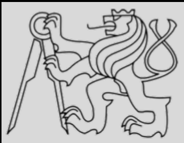
Lecture 10 – Selected parts of Python

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



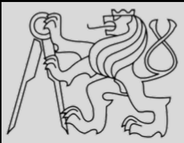
DATA TYPES


```
Python 3.6.3 (default, Oct  5 2017, 23:34:28)
[GCC 4.2.1 Compatible Apple LLVM 8.1.0 (clang-802.0.42)] on darwin
In[2]: type(11)
Out[2]: int
In[3]: type(11.1234)
Out[3]: float
In[4]: type("1.1234")
Out[4]: str
In[5]: type("Bob")
Out[5]: str
In[6]: type("""Hello, World!""")
Out[6]: str

In[7]:
```

- Integers (**int**) 1, 10, 124
- Strings (**str**) ”Hello, World!”
- Float (**float**) 1.0, 9.999

- Strings in Python can be enclosed in either single quotes (') or double quotes ("), or three of each (''' or """)

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



VARIABLES


```
Python Console
/opt/local/bin/python3.6 /Applications/PyCharm.a
Python 3.6.3 (default, Oct  5 2017, 23:34:28)
In[2]: my_name = "Bob"
In[3]: my_age = 17
In[4]: my_height = 183.5

In[5]:
```

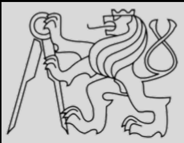
```
Special Variables
_ = {str} ''
__ = {str} ''
___ = {str} ''
my_age = {int} 17
my_height = {float} 183.5
my_name = {str} 'Bob'
```

- We use variables to **remember** values!
- Variable remembers a value via an assignment
- **name_of_variable** = **value_to_remember**
- Do not confuse **=** and **==** !

  = is **assignment** token such that *name_of_variable* = *value*

  == is operator to **test equality**
- Key property of a variable that **we can change its value**
- Naming convention: **with freedom comes responsibility!**

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



KEYWORDS

and	as	assert	break	class	continue
def	del	elif	else	except	exec
finally	for	from	global	if	import
in	is	lambda	nonlocal	not	or
pass	raise	return	try	while	with
yield	True	False	None		

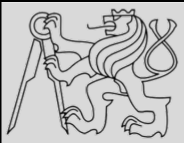
- Python keywords have **special** purpose
- Always choose names **meaningful** to human readers
- Use **comments (#)** and **blank lines** to improve readability

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

```
Python Console
/opt/local/bin/python3.6 /Applications/PyCharm.app/Contents/helpers
Python 3.6.2 (default, Sep 21 2017, 00:54:38)
In[2]: students = ['Anna', 'Bob', 'David', 'Mark', 'Brandon']
   ...: 
   ...: for student in students:
   ...:     if len(student) >= 5:
   ...:         print(student)
   ...: 
David
Brandon

In[3]:
```

- Statement is an **instruction** executable in Python
- Statements **do not produce any results**
- So far only assignment statements **=**
- Statement examples: *for, in, if …*

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



EXPRESSIONS

Python Console

```
/opt/local/bin/python3.6 /Applications/PyCharm.app/Contents/helpers
Python 3.6.2 (default, Sep 21 2017, 00:54:38)
In[2]: students = ['Anna', 'Bob', 'David', 'Mark', 'Brandon']
   ...:
   ...: for student in students:
   ...:     if len(student) >= 5:
   ...:         print(student)
   ...:
David
Brandon

In[3]:
```

- Expression is a combination of **values**, **variables**, **operators**, and **calls** to functions
- Built-in Python functions: *len, type, print*
- Value by itself is an expression
- Expression **produces result** (right side of an assignment)

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

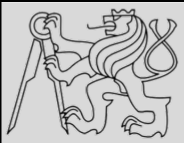
Python Console

```
/opt/local/bin/python3.6 /Applications/PyCharm.app/
Python 3.6.3 (default, Oct  5 2017, 23:34:28)
In[2]: minutes = 635
In[3]: hours = minutes / 60
In[4]: hours_floor_division = minutes // 60

In[5]:
```

```
Special Variables
_ = {str} ''
__ = {str} ''
___ = {str} ''
hours = {float} 10.583333333333334
hours_floor_division = {int} 10
minutes = {int} 635
```

- **OPERAND OPERATOR OPERAND**
- Operators are **special tokens** that represent computations like addition, subtraction, multiplication, division etc.
- The values the operator uses are called **operands**
- *When a variable name appears in the place of an operand, it is replaced with its value before the operation is performed*
- Division / vs *floor* division //

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



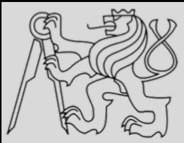
TYPE CONVERSION

```
>>> int(3.14)
3
>>> int(3.9999)          # This doesn't round to the closest int!
3
>>> int(3.0)
3
>>> int(-3.999)          # Note that the result is closer to zero
-3
>>> int(minutes / 60)
10
>>> int("2345")          # Parse a string to produce an int
2345
>>> int(17)              # It even works if arg is already an int
17
>>> int("23 bottles")
```

```
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '23 bottles'
```

- Functions, **int()**, **float()** and **str()** convert their arguments into types **int**, **float** and **str** respectively.
- The type converter **float()** can turn an **integer**, a **float**, or a syntactically legal **string** into a float
- The type converter **str()** turns its argument into a string
- One symbol can have different meaning depending on the data type(s) - **try** & **explore** & **understand**

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



ORDER OF OPERATIONS – PEMDAS

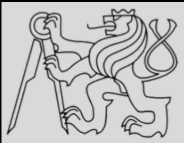

```
Python Console
/opt/local/bin/python3.6 /Applications/PyCharm.app/Contents/helpers/pydev
Python 3.6.3 (default, Oct  5 2017, 23:34:28)
In[2]: 2 ** 3 ** 2     # The right-most ** operator gets done first!
Out[2]: 512
In[3]: (2 ** 3) ** 2   # Use parentheses to force the order you want!
Out[3]: 64
```

- Evaluation depends on the rules of precedence:

1. **P**arentheses (for order, readability)
2. **E**xponentiation
3. **M**ultiplication and **D**ivision
4. **A**ddition and **S**ubtraction

- Order **left-to-right** evaluation on the same level, with the exception of exponentiation (**)

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



OPERATIONS ON STRINGS

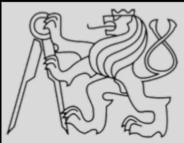

```
>>> message - 1          # Error
>>> "Hello" / 123        # Error
>>> message * "Hello"    # Error
>>> "15" + 2             # Error
```

```
Python Console
/opt/local/bin/python3.6 /Applications/PyCharm.app/Contents/helpers/pyc
Python 3.6.3 (default, Oct  5 2017, 23:34:28)
In[2]: name = "Bob"
In[3]: age = 17
In[4]: description = "My name is " + name + "and my age is "+ str(age)
In[5]: print(description)
My name is Boband my age is 17

In[6]:
```

```
Special Variables
_ = {str} ''
__ = {str} ''
___ = {str} ''
age = {int} 17
description = {str} 'My name is Boband my age is 17'
name = {str} 'Bob'
```

- You cannot perform mathematical operations on strings, even if the strings look like numbers
- The **+** operator represents **concatenation**, not addition
- The ***** operator also works on strings; it performs **repetition** (*one of the operands has to be a string; the other has to be an integer*)

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



PROGRAM STRUCTURE

Global

Function definitions

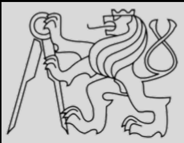
Main section

example.py

```python
# import modules used here -- sys is a very standard one
import sys


# Gather our code in a main() function
def main():
    print('Hello there', sys.argv[1])
    # Command line args are in sys.argv[1], sys.argv[2] ...
    # sys.argv[0] is the script name itself and can be ignored


# Standard boilerplate to call the main() function to begin
# the program.
if __name__ == '__main__':
    main()
```

- When python interpreter runs a source file as main program, it sets **__name__** variable to have a value **"__main__"**
- If being imported from another module, **__name__** will be set to the **module's name**

source https://developers.google.com/edu/python/introduction

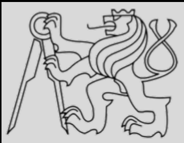


FUNCTION DEFINITION

```
def NAME( PARAMETERS ):
    STATEMENTS
```

- **Function** = named sequence of statements belonging together
- **Header line**: begins with a keyword **def**, ends with a colon **:**
- **Body**: one or more statements, each indented the same amount
- **Parameter list**: empty or any number of comma separated parameters (can have default value)
- Any **name** except for keywords and illegal identifiers
- Any **number of statements** inside the function, but **indented** from the **def** (standard indentation of **four spaces**)
- Function may (**fruitful**) or may not (**modifier**) produce a result

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



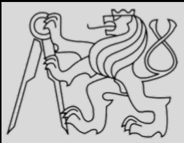
FLOW OF EXECUTION

Swapping variables

```
Python Console
/opt/local/bin/python3.6 /Applications/PyCharm.a
Python 3.6.3 (default, Oct  5 2017, 23:34:28)
In[2]: x = 7
In[3]: y = 10
In[4]: x, y = y, x
In[5]: print(x)
10
In[6]: print(y)
7
```

- Flow of execution = **order of statements execution** (*begins at the first statement of the program*)
- Statements are executed **one at a time**, in order **from top to bottom** (but **read the flow**, not top to bottom!)
- Python evaluates **expressions from left to right** (*during assignment right-hand side is evaluated first*)
- Function calls are like a **detour** in the flow of execution
- We can define one function inside another
- Function or class definitions **do not alter flow of execution**

source http://docs.python.org/3/reference/expressions.html#evaluation-order



THE FOR LOOP

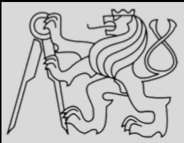

```
Python Console
/opt/local/bin/python3.6 /Applications/PyCharm.ap
Python 3.6.3 (default, Oct  5 2017, 23:34:28)
In[2]: for number in range(5):
   ...:     print(number)
   ...:
0
1
2
3
4
```

```
Special Variables
_ = {str} ''
__ = {str} ''
___ = {str} ''
number = {int} 4
```

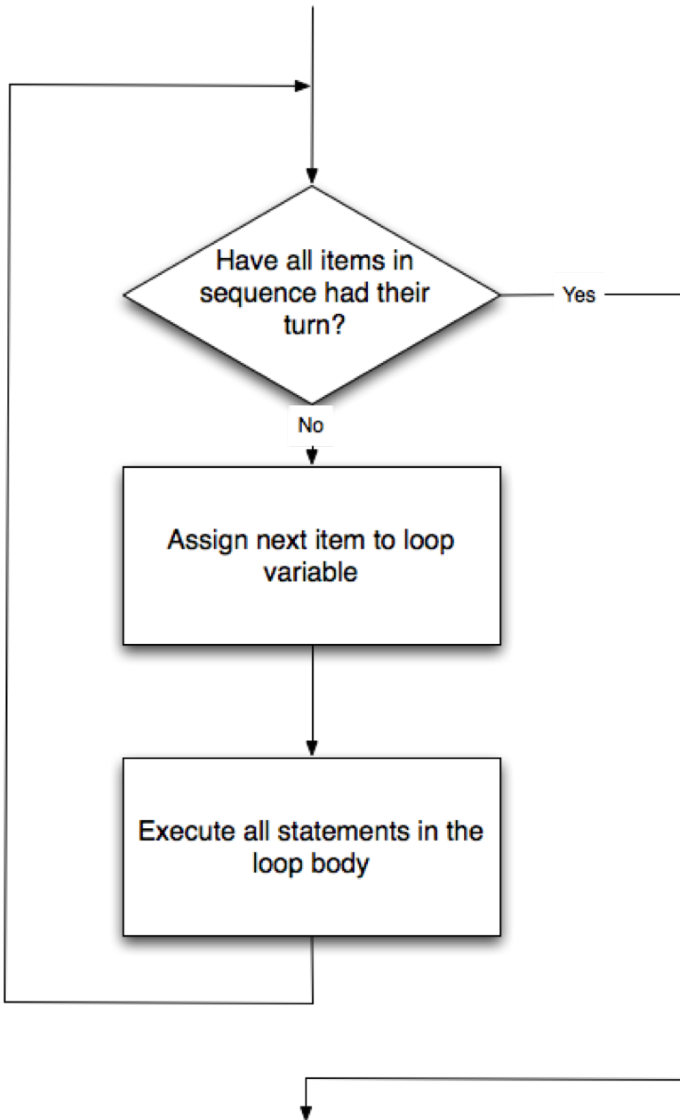
On each iteration or pass of the loop:

- Check to see if there are still more **items to be processed**
- If there are none left (the **terminating condition** of the loop) the loop has finished
- If there are items still to be processed, the **loop variable is updated** to refer to the next item in the list
- Program execution **continues at the next statement** after the loop body
- To explore: early **break**, or **for – else** loop

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



THE FOR LOOP – CONTROL FLOW



- Control flow (control of the flow of execution of the program)
- As program executes, the interpreter **always keeps track** of which statement is about to be executed
- Control flow until now has been strictly **top to bottom**, one statement at a time, **the for loop changes this!**

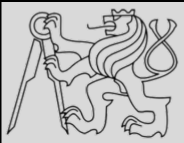
source http://openbookproject.net/thinkcs/python/english3e/hello_little_turtles.html

```
>>> 5 == (3 + 2)   # Is five equal 5 to the result of 3 + 2?
True
>>> 5 == 6
False
>>> j = "hel"
>>> j + "lo" == "hello"
True
```

```
x == y               # Produce True if ... x is equal to y
x != y               # ... x is not equal to y
x > y                # ... x is greater than y
x < y                # ... x is less than y
x >= y               # ... x is greater than or equal to y
x <= y               # ... x is less than or equal to y
```

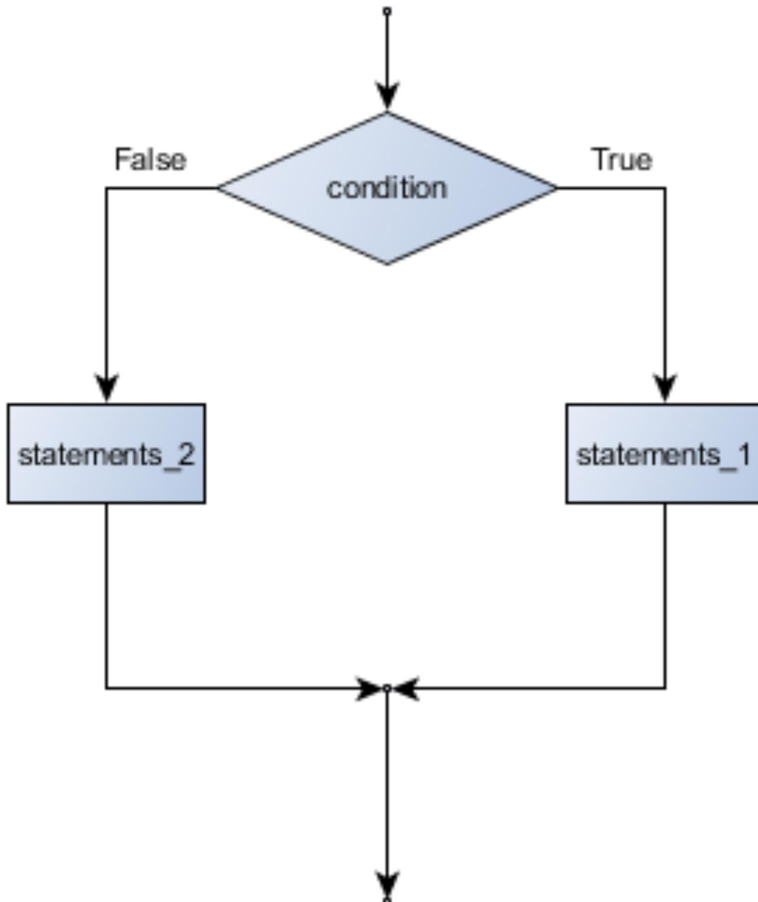
- Boolean expression is an expression that evaluates to produce a result which is a **Boolean value**
- Six common **comparison operators** which all produce a bool result (different from the mathematical symbols)

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



CONDITIONAL EXECUTION

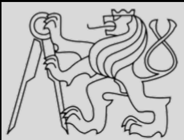

```
1  if BOOLEAN EXPRESSION:
2      STATEMENTS_1        # Executed if condition evaluates to True
3  else:
4      STATEMENTS_2        # Executed if condition evaluates to False
```




```
1  if True:
2      pass
3  else:
4      pass
```

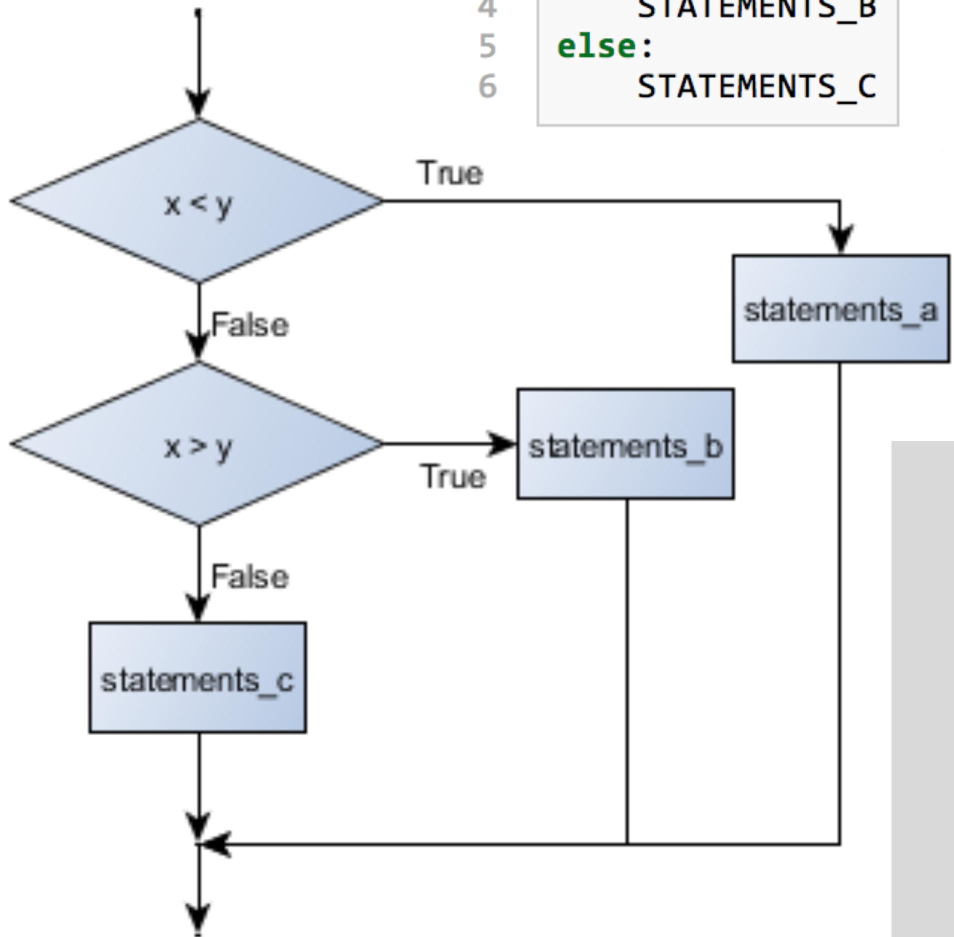
- Condition **IF – ELSE**
- Conditional statement – the ability to check conditions and change the behavior of the program accordingly

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



CONDITIONAL EXECUTION

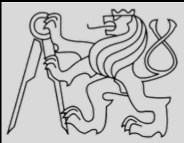
```python
if x < y:
    STATEMENTS_A
elif x > y:
    STATEMENTS_B
else:
    STATEMENTS_C
```




```python
if choice == "a":
    function_one()
elif choice == "b":
    function_two()
elif choice == "c":
    function_three()
else:
    print("Invalid choice.")
```

- Condition chaining
  IF – ELIF – ELSE
- Recommendation: handle all distinctive options by separate condition, use else to handle all other

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




```python
def sum_to(n):
    """ Return the sum of 1+2+3 ... n """
    ss  = 0
    v = 1
    while v <= n:
        ss = ss + v
        v = v + 1
    return ss

# For your test suite
test(sum_to(4) == 10)
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

```
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)
```

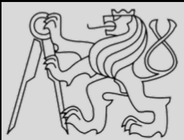
```
1   def sum_to(n):
2       """ Return the sum of 1+2+3 ... n """
3       ss  = 0
4       for v in range(n+1):
5           ss = ss + v
6       return ss
```

- 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

- 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 …

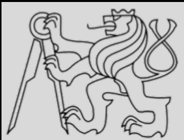
Source http://www.tutorialspoint.com/python/python_loop_control.htm

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n/x)
            break
```

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print( n, 'equals', x, '*', n/x)
            break
    else:
        # loop fell through without finding a factor
        print(n, 'is a prime number')
```

- Early return / early break
- Can be used to speed-up code execution
- Special condition: **FOR – ELSE**

source http://book.pythontips.com/en/latest/for_-_else.html



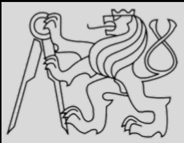
EXAMPLE

example.py

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

# import modules used here -- sys is a very standard one
import sys


# Gather our code in a main() function
def main():
    print('Hello there', sys.argv[0])
    # Command line args are in sys.argv[1], sys.argv[2] ...
    # sys.argv[0] is the script name itself and can be ignored
    for n in range(2, 10):
        print('n = ', n)
        for x in range(2, n):
            print('x = ', x)
            if n % x == 0:
                print(n, 'equals', x, '*', n // x)
                break
        else:
            # loop fell through without finding a factor
            print(n, 'is a prime number')


# Standard boilerplate to call the main() function to begin
# the program.
if __name__ == '__main__':
    main()
```

Run example

```
/opt/local/bin/python3.6 "/Users/mich
Hello there /Users/michalreinstein/Di
n =  2
2 is a prime number
n =  3
x =  2
3 is a prime number
n =  4
x =  2
4 equals 2 * 2
n =  5
x =  2
x =  3
x =  4
5 is a prime number
n =  6
x =  2
6 equals 2 * 3
n =  7
x =  2
x =  3
x =  4
x =  5
x =  6
7 is a prime number
n =  8
x =  2
8 equals 2 * 4
n =  9
x =  2
x =  3
9 equals 3 * 3

Process finished with exit code 0
```

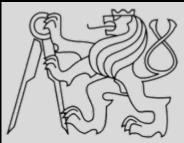


FUNCTIONS CALLING FUNCTIONS


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


def compute_area_rectangle(height, width):
    # use assert as function guard
    assert height >= 0 and width >= 0, 'Length cannot be negative'
    return height * width


def compute_area_square(side_length):
    return compute_area_rectangle(side_length, side_length)


if __name__ == '__main__':
    square_side_length = float(input('Input square side length (m)'))
    print(compute_area_square(square_side_length))

```

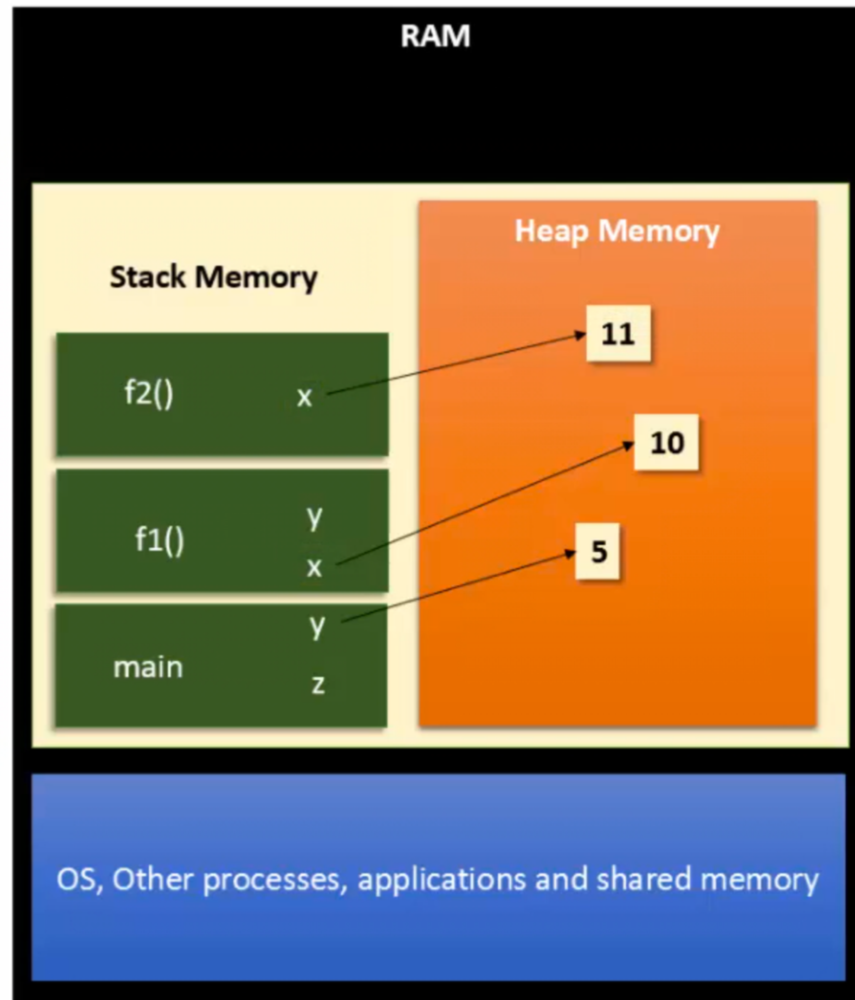
- Function **hide complex computation** behind a single command and capture abstraction of the problem.
- Functions can **simplify** a program
- Creating a new function can make a **program shorter** by eliminating **repetitive code**

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



FUNCTIONS CALLING FUNCTIONS

```
1  #!/usr/bin/env python
2
3
4  def f1(x):
5      x *= 2
6      y = f2(x)
7      return y
8
9
10 def f2(x):
11     x += 1
12     return x
13
14
15 if __name__ == '__main__':
16     y = 5
17     z = f1(y)
18     print(z)
19
```



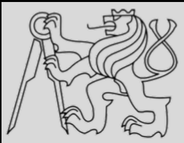

```
4  def f1(x):
5      x *= 2
6      y = f2(x)
7      return y
```

Shadows name 'y' from outer scope less... (⌘F1)

This inspection detects shadowing names defined in outer scopes.

```
12     return x
```

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



EXAMPLE

example_02.py | example_03.py | example_04.py

```python
def find_first_2_letter_word(xs):
    for wd in xs:
        if len(wd) == 2:
            return wd
    return "Nothing found"


print(find_first_2_letter_word(["This", "is", "a", "dead", "parrot"]))
print(find_first_2_letter_word(["I", "like", "cheese"]))
```

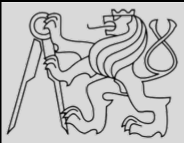
Run example_04

```
/opt/local/bin/python3.6 "/Users/mi
is
Nothing found

Process finished with exit code 0
```

- Return statement in the middle of a **for** loop – control **immediately returns** from the function

- EXAMPLE: *Let us assume that we want a function which looks through a list of words. It should return the first 2-letter word. If there is not one, it should return "Nothing found"*

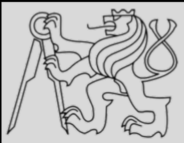
source http://openbookproject.net/thinkcs/python/english3e/fruitful_functions.html



MORE ABOUT PYTHON

- The methods and variables are created on **stack memory**
- The objects / 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 (unreferenced) 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



TUPLES

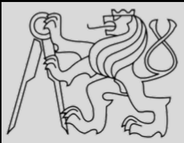
```
>>> julia = ("Julia", "Roberts", 1967, "Duplicity", 2009, "Actress", "Atlanta, Georgia")
```

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

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

- 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



INDEXING

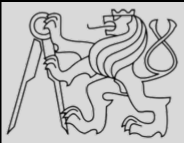
```
>>> fruit = "banana"
>>> m = fruit[1]
>>> print(m)
```

```
>>> 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: *expression fruit[1] selects second character from fruit, and creates new string containing this 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

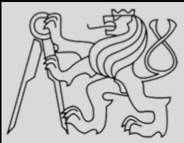



```
>>> my_string = "TEST"
>>> my_string[2] = "X"
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

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

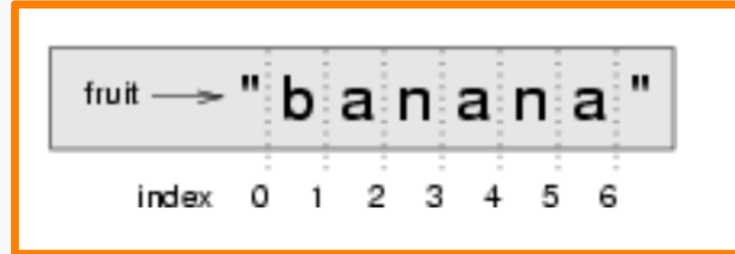
- Lists are **mutable** (we can change their elements)
- Strings are **immutable** (we cannot change their elements)
- Use **slicing principles** (indexes in between characters / items)

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



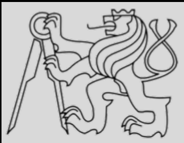
SLICING

```
>>> 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



STRINGS vs. LISTS

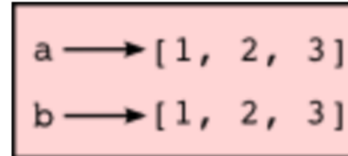
Strings

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

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

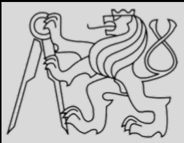
Lists

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



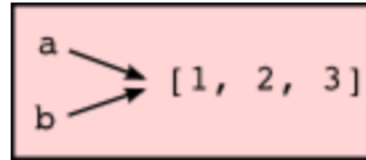
- Variables **a** and **b** refer to string object with letters "banana”
- Use **is** operator or **id** function to find out the **reference**
- Strings are **immutable**
- Not the case of lists: **a** and **b** have the same value (content) but do not refer to the same object

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



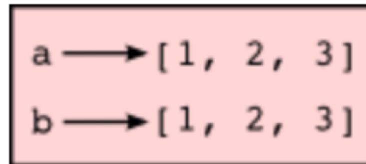
LISTS – ALIASING, CLONING

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




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

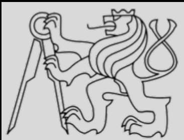
```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b
[1, 2, 3]
```




```
>>> 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 a creates a new list)

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

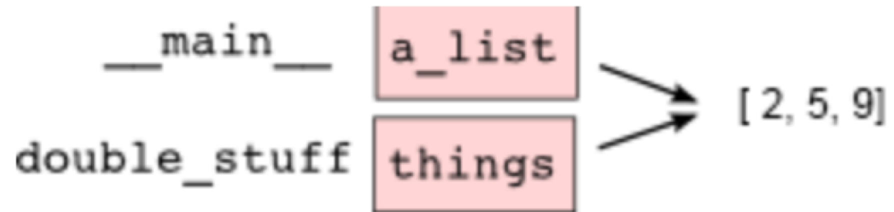


IMPORTANT – LIST PARAMETERS

```
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
```

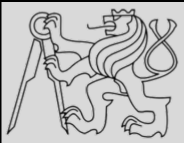
```
1  things = [2, 5, 9]
2  double_stuff(things)
3  print(things)
```

```
[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



MODULES – NAMESPACES

- Namespace is a **mapping from names to objects**
- Namespace is a **collection of identifiers** that belong to a *module*, *function*, or a *class*
- Namespace is **set of symbols** used to **organize objects** of various kinds so that can be **referred by name**
- Namespaces permit programmers to work on the same project without having **naming collisions** (*allow name reuse*)
- Often **hierarchically** structured
- Each name must be **unique in its namespace**
- Namespace is very general concept not limited to Python
- Each **module has its own namespace** – *we can use the same identifier name in multiple modules without causing an identification problem*

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

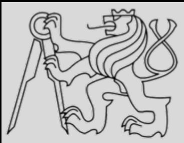
How are namespaces defined in Python?

- **Packages** (*collections of related modules*)
- **Modules** (*.py files containing definitions of functions, classes, variables, etc.)*
- **Classes, Functions** …

What is the difference between programs and modules?

- Both are stored in **.py files**.
- **Programs** (*scripts*) are designed to be executed
- **Modules** (*libraries*) are designed to be imported and used by other programs and other modules
- **Special case**: *.py* file is designed to be both a program and a module *(it can be executed as well as imported to provide functionality for other modules)*

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




```
# Module1.py

question = "What is the meaning of Life, the Universe, and Everything?"
answer = 42
```

```
# Module2.py

question = "What is your quest?"
answer = "To seek the holy grail."
```

```
import module1
import module2

print(module1.question)
print(module2.question)
print(module1.answer)
print(module2.answer)
```

```
What is the meaning of Life, the Universe, and Everything?
What is your quest?
42
To seek the holy grail.
```

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

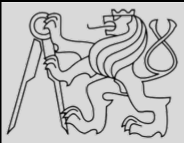
```
1  def f():
2      n = 7
3      print("printing n inside of f:", n)
4
5  def g():
6      n = 42
7      print("printing n inside of g:", n)
8
9  n = 11
10 print("printing n before calling f:", n)
11 f()
12 print("printing n after calling f:", n)
13 g()
14 print("printing n after calling g:", n)
```

```
printing n before calling f: 11
printing n inside of f: 7
printing n after calling f: 11
printing n inside of g: 42
printing n after calling g: 11
```

- Functions also have own namespaces created on function call
- Functions can read (**read-only**) variables in the outer scope

- EXAMPLE: *the three **n**'s above do not collide since they are each in a different namespace — three names for three different variables*

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



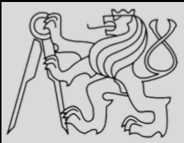
MODULES, NAMESPACES, FILES

- Python has a convenient and simplifying one-to-one mapping: **one module per file** – giving rise to **one namespace**
- Python takes the **module name from the file name**, and this becomes the **name of the namespace**

- EXAMPLE: math.py is a **filename**, the module is called **math**, and its namespace is **math**
(*in Python the concepts are more or less interchangeable*)

- NOTE: In other languages (e.g. C#) one module can span multiple files, or one file to have multiple namespaces, or many files to all share the same namespace

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



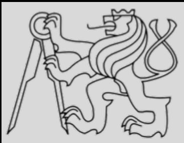
MODULES – SCOPE

- A **scope** is a *textual region of a Python program where a namespace is directly accessible*

What types of scopes can be defined?

- **Local scope** refers to identifiers declared **within a function** *(these identifiers are kept in the namespace that belongs to the function, and each function has its own namespace, local scope is created with each function call)*
- **Global scope** refers to all the identifiers declared **within the current module, or file**
- **Built-in scope** refers to all the identifiers built into Python *(those like range and min that can be used without having to import anything*)

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



MODULES – SCOPE

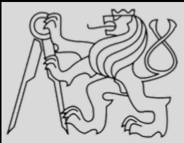

```
1  def range(n):
2      return 123*n
3
4  print(range(10))
```

```
1  n = 10
2  m = 3
3  def f(n):
4      m = 7
5      return 2*n+m
6
7  print(f(5), n, m)
```

What are the scope precedence rules?

- The *same name can occur in more than one* of these scopes, but the innermost, or local scope, will always take precedence over the global scope, and the global scope always gets used in preference to the built-in scope
- Names can be "**hidden**" from use if own variables or functions reuse those names
- EXAMPLE: *variables **n** and **m** are created just for the duration of the execution of **f** since they are created in the local namespace of function **f** (precedence rules apply)*

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



MODULES – THE DOT OPERATOR


```
1  import math
2  x = math.sqrt(10)
```

```
1  from math import cos, sin, sqrt
2  x = sqrt(10)
```

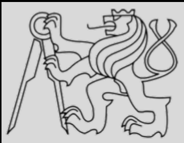
```
1  def area(radius):
2      import math
3      return math.pi * radius * radius
4
5  x = math.sqrt(10)      # This gives an error
```

```
1  from math import *    # Import all the identifiers from math,
2                        #   adding them to the current namespace.
3  x = sqrt(10)          # Use them without qualification.
```

```
1  >>> import math as m
2  >>> m.pi
3  3.141592653589793
```

- Variables defined **inside a module** are called **attributes** of the module (similar to class attributes)
- Attributes are accessed using the **dot** operator (**.**)

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



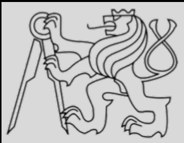

```
>>> eng2sp = {}
>>> eng2sp["one"] = "uno"
>>> eng2sp["two"] = "dos"
```

```
>>> print(eng2sp)
{"two": "dos", "one": "uno"}
```

- Strings, lists, and tuples — are sequence types *using integers as indices* to access the values they contain within them
- Dictionaries are Python's built-in **mapping type**
- They map **keys** (any **immutable** type) to **values** that can be any type (**heterogeneous**)
- The **empty dictionary** is denoted **{}**

- EXAMPLE: *Create a dictionary to translate English words into Spanish (the keys are strings). One way to create a dictionary is to start with the empty dictionary and add key : value pairs.*

source courtesy of Petr Posik BE5b33PR 2016/2017



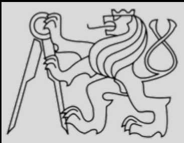
DICTIONARIES


```
Python Console
/opt/local/bin/python2.7 /Applications/PyCharm.app/Conte
Python 2.7.14 (default, Sep 27 2017, 12:15:00)
In[2]: keys = ['a', 'b', 'c']
In[3]: values = [1, 2, 3]
In[4]: my_dict = dict(zip(keys, values))
In[5]: print(my_dict)
{'a': 1, 'c': 3, 'b': 2}

In[6]:
```

```
▶ Special Variables
   _ = {str} ''
   __ = {str} ''
   ___ = {str} ''
▶ keys = {list} <type 'list'>: ['a', 'b', 'c']
▼ my_dict = {dict} {'a': 1, 'c': 3, 'b': 2}
     'c' (4555205408) = {int} 3
     'b' (4555203808) = {int} 2
     __len__ = {int} 3
     'a' (4555203768) = {int} 1
▶ values = {list} <type 'list'>: [1, 2, 3]
```

- Keys and values can be defined as separate lists (*order matters!*)
- Lists can be paired using **zip**
- Once paired a dictionary can be created using **dict**

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



FILES


```
1  myfile = open("test.txt", "w")
2  myfile.write("My first file written from Python\n")
3  myfile.write("---------------------------------\n")
4  myfile.write("Hello, world!\n")
5  myfile.close()
```

- To **store data** into a file we invoke the **write method** on the **handle** (*lines 2, 3 and 4*)
- NOTE*: Lines 2 – 4 should usually be replaced by a loop that writes more lines into the file, i.e. the content we want to store*
- *Line 5*: **closing** the file handle tells the system that writing the content is finished and makes the disk file available for reading by other programs

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

```
1  mynewhandle = open("test.txt", "r")
2  while True:                              # Keep reading forever
3      theline = mynewhandle.readline()     # Try to read next line
4      if len(theline) == 0:                # If there are no more lines
5          break                            #     leave the loop
6
7      # Now process the line we've just read
8      print(theline, end="")
9
10 mynewhandle.close()
```

- EXAMPLE: reading a file **line-at-a-time** using the mode argument is **"r"** for reading and method **readline()**
- More extensive logic into the body of the loop at line 8

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

```
f = open("somefile.txt")
content = f.read()
f.close()

words = content.split()
print("There are {0} words in the file.".format(len(words)))
```

- EXAMPLE: reading the **whole file at once** using method **read()**
- Read the complete *contents of the file into a string*, and then to use string-processing skills to work with the contents
- Not interested in the line structure of the file

- EXAMPLE: use the **split** method on strings which can break a string into words (*e.g. counting the number of words in a file*)

- NOTE: the **"r"** mode in line 1 is omitted since **by default** Python opens the file for reading

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

```python
def sum_number_pairs(infname, outfname):
    """Read data from input file, sum each row, write results to output
 file.

    (str, str) -> None

    infname: the name of the input file containing a pair of numbers
             separated by whitespace on each line
    outfname: the name of the output file
    """
    with open(infname, 'r', encoding='utf-8') as infile, \
         open(outfname, 'w', encoding='utf-8') as outfile:
        for pair in infile:
            pair = pair.strip()
            operands = pair.split()
            total = float(operands[0]) + float(operands[1])
            new_line = '{} {}\n'.format(pair, total)
            outfile.write(new_line)
```

When called, this function creates the required output file containing the sums.

```python
sum_number_pairs('number_pairs.txt', 'number_pairs_with_totals.txt')
!cat number_pairs_with_totals.txt
```

```
1 1 2.0
10 20 30.0
1.3 2.7 4.0
```

source courtesy of Petr Posik BE5b33PR 2016/2017

```
1  p = Point()
2  p.x = 7
3  p.y = 6
```

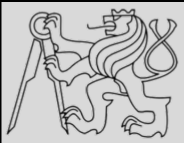
```
1  class Point:
2      """ Point class represents and manipulates x,y coords. """
3
4      def __init__(self, x=0, y=0):
5          """ Create a new point at x, y """
6          self.x = x
7          self.y = y
8
9  # Other statements outside the class continue below here.
```

```
>>> p = Point(4, 2)
>>> q = Point(6, 3)
>>> r = Point()          # r represents the origin (0, 0)
>>> print(p.x, q.y, r.x)
4 3 0
```

- EXAMPLE: to create a point (*instance of class Point*) at position (7, 6) currently needs three lines of code

- Make class initializer more general by adding parameters into the __init__ method
- The x and y parameters here are optional (default values of 0)

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

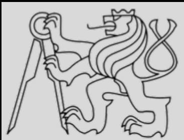


CLASS vs. TUPLE

```
1  class Point:
2      """ Create a new Point, at coordinates x, y """
3
4      def __init__(self, x=0, y=0):
5          """ Create a new point at x, y """
6          self.x = x
7          self.y = y
8
9      def distance_from_origin(self):
10         """ Compute my distance from the origin """
11         return ((self.x ** 2) + (self.y ** 2)) ** 0.5
```

- Advantage of using a class (*e.g. Point*) rather than a tuple is that **class methods are sensible operations** for points, but may not be appropriate for other tuples (*e.g. calculate the distance from the origin*)
- Class allows to **group together sensible operations** as well as data to apply the methods on
- Each instance of the class has its **own state**
- Method **behaves like a function** but it is invoked on a specific instance

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



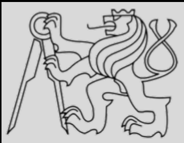
OBJECT METHODS

```
>>> p = Point(3, 4)
>>> p.x
3
>>> p.y
4
>>> p.distance_from_origin()
5.0
>>> q = Point(5, 12)
>>> q.x
5
>>> q.y
12
>>> q.distance_from_origin()
13.0
>>> r = Point()
>>> r.x
0
>>> r.y
0
>>> r.distance_from_origin()
0.0
```

```
1  class Point:
2      """ Create a new Point, at coordinates x, y """
3
4      def __init__(self, x=0, y=0):
5          """ Create a new point at x, y """
6          self.x = x
7          self.y = y
8
9      def distance_from_origin(self):
10         """ Compute my distance from the origin """
11         return ((self.x ** 2) + (self.y ** 2)) ** 0.5
```

- First parameter of a method refers to the instance being manipulated (parameter **self**)
- The caller of **distance_from_origin** does not explicitly supply an argument to match the self parameter

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

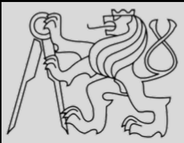


EXAMPLE – STATIC METHODS

```
1   class Person:
2       TITLES = ('Dr', 'Mr', 'Mrs', 'Ms')
3
4       def __init__(self, name, surname):
5           self.name = name
6           self.surname = surname
7
8       def fullname(self): # instance method
9           # instance object accessible through self
10          return "%s %s" % (self.name, self.surname)
11
12      @classmethod
13      def allowed_titles_starting_with(cls, startswith): # class method
14          # class or instance object accessible through cls
15          return [t for t in cls.TITLES if t.startswith(startswith)]
16
17      @staticmethod
18      def allowed_titles_ending_with(endswith): # static method
19          # no parameter for class or instance object
20          # we have to use Person directly
21          return [t for t in Person.TITLES if t.endswith(endswith)]
22
```

```
In[3]: jane = Person("Jane", "Smith")
In[4]: print(jane.fullname())
Jane Smith
In[5]: print(jane.allowed_titles_starting_with("M"))
['Mr', 'Mrs', 'Ms']
In[6]: print(Person.allowed_titles_starting_with("M"))
['Mr', 'Mrs', 'Ms']
In[7]: print(jane.allowed_titles_ending_with("s"))
['Mrs', 'Ms']
In[8]: print(Person.allowed_titles_ending_with("s"))
['Mrs', 'Ms']
```

SOURCE http://python-textbok.readthedocs.io/en/1.0/Classes.html# UNDER CC BY-SA 4.0 licence Revision 8e685e710775



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/

This lecture re-uses selected parts of the PYTHON TEXTBOOK

Object-Oriented Programming in Python

http://python-textbok.readthedocs.io/en/1.0/Classes.html#

(released under CC BY-SA 4.0 licence Revision 8e685e710775)