Lecture 4 – Compound data types, Traversals

https://cw.fel.cvut.cz/wiki/courses/be5b33prg/start

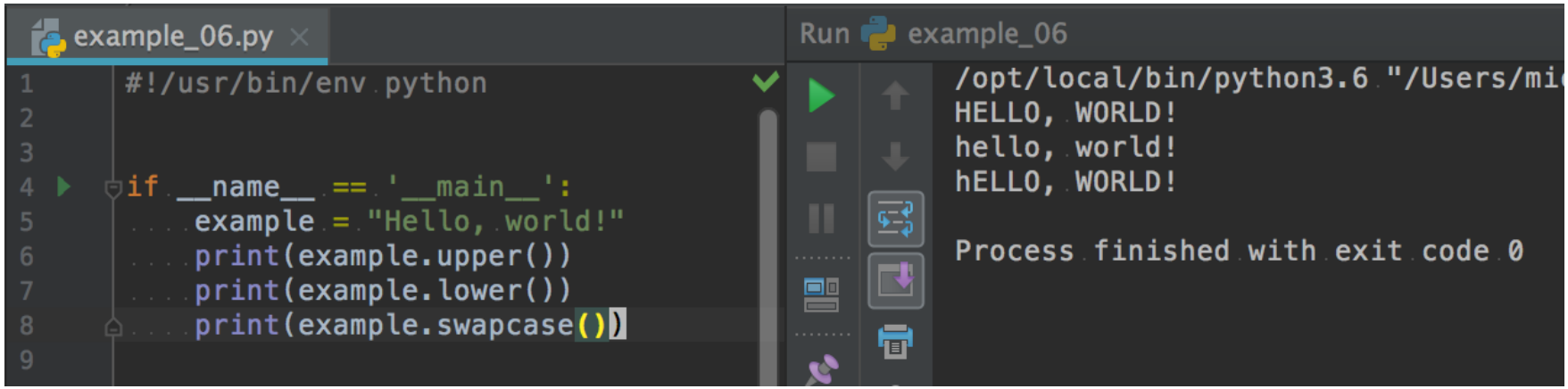
Tomas Jenicek

Czech Technical University in Prague,
Faculty of Electrical Engineering, Dept. of Cybernetics,
Center for Machine Perception
http://cmp.felk.cvut.cz/~jenicto2/
tomas.jenicek@fel.cvut.cz

- Everything in Python is **object**
- Python is **dynamically typed** language (*type changes with reference*)
- The methods and variables are created on the **stack memory**
- The objects and instances are created on the **heap memory**
- New **stack frame** is created on invocation of a function / method and references are assigned & counted
- Stack frames are destroyed as soon as the function / method returns
- Mechanism to clean up the dead objects is **Garbage collector** (*algorithm used is **Reference Counting** and immediate object removal if count == 0*)

source https://www.youtube.com/watch?v=arxWaw-E8QQ&t=1s

COMPOUND DATA TYPES

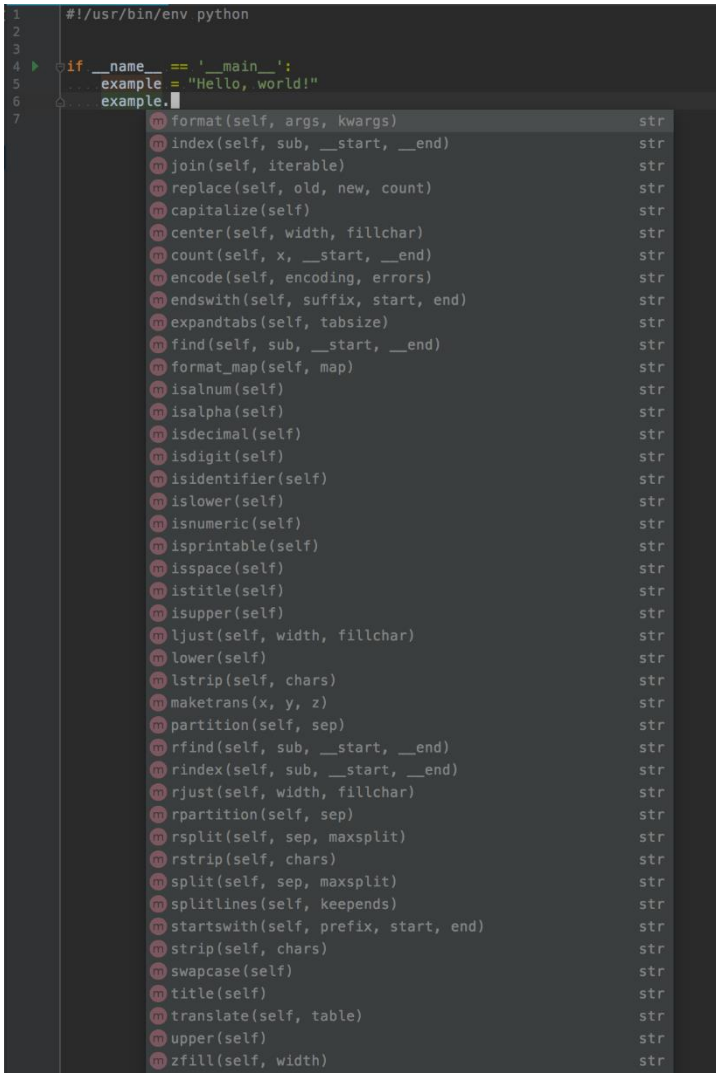


- So far built-in types like **int**, **float**, **bool**
- Compound data types:
  strings, **lists**, **dictionaries**, and **tuples** are different from the others because they are made up of smaller pieces (*characters in case of a string, items in case of a list*)

- Types comprising smaller pieces are **compound data types**

source http://openbookproject.net/thinkcs/python/english3e/strings.html

STRINGS



- Example: **upper** is a method that can be invoked on any string object to create a new string, where all the characters are in uppercase

- **lower**, **capitalize**, **swapcase** …
- Use documentation & help!

source http://openbookproject.net/thinkcs/python/english3e/strings.html

```
1  if word < "banana":
2      print("Your word, " + word + ", comes before banana.")
3  elif word > "banana":
4      print("Your word, " + word + ", comes after banana.")
5  else:
6      print("Yes, we have no bananas!")
```

```
1  greeting = "Hello, world!"
2  greeting[0] = 'J'              # ERROR!
3  print(greeting)
```

```
1  greeting = "Hello, world!"
2  new_greeting = "J" + greeting[1:]
3  print(new_greeting)
```

- Comparing strings: strings are **sorted** in the alphabetical order (except that all uppercase letters come before the lowercase)
- Strings are **immutable**
  (existing string cannot be changed, new one should be created instead)

source http://openbookproject.net/thinkcs/python/english3e/strings.html

```
>>> "p" in "apple"
True
>>> "i" in "apple"
False
>>> "ap" in "apple"
True
>>> "pa" in "apple"
False
```

```
def find(strng, ch):
    """
      Find and return the index of ch in strng.
      Return -1 if ch does not occur in strng.
    """
    ix = 0
    while ix < len(strng):
        if strng[ix] == ch:
            return ix
        ix += 1
    return -1

test(find("Compsci", "p") == 3)
test(find("Compsci", "C") == 0)
test(find("Compsci", "i") == 6)
test(find("Compsci", "x") == -1)
```

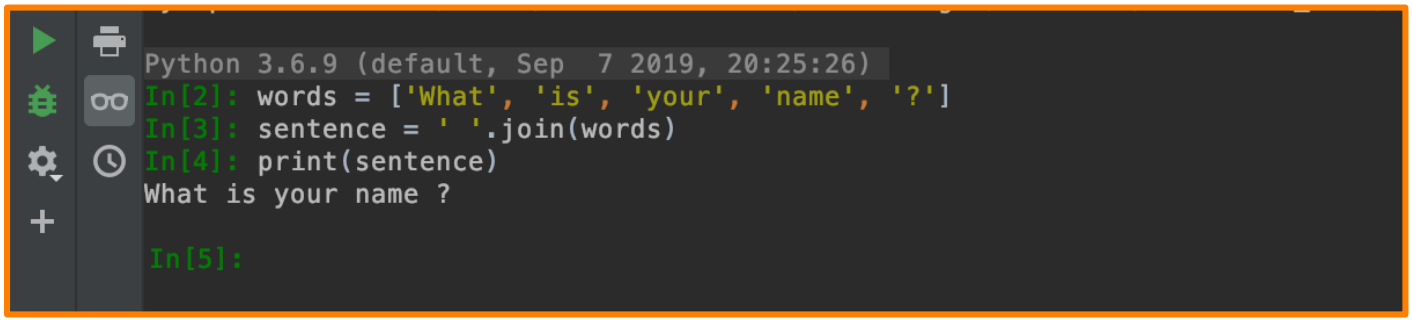
- The **in** / **not in** operator tests for **membership**
- Method **index** is the opposite of the indexing operator:
  it takes a character (item in case of a list) and finds the index of the character / item (*if not found then exception is raised*)
- Method **find** works for strings in a similar way
  (*if the character is not found, the function returns -1*)

source http://openbookproject.net/thinkcs/python/english3e/strings.html

```
>>> ss = "Well I never did said Alice"
>>> wds = ss.split()
>>> wds
['Well', 'I', 'never', 'did', 'said', 'Alice']
```

- The **split** method:
  it splits a single multi-word string into a list of individual words, removing all the whitespace between them (whitespace are: tabs, newlines, spaces)

- Explore the **join** method on your own!



source http://openbookproject.net/thinkcs/python/english3e/strings.html

STRINGS

```
1  s1 = "His name is {0}!".format("Arthur")
2  print(s1)
3
4  name = "Alice"
5  age = 10
6  s2 = "I am {1} and I am {0} years old.".format(age, name)
7  print(s2)
8
9  n1 = 4
10 n2 = 5
11 s3 = "2**10 = {0} and {1} * {2} = {3:f}".format(2**10, n1, n2, n1 * n2)
12 print(s3)
```

```
His name is Arthur!
I am Alice and I am 10 years old.
2**10 = 1024 and 4 * 5 = 20.000000
```

- The **format** method substitutes its arguments into the place holders (**numbers are indexes of the arguments**)
- Format specification — it is always introduced by the colon **:**
- Field is aligned to the **left** <, **center** ^, or **right** >
- Width allocated to the field within the result string
- Type conversion
- Specification of **decimal places**
  (**.2f** is useful for when rounding to two decimal places.)

source http://openbookproject.net/thinkcs/python/english3e/strings.html

```
1  >>> vocabulary = ["apple", "cheese", "dog"]
2  >>> numbers = [17, 123]
3  >>> an_empty_list = []
4  >>> print(vocabulary, numbers, an_empty_list)
5  ["apple", "cheese", "dog"] [17, 123] []
```

- A **list** is an **ordered collection of values**
- Values of a list are called its **elements** or **items**
- Similar to strings (**ordered collections of characters**) except that the elements of a list can be of **any type**

- **Lists and strings** — and other collections that maintain the order of their items — are called **sequences**
- **List within list** is said to be **nested**
- **List with no elements** is called an **empty** list, and is denoted **[]**

source http://openbookproject.net/thinkcs/python/english3e/lists.html

LISTS

```
1  students = [
2      ("John", ["CompSci", "Physics"]),
3      ("Vusi", ["Maths", "CompSci", "Stats"]),
4      ("Jess", ["CompSci", "Accounting", "Economics", "Management"]),
5      ("Sarah", ["InfSys", "Accounting", "Economics", "CommLaw"]),
6      ("Zuki", ["Sociology", "Economics", "Law", "Stats", "Music"])]
7
8  # Count how many students are taking CompSci
9  counter = 0
10 for (name, subjects) in students:
11     if "CompSci" in subjects:
12             counter += 1
13
14 print("The number of students taking CompSci is", counter)
```

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

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

- Expression evaluating to an integer can be used as an index
- Function **len** returns **length of a list** (number of its elements)
- Testing membership using **in** / **not in**
- Operators **+** (**concatenation**) and ***** (**repetition**)

source http://openbookproject.net/thinkcs/python/english3e/lists.html

LISTS

```
>>> a_list = ["a", "b", "c", "d", "e", "f"]
>>> a_list[1:3]
['b', 'c']
>>> a_list[:4]
['a', 'b', 'c', 'd']
>>> a_list[3:]
['d', 'e', 'f']
>>> a_list[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

```
>>> a_list = ["a", "d", "f"]
>>> a_list[1:1] = ["b", "c"]
>>> a_list
['a', 'b', 'c', 'd', 'f']
>>> a_list[4:4] = ["e"]
>>> a_list
['a', 'b', 'c', 'd', 'e', 'f']
```

```
>>> a_list = ["a", "b", "c", "d", "e", "f"]
>>> a_list[1:3] = []
>>> a_list
['a', 'd', 'e', 'f']
```

```
>>> my_string = "TEST"
>>> my_string[2] = "X"
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

```
>>> my_list = ["T", "E", "S", "T"]
>>> my_list[2] = "X"
>>> my_list
['T', 'E', 'X', 'T']
```

- Lists are **mutable** (we can change list elements)
- Use same **slicing principles** as for strings
- Use **del** to delete list elements

source http://openbookproject.net/thinkcs/python/english3e/lists.html

TUPLES


```
>>> julia = ("Julia", "Roberts", 1967, "Duplicity", 2009, "Actress", "Atlanta, Georgia")
```

```
>>> julia[2]
1967
```

```
>>> julia[0] = "X"
TypeError: 'tuple' object does not support item assignment
```

- The pair data example is an example of a **tuple**
- Tuple groups any number of items into a **compound value**
- Tuple is a **comma-separated sequence of values**

- Other languages often call it **records** (*some related information that belongs together)*
- Important: strings and tuples are **immutable** (*once Python creates a tuple in memory, it cannot be changed*)
- Elements of a tuple **cannot be modified**, **new tuple holding different information** should always be made instead!

source http://openbookproject.net/thinkcs/python/english3e/tuples.html

TUPLES


```
(name, surname, b_year, movie, m_year, profession, b_place) = julia
```

```
>>> b = ("Bob", 19, "CS")    # tuple packing
```

```
>>> b = ("Bob", 19, "CS")
>>> (name, age, studies) = b    # tuple unpacking
>>> name
'Bob'
>>> age
```

```
>>> (a, b, c, d) = (1, 2, 3)
ValueError: need more than 3 values to unpack
```

- Powerful **tuple assignment** (remember variable swapping?)
- Equivalent of **multiple assignment statements**
- Requirement: the number of **variables on the left** must match the **number of elements** in the tuple
- Tuple assignment is called tuple **packing** / **unpacking**

source http://openbookproject.net/thinkcs/python/english3e/tuples.html

```
1  def f(r):
2      """ Return (circumference, area) of a circle of radius r """
3      c = 2 * math.pi * r
4      a = math.pi * r * r
5      return (c, a)
```

- Use of tuples in functions as **return value**
- Function can *always only return a single value*, but by making that value a tuple, as many values can be **packed together** as is needed (*e.g. find the mean and the standard deviation*)
- Tuple items can themselves be other tuples (**nested tuples**)

- **Heterogeneous data structure**: can be composed of elements of different types (tuples, strings, lists)

source http://openbookproject.net/thinkcs/python/english3e/tuples.html

UNPACKING – PAIRED DATA

```
1  celebs = [("Brad Pitt", 1963), ("Jack Nicholson", 1937),
2                                 ("Justin Bieber", 1994)]
```

```
1  print(celebs)
2  print(len(celebs))
```

```
[("Brad Pitt", 1963), ("Jack Nicholson", 1937), ("Justin Bieber", 1994)]
3
```

```
1  for (nm, yr) in celebs:
2      if yr < 1980:
3          print(nm)
```

```
Brad Pitt
Jack Nicholson
```

- Example of paired data: *lists of names* and *lists of numbers*
- Advanced way of representing data: making a pair of things is as simple as putting them into parentheses (i.e. **tuples**)

source http://openbookproject.net/thinkcs/python/english3e/iteration.html

INDEXING

```
>>> fruit = "banana"
>>> m = fruit[1]
>>> print(m)
```

```
>>> m = fruit[0]
>>> print(m)
b
```

- Python uses **square brackets** to enclose the index – **indexing operator []**
- The expression in brackets is called an **index**
- Example: *The expression fruit[1] selects character number 1 from fruit, and creates a new string containing just this one character*
- Computer scientists always start **counting from zero**!
- An index specifies a **member of an ordered collection** (*in this case the collection of characters in the string*)
- Index indicates *which one you want*, hence the name
- Index can be any **integer expression** (not only value)

source http://openbookproject.net/thinkcs/python/english3e/strings.html

INDEXING

```
>>> fruit = "banana"
>>> list(enumerate(fruit))
[(0, 'b'), (1, 'a'), (2, 'n'), (3, 'a'), (4, 'n'), (5, 'a')]
```

```
>>> prime_nums = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
>>> prime_nums[4]
11
>>> friends = ["Joe", "Zoe", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]
>>> friends[3]
'Angelina'
```

- Use **enumerate** to visualize indices
- Note that indexing strings returns a string: Python has no special type for a single character (string of length = 1)
- Use **index** to extract elements from a list

source http://openbookproject.net/thinkcs/python/english3e/strings.html

INDEXING


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

```
1  sz = len(fruit)
2  last = fruit[sz]          # ERROR!
```

```
IndexError: string index out of range.
```

```
1  sz = len(fruit)
2  last = fruit[sz-1]
```

- Use **len** to extract the **number of elements** (indexing from 0!)
- Negative indices count backward from the end of the string
- *The expression fruit[-1] yields the last letter*
- Traversals: **while** vs. **for** comparison again!

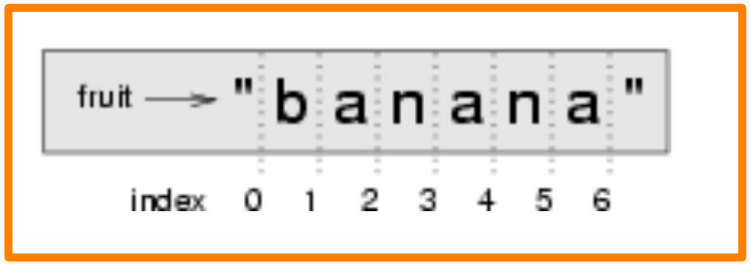
```
1  ix = 0
2  while ix < len(fruit):
3      letter = fruit[ix]
4      print(letter)
5      ix += 1
```

```
1  for c in fruit:
2      print(c)
```

source http://openbookproject.net/thinkcs/python/english3e/strings.html

SLICING


```
>>> s = "Pirates of the Caribbean"
>>> print(s[0:7])
Pirates
>>> print(s[11:14])
the
>>> print(s[15:24])
Caribbean
>>> friends = ["Joe", "Zoe", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]
>>> print(friends[2:4])
['Brad', 'Angelina']
```



- A **substring** of a string is obtained by taking a **slice**
- Slice a list to refer to some **sublist** of the items in the list
- The operator **[n:m]** returns the part of the string from the *n'th* character to the *m'th* character, **including the first but excluding the last** (*indices pointing between the characters*)
- Slice operator [n:m] **copies** out the part of the paper between the **n** and **m** positions
- Result of **[n:m]** will be of **length (m-n)**

source http://openbookproject.net/thinkcs/python/english3e/strings.html

SLICING

```
>>> fruit = "banana"
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
>>> fruit[3:999]
'ana'
```

- If you **omit the first index** (before the colon), the slice starts at the beginning of the string (or list)
- If you **omit the second index**, the slice extends to the end of the string (or list)
- If you provide value for n that is bigger than the length of the string (or list), the slice will take all the values up to the end
- No **"out of range" error** like the normal indexing operation

source http://openbookproject.net/thinkcs/python/english3e/strings.html

TRAVERSAL – THE FOR LOOP

```
1   def mysum(xs):
2       """ Sum all the numbers in the list xs, and return the total. """
3       running_total = 0
4       for x in xs:
5           running_total = running_total + x
6       return running_total
7
8   # Add tests like these to your test suite ...
9   test(mysum([1, 2, 3, 4]) == 10)
10  test(mysum([1.25, 2.5, 1.75]) == 5.5)
11  test(mysum([1, -2, 3]) == 2)
12  test(mysum([ ]) == 0)
13  test(mysum(range(11)) == 55)  # 11 is not included in the list.
```

- Automate **repetitive tasks** without errors
- Repeated execution of a set of statements is called **iteration**

- Already explored **for**, now explore **while**
- Running through all items in a list is **traversing** / **traversal**

source http://openbookproject.net/thinkcs/python/english3e/iteration.html

TRAVERSAL – THE WHILE LOOP

```
1   def sum_to(n):
2       """ Return the sum of 1+2+3 ... n """
3       ss  = 0
4       v = 1
5       while v <= n:
6           ss = ss + v
7           v = v + 1
8       return ss
9
10  # For your test suite
11  test(sum_to(4) == 10)
12  test(sum_to(1000) == 500500)
```

- The **while** statement has same meaning as in English
- Evaluate the condition (*at line 5*) either **False** or **True**.
- If the value is **False**, exit the while statement and continue execution at the next statement (*line 8 in this case*)
- If the value is **True**, execute each of the statements in the body (*lines 6 and 7*), then go back to the **while** statement

source http://openbookproject.net/thinkcs/python/english3e/iteration.html

TRAVERSAL – THE WHILE LOOP

```
1  def sum_to(n):
2      """ Return the sum of 1+2+3 ... n """
3      ss  = 0
4      v = 1
5      while v <= n:
6          ss = ss + v
7          v = v + 1
8      return ss
9
10 # For your test suite
11 test(sum_to(4) == 10)
12 test(sum_to(1000) == 500500)
```

```
1  def sum_to(n):
2      """ Return the sum of 1+2+3 ... n """
3      ss  = 0
4      for v in range(n+1):
5          ss = ss + v
6      return ss
```

- The while loop is **more work** than the equivalent for loop
- Need to **manage the loop variable**: give it an **initial** value, **test for completion**, update it in the body to enable **termination**

- Note: *range generates a list up to but excluding the last value*

source http://openbookproject.net/thinkcs/python/english3e/iteration.html

TRAVERSAL – WHILE vs. FOR

- Use a **for** loop if you know how many times the loop will execute (**definite iteration** — we know ahead some definite bounds for what is needed)
- Use a **for** to loop over **iterables** (to be explored in later classes) usually in combination with **in**
- Use **while** loop if you are required to repeat computation until given condition is met, and you cannot calculate in advance when this will happen (**indefinite iteration** — we do not know how many iterations will be needed)

source http://openbookproject.net/thinkcs/python/english3e/iteration.html

TRAVERSAL – BREAK vs. CONTINUE


```
1  while True:
2      play_the_game_once()
3      response = input("Play again? (yes or no)")
4      if response != "yes":
5          break
6  print("Goodbye!")
```

```
1  for i in [12, 16, 17, 24, 29, 30]:
2      if i % 2 == 1:       # If the number is odd
3          continue         # Don't process it
4      print(i)
5  print("done")
```

```
12
16
24
30
done
```

- The **break** statement in Python terminates the current loop and resumes execution at the next statement
- The **continue** statement in Python returns the control to the beginning of the current loop
- The **continue** statement rejects all the remaining statements in the current iteration of the loop …

Source http://www.tutorialspoint.com/python/python_loop_control.htm

EXAMPLE

```
1   import random                       # We cover random numbers in the
2   rng = random.Random()               #  modules chapter, so peek ahead.
3   number = rng.randrange(1, 1000) # Get random number between [1 and 1000).
4
5   guesses = 0
6   msg = ""
7
8   while True:
9       guess = int(input(msg + "\nGuess my number between 1 and 1000: "))
10      guesses += 1
11      if guess > number:
12          msg += str(guess) + " is too high.\n"
13      elif guess < number:
14          msg += str(guess) + " is too low.\n"
15      else:
16          break
17
```

- Guessing **game**
- This program makes use of the mathematical law of **trichotomy** (*given real numbers a and b, exactly one of these three must be true: a > b, a < b, or a == b*)

source http://openbookproject.net/thinkcs/python/english3e/iteration.html

NESTED DATA


```
students = [
    ("John", ["CompSci", "Physics"]),
    ("Vusi", ["Maths", "CompSci", "Stats"]),
    ("Jess", ["CompSci", "Accounting", "Economics", "Management"]),
    ("Sarah", ["InfSys", "Accounting", "Economics", "CommLaw"]),
    ("Zuki", ["Sociology", "Economics", "Law", "Stats", "Music"])]
```

```
# Print all students with a count of their courses.
for (name, subjects) in students:
    print(name, "takes", len(subjects), "courses")
```

```
John takes 2 courses
Vusi takes 3 courses
Jess takes 4 courses
Sarah takes 4 courses
Zuki takes 5 courses
```

```
# Count how many students are taking CompSci
counter = 0
for (name, subjects) in students:
    for s in subjects:                    # A nested loop!
        if s == "CompSci":
            counter += 1

print("The number of students taking CompSci is", counter)
```

```
The number of students taking CompSci is 3
```

- Data structure — a mechanism for **grouping** and **organizing** data to make it easier to use

source http://openbookproject.net/thinkcs/python/english3e/iteration.html

REFERENCES – STRINGS vs. LISTS

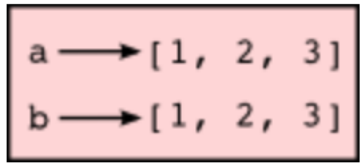
Strings

```
1  a = "banana"
2  b = "banana"
```


```
>>> a is b
True
```

Lists

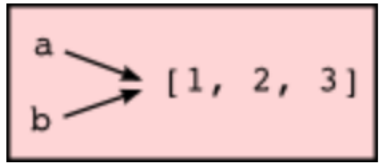
```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a == b
True
>>> a is b
False
```



- Variables **a** and **b** refer to string object with letters "banana"
- Use **is** operator or **id** function to find out the **reference**
- Strings are **immutable**
  Python optimizes resources by making two names that refer to the same string value refer to the same object
- Not the case of lists: **a** and **b** have the same value (content) but do not refer to the same object

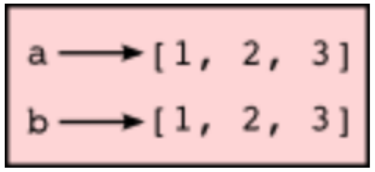
source http://openbookproject.net/thinkcs/python/english3e/lists.html

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




```
>>> b[0] = 5
>>> a
[5, 2, 3]
```

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b
[1, 2, 3]
```




```
>>> b[0] = 5
>>> a
[1, 2, 3]
```

- If we assign one variable to another, both variables refer to the same object
- The **same list has two different names** we say that it is **aliased** (*changes made with one alias affect the other*)
- Recommendation:
  Avoid aliasing when you are working with mutable objects!
- If need to modify a list and keep a copy of the original use the **slice operator** (*taking any slice of creates a new list*)

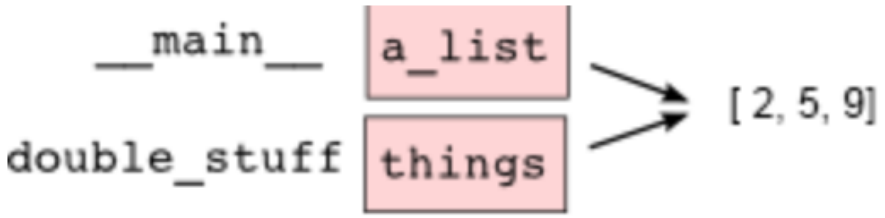
source http://openbookproject.net/thinkcs/python/english3e/lists.html

LIST PARAMETERS

```
1  def double_stuff(a_list):
2      """ Overwrite each element in a_list with double its value. """
3      for (idx, val) in enumerate(a_list):
4          a_list[idx] = 2 * val
```

```
1  things = [2, 5, 9]
2  double_stuff(things)
3  print(things)
```

```
[4, 10, 18]
```



- Passing a **list as an argument** passes a **reference** to the list, **not a copy or clone** of the list!
- So parameter passing creates an **alias!**

source http://openbookproject.net/thinkcs/python/english3e/lists.html

LIST METHODS

```
>>> mylist = []
>>> mylist.append(5)
>>> mylist.append(27)
>>> mylist.append(3)
>>> mylist.append(12)
>>> mylist
[5, 27, 3, 12]
```

```
>>> mylist.insert(1, 12)  # Insert 12 at pos 1, shift other items up
>>> mylist
[5, 12, 27, 3, 12]
>>> mylist.count(12)       # How many times is 12 in mylist?
2
>>> mylist.extend([5, 9, 5, 11])   # Put whole list onto end of mylist
>>> mylist
[5, 12, 27, 3, 12, 5, 9, 5, 11])
>>> mylist.index(9)                # Find index of first 9 in mylist
6
>>> mylist.reverse()
>>> mylist
[11, 5, 9, 5, 12, 3, 27, 12, 5]
>>> mylist.sort()
>>> mylist
[3, 5, 5, 5, 9, 11, 12, 12, 27]
>>> mylist.remove(12)             # Remove the first 12 in the list
>>> mylist
[3, 5, 5, 5, 9, 11, 12, 27]
```

Explore list methods on your own!

```python
def double_stuff(a_list):
    """ Return a new list which contains
        doubles of the elements in a_list.
    """
    new_list = []
    for value in a_list:
        new_elem = 2 * value
        new_list.append(new_elem)

    return new_list
```

```python
def double_stuff(a_list):
    """ Overwrite each element in a_list with double its value. """
    for (idx, val) in enumerate(a_list):
        a_list[idx] = 2 * val
```

- Concept: **pure functions** vs. **modifiers**
- Pure function does not produce **side effects**!
- Pure function communicates with the calling program **only through parameters** (it does not modify) and a **return value**
- *Do not alter the input parameters unless really necessary*
- Programs that use pure functions are **faster to develop** and **less error-prone** than programs that use modifiers

Source by Tomas Svoboda PRG 2016/2017

REFERENCES

This lecture re-uses selected parts of the OPEN BOOK PROJECT
Learning with Python 3 (RLE)
http://openbookproject.net/thinkcs/python/english3e/index.html
available under GNU Free Documentation License Version 1.3)

- Version date: October 2012
- by Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers (based on 2nd edition by Jeffrey Elkner, Allen B. Downey, and Chris Meyers)
- Source repository is at https://code.launchpad.net/~thinkcspy-rle-team/thinkcspy/thinkcspy3-rle
- For offline use, download a zip file of the html or a pdf version from http://www.ict.ru.ac.za/Resources/cspw/thinkcspy3/