True or False?

Basic Programming in Python

Sebastian Höffner Aline Vilks

Wed, 19 Apr 2017

How to search for math symbols

Remember last week's weird brackets?

$d(L, S) = \lfloor 5 + 1.15S + 0.1L \rfloor$

Whenever you come across a mathematical symbol you do not know, Wikipedia's List of mathematical symbols[1] might be useful.

Another option might be detexify[2].

[1] https://en.wikipedia.org/wiki/List_of_mathematical_symbols
[2] http://detexify.kirelabs.org/

How to search for math symbols

Remember last week's weird brackets?

$d(L, S) = \lfloor 5 + 1.15S + 0.1L \rfloor$

Whenever you come across a mathematical symbol you do not know, Wikipedia's List of mathematical symbols[1] might be useful.

Another option might be detexify[2].

[1] https://en.wikipedia.org/wiki/List_of_mathematical_symbols
[2] http://detexify.kirelabs.org/

Names and explanations of symbols can be looked up, examples included

Common mistakes & conventions: File names

Again, please name your files as we specify them. Bright side: this week we only got `*.py` files! (And sometimes supplementary material: documentation, cute images etc.)

Common mistakes & conventions: Whitespace

Please use spaces around math operators, after commas, and after #.

```
def area(base, side, height):
    return base * side + height * base / 2



# Calculates the St. Nick home area
print('My area:', area(5, 10, 3))
```

Output:

```
My area: 57.5
```

Common mistakes & conventions: Code order

Try to put functions definitions together to the top of your files

```python
def fun1():
    pass


def fun2():
    pass


# prints, calls, etc. here and not between the functions
```

Common mistakes: Variable names

Variable (and function) names should only use these characters:

```
import string

print(string.ascii_lowercase, string.digits, '_', sep='')
```

Output:

```
abcdefghijklmnopqrstuvwxyz0123456789_
```

They should not start with digits!

Common mistakes & conventions: Naming things

Variable names should (usually) tell us what is behind them

```python
def a(b, s, h):
    return b * s + b * h / 2

def area(base, side, height):
    return base * side + base * height / 2
```

Common mistakes & conventions: Random print statements

Try not to clutter print statements which just print some numbers

```
area = 41.3
side = 23.1
print(area)
print(side)
print('Area:', area)
print('Side:', side)
```

Output:

```
41.3
23.1
Area: 41.3
Side: 23.1
```

Common mistakes & conventions: `damage_taken`

For the castle crashers exercise you needed to call the `damage_taken` function for each individual hit.

Another data type: Boolean

There are only two things that can be expressed with the boolean data type:

- That something is **True**

and

- that something is **False**

There are only two things that can be expressed with the boolean data type:

- That something is **True**

and

- that something is **False**

Nevertheless it is an extremely useful and thus important concept in programming.

Another data type: Boolean

We can *assign* these values to variables. For example:

```
parrot_alive = True
```

Another data type: Boolean

We can *assign* these values to variables. For example:

```
parrot_alive = True
```

We assigned the value `True` to the placeholder `parrot_alive`.

Mind the spelling with a capital **T**!

Another data type: Boolean

And we can check whether an expression is *true* or *false*.

```
>>> 5 > 42
False
>>> 5 < 42
True
```

We can also check the truth value of previously assigned variables.

```
>>> parrot_alive = True
>>> parrot_alive
True
```

Comparison

We can compare numbers using the following operators:

Operator	Comparison	`True`	`False`
`==`	equal	`1 == 1`	`5 == 3`
`!=`	not equal	`2.3 != 2.313`	`5 != 5`
`<`	less than	`2.5 < 9`	`4 < 3`
`>`	greater than	`2.4 > 2.399`	`0.1 > 5`
`<=`	less than or equal	`3 <= 3`	`4 <= 3`
`>=`	greater than or equal	`2.4 >= 2.399`	`0 >= 5`

True or False?

Comparison

Comparison

We can compare numbers using the following operators:

Operator	Comparison	True	False
`==`	equal	`1 == 1`	`5 == 3`
`!=`	not equal	`2.3 != 2.313`	`5 != 5`
`<`	less than	`2.5 < 9`	`4 < 3`
`>`	greater than	`2.4 > 2.399`	`0.1 > 5`
`<=`	less than or equal	`3 <= 3`	`4 <= 3`
`>=`	greater than or equal	`2.4 >= 2.399`	`0 >= 5`

It is possible to compare strings the same way, but it follows less obvious rules.

Chaining

Comparisons can be chained, which is mostly useful for boundary checks:

```
>>> 1 < 2 <= 4 > 3 == 3 != 5
True
>>> 4 * 8 < 5 * 9 == 45 > 4.2 * 9 < 2
False
>>> a = 5
>>> 2 < a < 6  # This is a common application
True
```

Unrolling chained comparisons

Comparisons are done from left to right and "chained" with `and`.

```
>>> 1 < 5 < 4
False
>>> 1 < 5 and 5 < 4
False
```

Comparing `True` and `False`

What do you expect from the following three statements?

```
>>> (1 < 2) < 2
>>> True == 0
>>> False < True
```

True or False?

└─Comparing True and False

Comparing True and False

What do you expect from the following three statements?

```
>>> (1 < 2) < 2
>>> True == 0
>>> False < True
```

- `True` (`True < 2`)
- `False` (because `True == 1`)
- `True`

Careful! `True` is equal to `1`, and `1` only, but for `if` (next slide) every non-zero number is considered `True`!

Using truth values

What does the following code snippet do? What happens when you change the age to 23?

```
age = 17
if age >= 16:
    print('You may buy beer in Germany.')
if age >= 21:
    print('You may buy beer in the US.')
```

if-statements

`if` is the most basic control flow tool we have.

```
c = 4
if c < 5:
    c = 5
print(c)
```

Output:

```
5
```

Intermezzo: indentation

In Python lines of code with the same indentation level are considered a block.

We can not arbitrarily indent our code, but only after certain keywords, like `if`.

```
if condition:          # if this condition is True
    print('Hello')  # this line will be executed
    print('World')  # and this line as well
print('Good bye')   # this line will ALWAYS be executed
```

True or False?

Intermezzo: indentation

In Python lines of code with the same indentation level are considered a block.

We can not arbitrarily indent our code, but only after certain keywords, like if.

```
if condition:       # if this condition is True
    print('Hello')  # this line will be executed
    print('World')  # and this line as well
print('Good bye')   # this line will ALWAYS be executed
```

We always indent to the next level with four spaces.

`if` and `else`

Let's take a look at the Collatz conjecture.

$$f(x) = \begin{cases} x/2 & \text{if } x \text{ is even} \\ 3x + 1 & \text{if } x \text{ is odd} \end{cases} \tag{1}$$

Let's do it in Python!

Let's take a look at the Collatz conjecture.

$$f(x) = \begin{cases} x/2 & \text{if } x \text{ is even} \\ 3x+1 & \text{if } x \text{ is odd} \end{cases} \quad (1)$$

Let's do it in Python!

Often `if` is not enough, e.g. in the Collatz conjecture.

Collatz conjecture

```
def collatz(x):
    if x % 2 == 0:
        return x // 2
    else:
        return 3 * x + 1


x = 5
y = collatz(x)
print(y)
```

Output:

```
16
```

Collatz conjecture

```python
def collatz(x):
    if x % 2 == 0:
        return x // 2
    return 3 * x + 1

x = 5
y = collatz(x)
print(y)
```

Output:

```
16
```

What about more cases?

We can also use `elif`, short for `else if`.

```
age = 23
if age >= 21:
    print('You may buy beer in the US.')
elif age >= 16:
    print('You may buy beer in Germany.')
else:
    print('You may not buy beer.')
```

Execution order

What is the difference between these three?

`age = 23`	`age = 23`	`age = 23`
`if age >= 21:`	`if age >= 16:`	`if age >= 16:`
`    beer = 'US'`	`    beer = 'GER'`	`    beer = 'GER'`
`elif age >= 16:`	`elif age >= 21:`	`if age >= 21:`
`    beer = 'GER'`	`    beer = 'US'`	`    beer = 'US'`
`else:`	`else:`	`if age >= 0:`
`    beer = 'No'`	`    beer = 'No'`	`    beer = 'No'`

Execution order

What is the difference between these three?

```
age = 23              age = 23              age = 23
if age >= 21:         if age >= 16:         if age >= 16:
    beer = 'US'           beer = 'GER'          beer = 'GER'
elif age >= 16:       elif age >= 21:       if age >= 21:
    beer = 'GER'          beer = 'US'           beer = 'US'
else:                 else:                 if age >= 0:
    beer = 'No'           beer = 'No'           beer = 'No'
```

The evaluation order matters.

1. correct
2. `beer == 'GER'`, US missing
3. `beer == 'No'`, all get evaluated

Rule of thumb: most constraining conditions first!

Control flow

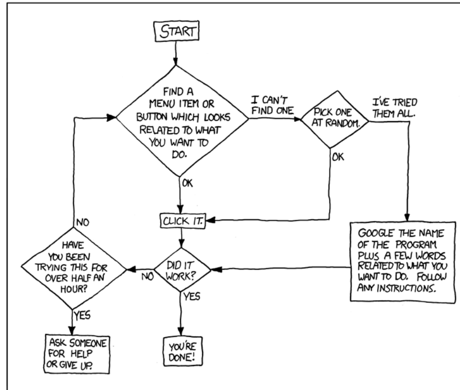

Figure 1: Tech Support Cheat Sheet: 'Hey Megan, it's your father. How do I print out a flowchart?' (Munroe 2009)

Control flow

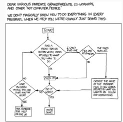

Figure 1: Tech Support Cheat Sheet: 'Hey Megan, it's your father. How do I print out a flowchart?' (Munroe 2009)

When we talk about control flow we talk about how a program works through data step by step.

How to control flow?

- functions
- `if` statements
- loops

Loops

```python
for i in range(10):
    print(i, end=', ')
```

Output:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

```python
i = 0
while i < 10:
    print(i, end=', ')
    i = i + 1
```

Output:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

Loops

```
for i in range(10):
    print(i, end=', ')
```

Output:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

```
i = 0
while i < 10:
    print(i, end=', ')
    i = i + 1
```

Output:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

Python uses `for` and `while`.

They are mostly exchangeable with a bit of work, but in most cases you will only need `for`.

`range(stop)` returns a “list” of integers from `0` to `stop`, but excludes `stop`. For example `range(4)` gives four values: 0, 1, 2, and 3.

While

- **while** a condition is true, do something

```
counter = 1
while counter <= 5:
    print(counter, end=', ')
    counter = counter + 1
```

Output:

```
1, 2, 3, 4, 5,
```

Stopping infinite loops

```
while True:
    print('.', end=' ')
```

You can stop program execution with **Control + C**!

Stopping infinite loops

```python
while True:
    print('.', end=' ')
```

You can stop program execution with **Control** + **C**!

It can very easily happen that you get your conditions wrong or you forget to change the variable in the condition and your code keeps looping until the end of time.

For

- for each element in this iterable, do something

```
for counter in range(6):
    print(counter, end=', ')
```

Output:

```
0, 1, 2, 3, 4, 5,
```

For

- for each element in this iterable, do something

```python
for counter in range(6):
    print(counter, end=', ')
```

Output:

```
0, 1, 2, 3, 4, 5,
```

To loop over some collection of values is called “iteration”.

Thus, collections of values which allow “iterations” are called “iterables”.

For and strings

```python
for item in 'Python':
    print(item, end=', ')
```

For and strings

```python
for item in 'Python':
    print(item, end=', ')
```

Output:

```
P, y, t, h, o, n,
```

Break things...

```
counter = 1
while True:
    if counter > 5:
        break
    print(counter, end=', ')
    counter = counter + 1
```

Output:

```
1, 2, 3, 4, 5,
```

Break things...

```python
counter = 1
while True:
    if counter > 5:
        break
    print(counter, end=', ')
    counter = counter + 1
```

Output:

```
1, 2, 3, 4, 5,
```

Break stops the current loop and jumps to the end.

. . . Break some more. . .

```
for letter in 'Python':
    counter = 0
    while counter < 5:
        counter = counter + 1
        print(letter, end='')
        if letter == 't':
            break
```

Output:

```
PPPPPyyyyythhhhhooooonnnnn
```

True or False?

2017-04-19

└─ . . . Break some more. . .

. . . Break some more. . .

```
for letter in 'Python':
    counter = 0
    while counter < 5:
        counter = counter + 1
        print(letter, end='')
        if letter == 't':
            break
```

Output:

```
PPPPPyyyyythhhhhhoooooonnnnn
```

In this example we only break the inner loop!

. . . Then continue

```
for item in 'Python':
    if item == 'y':
        continue
    print(item, end=', ')
```

Output:

```
P, t, h, o, n,
```

True or False?

└─ . . . Then continue

. . . Then continue

```
for item in 'Python':
    if item == 'y':
        continue
    print(item, end=', ')
```

Output:

```
P, t, h, o, n,
```

Continue skips the remainders of the loop body and jumps back to the top.

If continue is at the end of the loop body, nothing special happens – the loop would “continue” at this point anyway.

Your third homework

- Learn more about the different control flow operations `if`, `for`, and functions by implementing the classic example problems "99 bottles" and "Fizz Buzz".
- Draw some beautiful things with the turtle.

The last slide

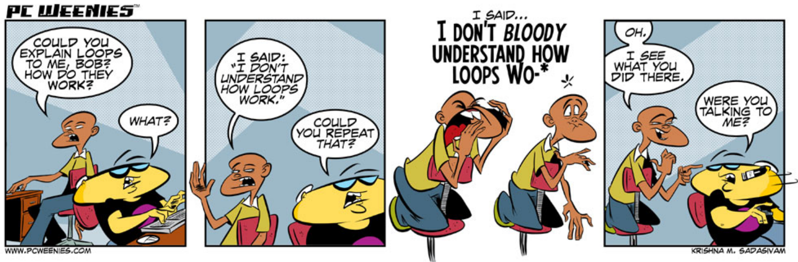

Figure 2: Loopy de loop (Sadasivam 2012)

References

Munroe, Randall. 2009. "Tech Support Cheat Sheet." *Xkcd. A Webcomic of Romance, Sarcasm, Math, and Language.*, no. 627 (August).

Sadasivam, Krishna M. 2012. "Loopy de Loop." *The PC Weenies*, January.