PRG – PROGRAMMING ESSENTIALS

Lecture 2 – Program flow, Conditionals, Loops
https://cw.fel.cvut.cz/wiki/courses/be5b33prg/start

Milan Nemy

Czech Technical University in Prague, Faculty of Electrical Engineering, Dept. of Cybernetics

https://beat.ciirc.cvut.cz/people/milan-nemy/milan.nemy@cvut.cz

- Problem formulation (input / output)
- Formalism (math?)
- Algorithm (steps)
- Implementation (engineering)
- Testing (are we good?)

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]:

my_age = {int} 17
my_height = {float} 183.5

my_name = {str} 'Bob'
```

- We use variables to remember things!
- The assignment statement gives a value to a variable
- 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!

VARIABLES



n p

```
cannot begin with a number

>>> 76trombones = "big parade"

SyntaxError: invalid syntax

>>> more$ = 1000000

SyntaxError: invalid syntax

>>> class is reserved keyword

SyntaxError: invalid syntax

>>> class = "Computer Science 101"

SyntaxError: invalid syntax
```

- The longer life the longer name: very_long_name_of_my_var
- The more important the longer name
- Meaningful name does not add the meaning just by itself, the code must do this!
- Illegal name causes a syntax error
- Capitals: Variable vs variable



KEYWORDS



m p

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



BUILT-IN FUNCTIONS



m p

		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	import()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

- Built-in functions have special purpose
- Study https://docs.python.org/3.4/library/functions.html

DATA TYPES



7

```
Python 3.6.3 (default, .0ct ..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 """)

OPERATORS & OPERANDS



m p

```
/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

X In [4]: .hours_floor_division = .minutes // .60

In [5]:

In [5]:

minutes = {str} "

hours = {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 //

ORDER OF OPERATIONS – PEMDAS



```
/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
X In[3]: (2 ** 3) ** 2 ... # Use parentheses to force the order you want!
Out[3]: 64
```

- Evaluation depends on the rules of precedence:
- Parentheses (for order, readability)
- 2. Exponentiation
- 3. Multiplication and Division
- 4. Addition and Subtraction
- Order left-to-right evaluation on the same level, with the exception of exponentiation (**)



MODULUS OPERATOR



m p

```
Python Console
   /opt/local/bin/python3.6 /Applications/PyCharm.app/Contents/helpers
                                                                                 🚟 Special Variables
   Python 3.6.3 (default, Oct 5 2017, 23:34:28)
                                                                                 器 _ = {str} "
       2]: total_secs = int(input("How many seconds, in total?"))
       ..: hours = total_secs // 3600
                                                                                 [ ] __ = {str} "
     secs_still_remaining.= total_secs.%.3600
      minutes = . secs_still_remaining.//.60
                                                                                 \mathbb{R} hours = {int} 58
           secs_finally_remaining = secs_still_remaining % 60
                                                                                 \mathbb{R} minutes = {int} 59
           print("Hrs=", hours, "..mins=", minutes,
                                                                                 secs_finally_remaining = {int} 5
                                      "secs=". secs finally remaining)
                                                                                 secs_still_remaining = {int} 3545
    How many seconds, in total?>7.212345
    Hrs= .58 . . .
              mins= 59 secs= 5
                                                                                 III total_secs = {int} 212345
```

- The modulus operator works on integers (integer expressions)
- Definition: modulus is the remainder when the first number is divided by the second
- Modulus operator is a percent sign %
- Syntax is the same as for other operators
- The same precedence as the multiplication operator



TYPE CONVERSION



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

OPERATIONS ON STRINGS



- 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)

13

Built-in function to get input from a user:

input("Message to the user!")

- User input is stored as string
- Combine with type conversion

COMPOSITION

n p

```
Python Console
  /opt/local/bin/python3.6/Applications/PyCharm.app/Contents/helpers/pydev/pydevcd
                                                                                              B Special Variables
   Python 3.6.3 (default, Oct 5 2017, 23:34:28)
    <u>In[2]:.response.=.input("What.is.your.radius?.")</u>
        .:.r.=.float(response)
       ..: area = 3.14159 * r**2
          .print("The area is ", area)
                                                                                              380.13239 area = {float}
   What is your radius? >7.11
   The area is 380.13239
                                                                                              |R| r = \{float\} 11.0
    [n[3]: r.=.float(.input("What.is.your.radius?.").)
                                                                                              \mathbb{R} response = {str} '11'
           print("The area is ", 3.14159 * r**2)
   What is your radius? >7.11
   The area is 380.13239
          .print("The.area.is.",.3.14159*float(input("What.is.your.radius?"))**2)
   What is your radius?>7.11
   The area is 380,13239
```

- Combination of the elements of a program: variables, expressions, statements, and function calls
- One of the most useful features of programming languages
- Take small building blocks and compose them into larger chunks

THE FOR LOOP



```
Python Console
  /opt/local/bin/python3.6 /Applications/PyCharm.app/Contents/helpers/pydev/pydevconsol
                                                                                            Special Variables
        :.for.friend.in.["Joe",."Zoe",."Brad",."Angelina",."Zuki",."Thandi",."Paris"]:
               invite = "Hi.".+.friend .+."...Please.come.to.my.party.on.Saturday!"
                                                                                               ₩ __ = {str} "
              print(invite)
                                                                                                   ___ = {str} ''
                                                                                                III friend = {str} 'Paris'
  Hi Joe. Please come to my party on Saturday!
   Hi Zoe. Please come to my party on Saturday!
                                                                                                Invite = {str} 'Hi Paris. Please come to my party on Saturday!'
   Hi Brad. Please come to my party on Saturday!
Hi Angelina. Please come to my party on Saturday!
   Hi Zuki. Please come to my party on Saturday!
Hi Thandi. Please come to my party on Saturday!
   Hi Paris. Please come to my party on Saturday!
```

- The variable friend at line 1 is the loop variable
- Lines 2 and 3 are the loop body
- The loop body is always indented
- The <u>indentation</u> determines exactly what <u>statements</u> are "in the body of the loop"
- At the end of each execution of the body of the loop, Python returns to the for statement, to see if there are more items to be handled, and to assign the next one to the loop variable

THE FOR LOOP

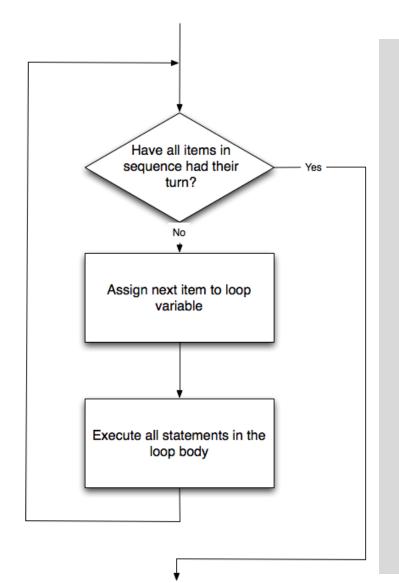


16

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

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!

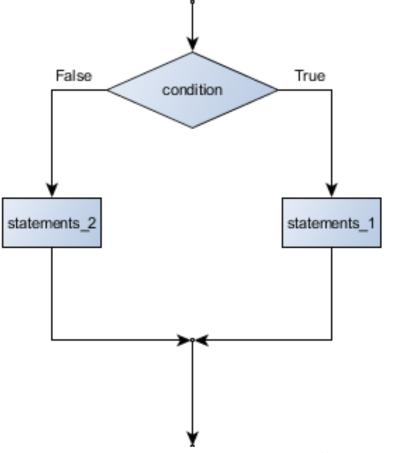




m p

18

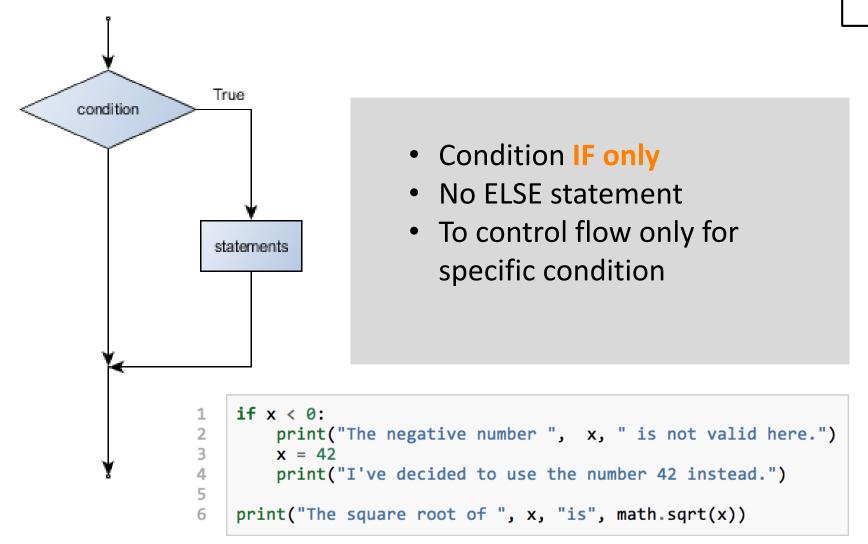
```
if BOOLEAN EXPRESSION:
    STATEMENTS_1  # Executed if condition evaluates to True
else:
    STATEMENTS_2  # Executed if condition evaluates to False
```



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

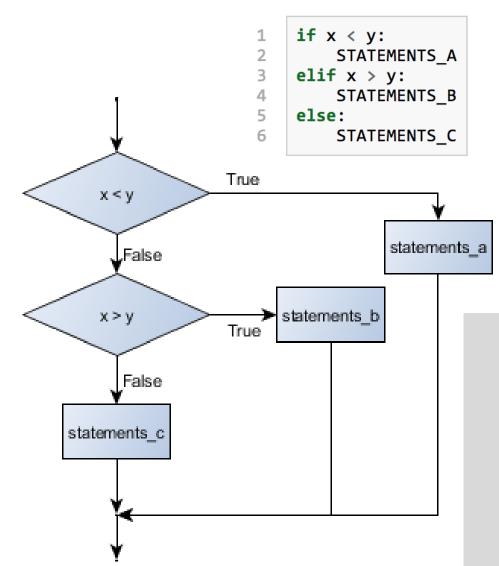
source http://openbookproject.net/thinkcs/python/english3e/conditionals.html







m p



```
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





m p

```
if 0 < x:  # Assume x is an int here
  if x < 10:
    print("x is a positive single digit.")</pre>
```

```
False
                                                     True
                                     x < y
                               True
     False
                                                   statements a
                   x > y
statements_c
                            statements b
```

```
1  if x < y:
2    STATEMENTS_A
3  else:
4    if x > y:
5        STATEMENTS_B
6    else:
7    STATEMENTS_C
```

- Nesting conditions builds hierarchy of decisions (decision trees)
- Nesting may reduce readability and clarity





```
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

BOOLEAN VALUES & EXPRESSIONS

```
>>> type(True)
<class 'bool'>
>>> type(true)
Traceback (most recent call last):
   File "<interactive input>", line 1, in <module>
NameError: name 'true' is not defined
```

- Test conditions and change the program behavior depending on the outcome of the tests
- Boolean value is either True or False
- Named after the British mathematician, George Boole, who first formulated Boolean algebra



BOOLEAN VALUES & EXPRESSIONS



m p

```
>>> 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</pre>
```

- 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)



LOGICAL OPERATORS



m p

- three logical operators, and, or, and not, that allow to build more complex expressions from simple Boolean expressions
- semantics (meaning) of these operators is similar to natural language equivalent

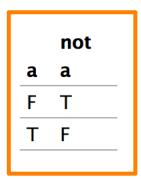


TRUTH TABLES

٠,	L
/	O
_	_

		a and
a	b	b
False	False	False
False	True	False
True	False	False
True	True	True

		a or
a	b	b
F	F	F
F	Т	Т
Т	F	Т
Т	Т	Т



Short-circuit evaluation:

- OR if the expression on the left of the operator yields True,
 Python does not evaluate the expression on the right
- AND if the expression on the left yields False, Python does not evaluate the expression on the right.
- Truth table list of all the possible inputs to give the results for the logical operators



BOOLEAN ALGEBRA – LOGIC OPPOSITES



m p

```
operator logical opposite
```

- Each of the six relational operators has a logical opposite
- Recommendation: not operators may reduce readability, use logical opposites instead



BOOLEAN ALGEBRA



m p

```
n * 0 == 0
```

```
x and False == False
False and x == False
y and x == x and y
x and True == x
True and x == x
x and x == x
```

```
x or False == x
False or x == x
y or x == x or y
x or True == True
True or x == True
x or x == x
```

$$not (not x) == x$$

DE MORGAN'S LAWS



```
not (x \text{ and } y) == (\text{not } x) \text{ or } (\text{not } y)
not (x \text{ or } y) == (\text{not } x) \text{ and } (\text{not } y)
```

```
if not ((sword_charge >= 0.90) and (shield_energy >= 100)):
    print("Your attack has no effect, the dragon fries you to a crisp!")
else:
    print("The dragon crumples in a heap. You rescue the gorgeous princess!")
```

- De Morgan's laws rules allow the expression of conjunctions and disjunctions in terms of each other via negation
- Example: suppose we can slay the dragon only if our magic sword is charged to 90% or higher and we have 100 or more energy units in our protective shield



DE MORGAN'S LAWS



m p

30

```
if (sword_charge < 0.90) or (shield_energy < 100):
    print("Your attack has no effect, the dragon fries you to a crisp!")
else:
    print("The dragon crumples in a heap. You rescue the gorgeous princess!")

if (sword_charge >= 0.90) and (shield_energy >= 100):
    print("The dragon crumples in a heap. You rescue the gorgeous princess!")
else:
    print("Your attack has no effect, the dragon fries you to a crisp!")
```

• Example: suppose we can slay the dragon only if our magic sword is charged to 90% or higher and we have 100 or more energy units in our protective shield

EXAMPLE



m p

31

p	q	r	(not (p and q)) or r
F	F	F	?
F	F	Т	?
F	Т	F	?
F	Т	Т	?
Т	F	F	?
Т	F	Т	?
Т	Т	F	?
Т	Т	Т	?

• Example: complete the table ..

REFERENCES

32

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 <u>GNU Free Documentation License</u> <u>Version 1.3</u>)

- 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/