{count}(res, e) = \texttt{count}(L, e)$
     where $\texttt{count}(A, e) := |[i : i \in \{0, \ldots, |A| - 1\}, A_i = e]|$

As you can see,

- It helps spot possible mistakes.
- We end up having an unambiguous specification.
- *It is hard work*, even for simple and small functions.

References

- Chapters 3, 5 and 6 of the book
  http://greenteapress.com/thinkpython/thinkpython.html
- Boolean operations
  http://docs.python.org/reference/expressions.html#boolean-operations
- Python: Myths about indentation
  http://www.secnetix.de/olli/Python/block_indentation.hawk
- The Python Standard Library
  http://docs.python.org/library/