Lecture 4 – Compound data types, Traversals
https://cw.fel.cvut.cz/wiki/courses/be5b33prg/start

Michal Reinštein

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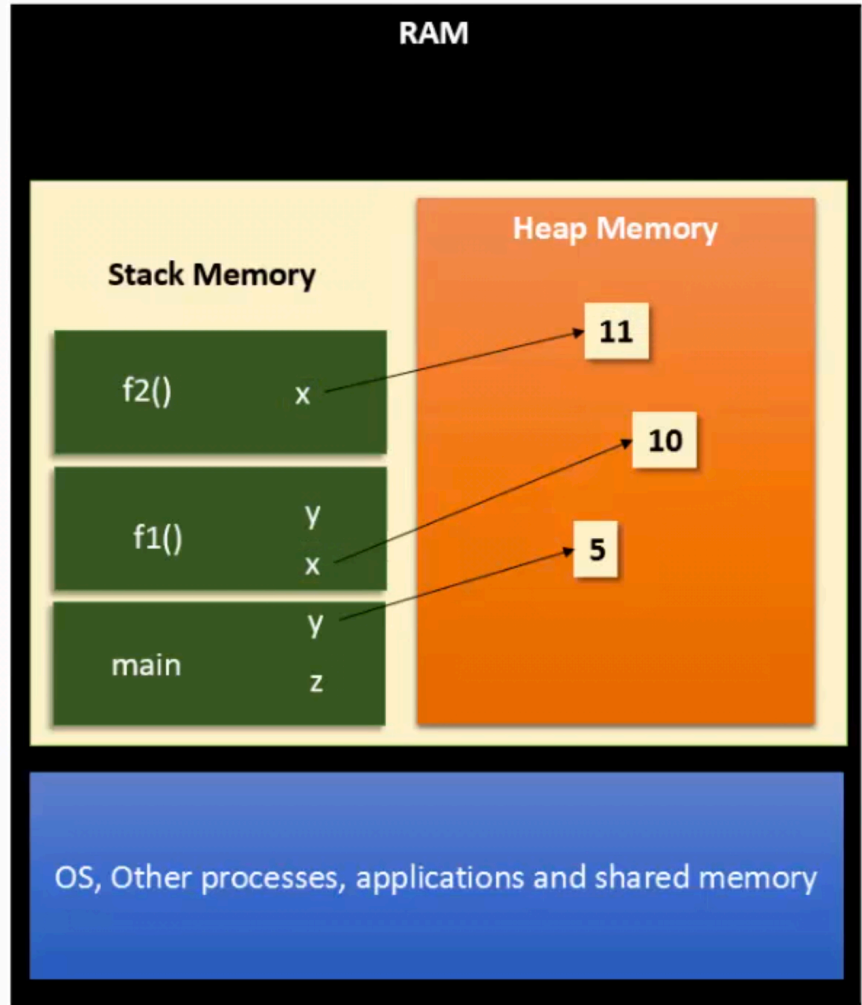Center for Machine Perception
http://cmp.felk.cvut.cz/~reinsmic/
reinstein.michal@fel.cvut.cz

```python
#!/usr/bin/env python


def f1(x):
    x *= 2
    y = f2(x)
    return y


def f2(x):
    x += 1
    return x



if __name__ == '__main__':
    y = 5
    z = f1(y)
    print(z)

```

RAM

Heap Memory

Stack Memory

f2()   x   11

f1()   y
            x   10

main   y   5
            z

OS, Other processes, applications and shared memory

```python
def f1(x):
    x *= 2
    y = f2(x)
    return y
```
Shadows name 'y' from outer scope less... (⌘F1)

This inspection detects shadowing names defined in outer scopes.
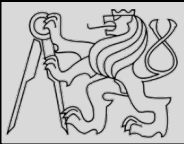
```python
    x += 1
    return x
```

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

	Python	JAVA / C
Statement	x = 10	int x = 10;
Data type declaration	Not needed. Dynamically typed.	Mandatory. Statically typed.
What is 10?	An Object created on heap memory.	A primitive data stored in 4 byte
What does x contain?	Reference to Object 10	Memory location where 10 is stored
x = x + 1	x starts referring to a new object whose value is 11	x continues to point to the same memory, with value equal to 11
x = 10 y = 10	Both x and y will refer to the same object.	x and y are two variables pointing to different memory locations.

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

- The methods and variables are created on **stack memory**
- The objects and instances are created on **heap memory**
- New **stack frame** is created on invocation of a function / method
- Stack frames are destroyed as soon as the function / method returns
- Mechanism to clean up the dead objects is **Garbage collector**
- Everything in Python is **object**
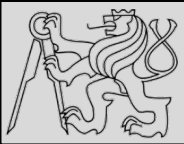- Python is **dynamically typed** language

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

```python
def mysum(xs):
    """ Sum all the numbers in the list xs, and return the total. """
    running_total = 0
    for x in xs:
        running_total = running_total + x
    return running_total

# Add tests like these to your test suite ...
test(mysum([1, 2, 3, 4]) == 10)
test(mysum([1.25, 2.5, 1.75]) == 5.5)
test(mysum([1, -2, 3]) == 2)
test(mysum([ ]) == 0)
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

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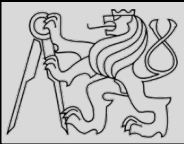
- The **while** statement has same meaning as in English
- Evaluate the condition (*at line 5*) either **False** or **True**.
- If the value is **False**, <u>exit the while statement</u> and continue execution at the next statement (*line 8 in this case*)
- If the value is **True**, <u>execute each of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

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

```python
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)
```

```python
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**
- <u>Note:</u> *range generates a list up to but excluding the last value*

- Use a **for** loop if you know <u>how many times the loop</u> 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 <u>repeat computation until given condition is met</u>, and you cannot calculate in advance when this will happen (**indefinite iteration** — we do not know how many iterations will be needed)

```python
while True:
    play_the_game_once()
    response = input("Play again? (yes or no)")
    if response != "yes":
        break
print("Goodbye!")
```

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

```
12
16
24
30
done
```

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

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

EXAMPLE

10

```python
import random                        # We cover random numbers in the
rng = random.Random()                #  modules chapter, so peek ahead.
number = rng.randrange(1, 1000)      # Get random number between [1 and 1000).

guesses = 0
msg = ""

while True:
    guess = int(input(msg + "\nGuess my number between 1 and 1000: "))
    guesses += 1
    if guess > number:
        msg += str(guess) + " is too high.\n"
    elif guess < number:
        msg += str(guess) + " is too low.\n"
    else:
        break

input("\n\nGreat, you got it in {0} guesses!\n\n".format(guesses))
```

- 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*)

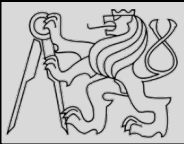source http://openbookproject.net/thinkcs/python/english3e/iteration.html

- So far built-in types like **int**, **float**, **bool**
- <u>Compound data types</u>: **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

```
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**)

```
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"])]
```

```
1  # Print all students with a count of their courses.
2  for (name, subjects) in students:
3      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
```

```
1  # Count how many students are taking CompSci
2  counter = 0
3  for (name, subjects) in students:
4      for s in subjects:                # A nested loop!
5          if s == "CompSci":
6              counter += 1
7
8  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

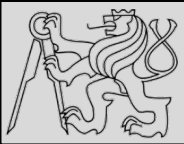source http://openbookproject.net/thinkcs/python/english3e/iteration.html

```
>>> 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*)
- <u>Important</u>: 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

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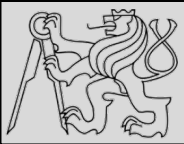
- Powerful **tuple assignment** (remember variable swapping)
- Equivalent of **multiple assignment statements**
- Requirement: the number of **variables on the left** must match the **number of elements** in the tuple
- Tuple assignment is called tuple **packing** / **unpacking**

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

```python
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)

```python
#!/usr/bin/env python


if __name__ == '__main__':
    example = "Hello, world!"
    example.
```

format(self, args, kwargs)	str
index(self, sub, __start, __end)	str
join(self, iterable)	str
replace(self, old, new, count)	str
capitalize(self)	str
center(self, width, fillchar)	str
count(self, x, __start, __end)	str
encode(self, encoding, errors)	str
endswith(self, suffix, start, end)	str
expandtabs(self, tabsize)	str
find(self, sub, __start, __end)	str
format_map(self, map)	str
isalnum(self)	str
isalpha(self)	str
isdecimal(self)	str
isdigit(self)	str
isidentifier(self)	str
islower(self)	str
isnumeric(self)	str
isprintable(self)	str
isspace(self)	str
istitle(self)	str
isupper(self)	str
ljust(self, width, fillchar)	str
lower(self)	str
lstrip(self, chars)	str
maketrans(x, y, z)	str
partition(self, sep)	str
rfind(self, sub, __start, __end)	str
rindex(self, sub, __start, __end)	str
rjust(self, width, fillchar)	str
rpartition(self, sep)	str
rsplit(self, sep, maxsplit)	str
rstrip(self, chars)	str
split(self, sep, maxsplit)	str
splitlines(self, keepends)	str
startswith(self, prefix, start, end)	str
strip(self, chars)	str
swapcase(self)	str
title(self)	str
translate(self, table)	str
upper(self)	str
zfill(self, width)	str

- 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

```
>>> 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**
- <u>Example</u>*: 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

```
>>> 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

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

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

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

IndexError: string index out of range.

- 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

```
>>> 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']
```

fruit ⟶ "b a n a n a"

index   0   1   2   3   4   5   6
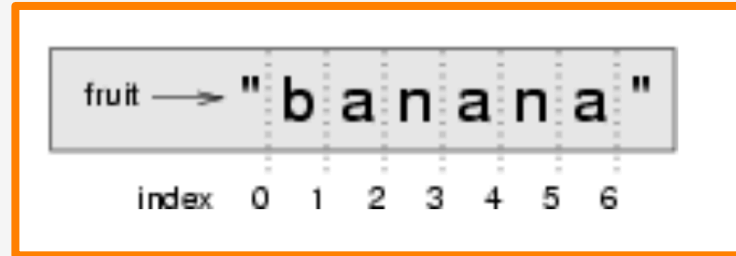
- A **substring** of a string is obtained by taking a **slice**
- Slice a list to refer to some **sublist** of the items in the list
- The operator **[n:m]** returns the part of the string from the *n'th* character to the *m'th* character, **including the first but excluding the last** (*indices pointing 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

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

- If you **omit the first index** (before the colon), the slice <u>starts at the beginning</u> of the string (or list)
- If you **omit the second index**, the slice <u>extends to the end</u> 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 <u>take all the values</u> up to the end
- No **"out of range" error** like the normal indexing operation

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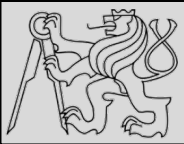
- <u>Comparing strings</u>: strings are **sorted** in the alphabetical order (except that all uppercase letters come before the lowercase)
- Strings are **immutable**
  (existing string cannot be change, new one should be created instead)

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

```python
1   def find(strng, ch):
2       """
3         Find and return the index of ch in strng.
4         Return -1 if ch does not occur in strng.
5       """
6       ix = 0
7       while ix < len(strng):
8           if strng[ix] == ch:
9               return ix
10          ix += 1
11      return -1
12
13  test(find("Compsci", "p") == 3)
14  test(find("Compsci", "C") == 0)
15  test(find("Compsci", "i") == 6)
16  test(find("Compsci", "x") == -1)
```

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

```
>>> 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)

```
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 of conversion
- Specification of **decimal places** (**.2f** is useful for working with currencies 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

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

# Count how many students are taking CompSci
counter = 0
for (name, subjects) in students:
    if "CompSci" in subjects:
        counter += 1

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

```
>>> 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 their elements)
- Use same slicing principles as for strings
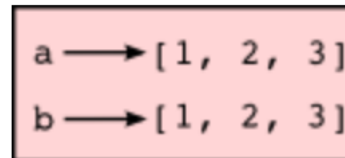- Use **del** to delete list elements

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

Strings

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

a ────→ "banana"
b ────→

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

Lists

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

a ────→ [1, 2, 3]

b ────→ [1, 2, 3]

- 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

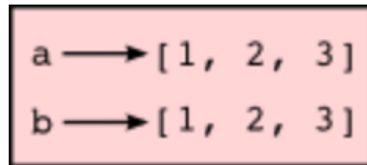source http://openbookproject.net/thinkcs/python/english3e/lists.html

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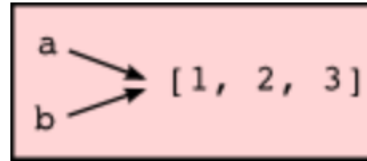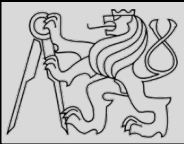
```
>>> b[0] = 5
>>> a
[5, 2, 3]
```

```
>>> 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_)
- <u>Recommendation</u>: **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** (<u>taking any slice of a creates a new list</u>)

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

```python
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
```

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

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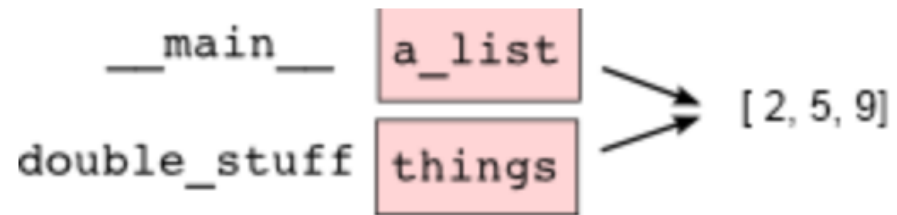

- Passing a **list as an argument** passes a **reference** to the list, **not a copy or clone** of the list
- So parameter passing creates an **alias**

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

```
>>> 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!

Source by Tomas Svoboda PRG 2016/2017

```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
```

- <u>Concept</u>: **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**
- <u>Do not alter the input parameters unless really necessary</u>
- 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

functions that produce lists

- `def fcn(par):`

- `initialize result as empty list`

- `loop`

  - `create a new element`

  - `add to the result`

- `return result`