Code Style and Conditional Execution

CSE/IT 107L

NMT Department of Computer Science and Engineering

"Programs must be written for people to read, and only incidentally for machines to execute."

— H. Abelson and G. Sussman (*Structure and Interpretation of Computer Programs*)

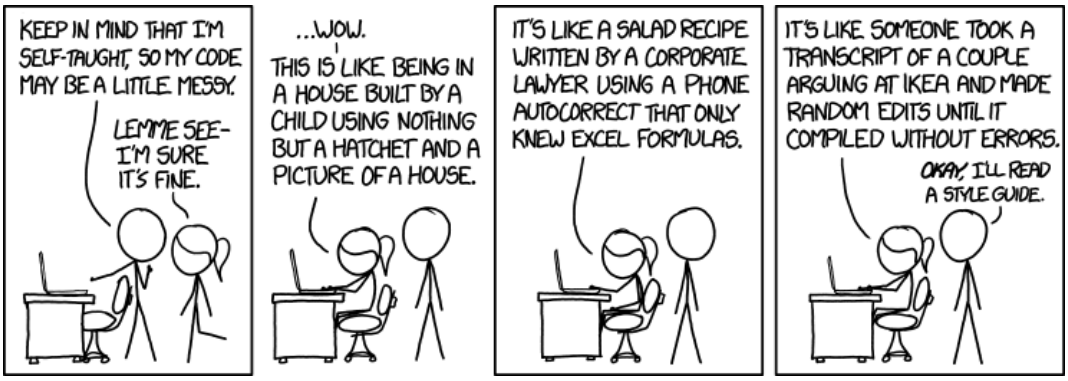


Figure 1: `http://xkcd.com/1513`

Introduction

So far, you have learned several programming techniques, but you are still missing practice with a fundamental element of computation: conditionals.

This document teaches you how to use Python code to make decisions on which parts of the code to run. We can do this using conditional statements, which look like this: `if this: ...do_that...`. You have learned about numeric values and string values, but to use conditional statements, you should know about boolean values: `True` and `False`. You also need to learn boolean statements such as `4 < 5`, or the statement `user_input != 2 or user_input != 3`.

Contents

1 Booleans: `True` or `False`

A common activity when programming is determining whether something is true or false. For example, checking if a variable is less than five or if the user entered the correct password. Any statement that results in a true or a false value is called a boolean statement, the result (true or false) is called a boolean value.

Booleans are a new type of Python value. Like ints and floats, there are many calculations that result in a boolean. These are called boolean statements. However, there are only two boolean values: `True` and `False`.

1.1 Numbers to Booleans with Comparison Operators `==`, `!=`, `<=`, `<`, `>`, `>=`

The boolean operators `==`, `!=`, `<=`, `<`, `>`, `>=` are used for comparing numbers. An example:

```
>>> 12 < 5
False
>>> 12 >= 5
True
>>> x = 5
>>> x < 3
False
>>> print(x < 6)
True
```

The boolean *values* are `True` and `False`. The boolean *statements* are `x < 3`, `x < 6`, `12 >= 5`, and `12 < 5`. You can compare variables or literal values.

Here are more examples:

```
>>> x = 3
>>> y = 6
>>> print(x < y)
True
>>> x > y
False
>>> x <= y
True
```

The operator < means "less than," > means "greater than," <= means "less than or equal to," and >= means "greater than or equal to".

Finally, we can test if two values are equal (==) or not equal (`!=`).

```
>>> x = 3; y = 3; z = 4
>>> print(x == y)
True
>>> print(x == z)
False
>>> print(y != 5)
```

```
True
>>> print(y != x)
False
```

It is important to remember that = is for assigning to a variable and == to test if two values are equal.

1.2 Values to Booleans with Equality Operators

The equality operators == and != can be used to compare any two values. For example, they can compare booleans, strings, numbers, and functions (by name, not behavior).

```
>>> True == False
False
>>> True == True
True
>>> False != False
False
>>> "Hello world!" == "Hello machine!"
False
>>> "Hello world!" != "Hello machine!"
True
>>> 4.5 == 4.50000
True
```

1.3 Booleans to Booleans with `and`, `or`, and `not`

We can combine boolean statements using `and` and `or`:

```
>>> x = 3; y = 5; z = 8
>>> print(x < y and y < z)
True
>>> print(x > y and y < z)
False
>>> print(True and False)
False
>>> print(True or False)
True
```

If you combine two boolean statements that are true using `and`, the result will be true. In all other cases the result is false. Since `x < y` is true and `y < z` is true, we have that `x < y and y < z` is true. In addition to this, there is the `not` operator to negate a boolean statement. You can also put a boolean statement in parentheses to do more complicated combinations:

```
>>> x = 3; y = 5; z = 8
>>> print(not True)
False
```

```
>>> print(not (x > y and y < z))
True
```

A	B	A and B	A or B	not A
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True
Summary:		true if both *a* and *b*	true if both or either *a* or *b*	true if *a* is false

Table 1: A truth table for the boolean operators `and`, `or`, `not`. Because there are only two boolean values, it is possible to list every result of a boolean operators.

Boolean statements can be combined to form more complicated statements, such as:

```
>>> x = 3
>>> x >= 0 and x <= 10
True
>>> x <= 10 and x >= 20
False
>>> y = 10
>>> (x >= 10 and x <= 15) or (x < y and y >= 0)
True
>>> x >= 10 and (x <= 15 or x < y) and y >= 0
False
```

1.4 Possible Mistakes, Python is not English

This code is incorrect or ambiguous:

```
>>> x = 1
>>> x == 4 or 6 or 8
6
>>> 7 == 4 and 8
False
>>> x not == 4
Traceback (most recent call last):
  File ``<stdin>'', line 1
    x not == 4
          ^
SyntaxError: invalid syntax
>>> not 7 == 4 or x == 1
True
```

Maybe this is what was meant:

```
>>> x = 1
>>> x == 4 or x == 6 or x == 8
False
>>> 7 == 4 and 7 == 8
False
>>> not (4 == x)
True
>>> 4 != x # this is more concise
True
>>> not (7 == 4 or x == 1)
False
>>> (not 7 == 4) or x == 1 # or this?
True
```

2 Using Conditional Execution to Make Decisions

In Python, *conditional statements* are the keywords `if` and `elif` followed by a boolean (value or statement), or the lone keyword `else`. This section is about using these keywords.

The primary use for boolean values is to determine which part of your code to follow. This is accomplished using `if` and `elif`, `else`, and code indentation.

2.1 To Run Or Not To Run: `if` Statements

An `if` statement is the keyword `if` followed by a boolean statement, a colon, and any number of lines of Python code indented by 4 spaces. The lines of indented code only run if the boolean statement is `True`. It looks like this:

```
if boolean:
    # if boolean1 is True, run this code
    pass
# in any case, this code runs
```

(The `pass` statement does nothing, it is only in the code sample because statements require indented code and inline comments don't count). Here is an example of using an `if` statement.

```
user_input = input("Guess what language this program was written in: ")

if user_input == "Python":
    print("You answered correctly.")
```

The program reads user input and then uses the boolean operator == to check if the input is equal to "Python". If it is, the code that is indented by 4 spaces below the if-statement runs.

Conditionals can have as many lines of code as needed:

```
user_input = input("Type a negative number: ")
user_input = float(user_input)
if user_input < 0:
    print("The input is less than 0")
    print("Square of the input:")
    print(user_input ** 2)
    print("Computation finished.")
```

In this example, the 4 print statements run only when the user's input is less than 0.

2.2 4 Space Indentation

Here is what happens if you try to indent code when you don't have an `if` or a `for`:

```
# This code is in badindent.py
print("Hello.") # this line is OK
    print("Goodbye.")
```

```
$ python3 badindent.py
  File "badindent.py", line 3
    print("Goodbye.")
    ^
IndentationError: unexpected indent
```

Python raises an `IndentationError` with the message "unexpected indent".

So far, we have seen two features that rely on the code's indentation: conditional statements and for-loops.

For example, in the following code sample the line that prints the square root of the user input runs whether the if conditional runs or not. This means conditional statements are unaffected by all unindented code that comes after.

```
user_input = input("Type a non-negative integer: ")
user_input = int(user_input)
if user_input < 0:
    print("This number is negative. Will use 0 instead.")
    user_input = 0
# All code after is ignored by the conditional.
# This line runs unconditionally:
print(user_input ** 0.5)
```

2.3 Run This Or That: `if-else` Statements

After the indented code, a conditional can also have the keyword `else` followed by a colon and any number of lines of code indented by 4 spaces. This code runs whenever the `if` statement's boolean condition is `False`. This is what this kind of conditional looks like:

```
if boolean1:
    # if boolean1 is True, run this code
    pass
else:
    # run this code if boolean1 is False
    pass
# in any case, this code runs
```

The goal of the following example is to check if the user's input is equal to a secret number. The program should print a message that either congratulates the user for guessing correctly or a message saying they did not guess the secret. No matter what the input is, the program finishes by printing "Thanks for playing!".

```
secret = 6
print("I have loaded a number between 0 and 10."
```

```
user_input = input("Can you guess what it is? ")
user_input = int(user_input)

if user_input == secret:
    print("You guessed correctly!")
else:
    print("You guessed incorrectly :(")
print("Thanks for playing!")  # this line always runs
```

This is how the code works: either the first `print()` statement runs or the second `print()` statement runs, but never both. Which one runs is determined by Python: if the boolean statement (called *condition*) following the `if` evaluates to `True`, then Python will run the indented code following the `if` and then skip until after the indented code of the `else`.

This code checks and prints whether a number is even or odd:

```
x = 5

if x % 2 == 0:
    print("x is even.")
else:
    print("x is odd.")
```

Remember that % is the modulus operator: it gives you the remainder of the division.

```
password = "hunter2"

user_pass = input("Please input the password: ")

if password == user_pass:
    print("Password is correct. Welcome!")
else:
    print("Invalid password.")
```

2.4 Arbitrarily Many Decisions: `if-elif-else` Statements

Any number of `elif` statements can be placed after an if statement. They each resemble another `if`, the components are: the keyword `elif`, a boolean statement, a colon, and indented code. It could be followed by another `elif`, or by an `else`.

```
if boolean1:
    # if boolean1 is True, run this code
    pass
elif boolean2:
    # if boolean1 is False but boolean2 is True, run this code
    pass
elif boolean3:
    # similarly, if boolean2 is False; check boolean3
```

```
    pass
else:
    # run this code if boolean1, boolean2, and boolean3 are False
    pass
# in any case, this code runs
```

In some cases, it could be that there are multiple passwords. Try running the following code:

```
password = "hunter2"
also_password = "hunter3"
another_password = "hunter4"
user_pass = input("Please input the password: ")

if password == user_pass:
    print("Welcome, administrator.")
elif user_pass == also_password:
    print("Welcome, administrator.")
elif user_pass == another_password:
    print("Welcome, manager.")
else:
    print("Wrong password.")
```

In this code, we used the `elif` statement: when the condition following `if` turns out to be false, Python checks the first `elif` statement. If that condition turns out to be true, it runs the code following that `elif` statement or move on to the next `elif`. Only when none of the conditions are true, does the code following `else` run.

We reduce the repetition by using an `or` statement to check for two different passwords:

```
user_pass = input("Please input the password: ")

if user_pass == "hunter2" or user_pass == "hunter3":
    print("Welcome, administrator.")
elif user_pass == "hunter4":
    print("Welcome, manager.")
else:
    print("Wrong password.")
```

Here is an example that compares a number and prints out a message indicating that it's positive, negative, or zero:

```
user_input = input("Please type a number: ")
user_input = int(user_input)
if user_input < 0:
    print("Input is negative.")
elif user_input > 0:
    print("Input is positive.")
else:
    print("Input is zero.")
```

You can insert as many `elif` statements as you want after an `if` statement. The `else` statement is always optional.

Remember the code that follows `if` or `elif` or `else` **must** be indented if is part of the conditional.

3 More Ranges

For-loops have many more features, we will show you one more in this lab. There are various other ways to use the `range` function. You could, for example, write a for loop that starts at 48 and iterates through every other number up to 100.

`range(E)`	numbers from 0 to `E`, not including `E`
`range(B, E)`	numbers from `B` to `E`, not including `E`
`range(B, E, S)`	numbers from `B` to `E`, not including `E`, skipping every `S` number

Do some experiments with range to see what numbers it gives. For reasons that we cannot yet explain to you, you have to write `list(range(...))` to print the contents of the range.

```
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> list(range(1, 5))
[1, 2, 3, 4]
>>> list(range(1, 7, 2))
[1, 3, 5]
```

4 More Math

The math module provides many new mathematical features that you can access by writing the line `import math`. This module contains variables that contain values of the mathematical constants π and e:

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
```

It also contains functions such as the square root (`math.sqrt()`, the logarithm (`math.log()`, `math.log10()`), a power function (`math.pow()`), functions for converting to degrees (`math.degrees()`) and radians (`math.radians()`), the factorial function (`math.factorial()`), a function for finding the greatest common divisor between two numbers (`math.gcd()`), and several others.

```
>>> import math
>>> math.sqrt(16)
4.0
>>> math.log(8, 2) # log2(8)
3.0
```

```
>>> math.pow(2, 3) # 2^3
8.0
>>> math.degrees(math.pi / 2)
90.0
>>> math.radians(90.0)
1.5707963267948966
>>> math.factorial(20)
2432902008176640000
>>> math.gcd(49, 21)
7
```

There are more functions in the math library, you can find them documented here:

`https://docs.python.org/3.0/library/math.html`

5 Reading Function Documentation Using the `help()` Function

Use the `help()` function to read the documentation of any of the Python functions we have used. You can also read the documentation of modules and their functions after you import them. For example:

```
>>> help(print)
Help on built-in function print in module builtins:
<BLANKLINE>
print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
<BLANKLINE>
    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
<BLANKLINE>
```

You may need to press 'q' to exit some help screens.

```
>>> help(input)
Help on built-in function input in module builtins:
<BLANKLINE>
input(prompt=None, /)
    Read a string from standard input.  The trailing newline is stripped.
<BLANKLINE>
    The prompt string, if given, is printed to standard output without a
    trailing newline before reading input.
<BLANKLINE>
    If the user hits EOF (*nix: Ctrl-D, Windows: Ctrl-Z+Return), raise EOFError.
    On *nix systems, readline is used if available.
<BLANKLINE>
```

```
>>> import math
>>> help(math.sqrt)
Help on built-in function sqrt in module math:
<BLANKLINE>
sqrt(...)
    sqrt(x)
<BLANKLINE>
    Return the square root of x.
<BLANKLINE>
```

6 Code Style

As you may have noticed, there are many ways to write the same Python code. Four code samples are listed below that perform the same task: ask for an integer and print 2 to the power of that integer (2^{input}.)

```
user_input = input('Enter an integer: ')
user_input = int(user_input)
result = 2 ** user_input
print(result)
```

```
user_input=input('Enter an integer: ')
user_input=int(user_input)
result=2**user_input
print(result)
```

```
power = int(input('Enter an integer: '))
print(2 ** power)
```

```
USER_INPUT = input('Enter an integer: ')
USER_INPUT = int(USER_INPUT)
RESULT = 2 ** USER_INPUT
print(RESULT)
```

If left to their own devices, most people start conform- ing to their own code style just by preferring a certain way to write something over another. For example, a common parameter of style guides is the use of a certain number of spaces for indentation. Also, some people put spaces before each colon, and some people do not.

This is fine for personal projects, but not so if you know your Python file will be part of a group of code. What if the turtle module used all uppercase names with no underscores and the math module used lowercase names with underscores? This would make both of the module's function names harder to memorize and use. For large projects, group projects, and some individual assignments in CS classes, programmers are required to follow a code style guide. Code style guides dictate the minutiae of your program's appearance.

6.1 Style Guide — Short Version of PEP 8

The Python Enhancement Proposal 8, known as PEP 8, is a style guide that dictates how Python code should look. In this class, we will use a shorter version.

1. Use 4 spaces for indentation.
2. Split long function definitions and calls using hanging indentation.

3. Keep lines shorter than 79 characters.
4. Keep docstrings and comments shorter than 72 characters.
5. Separate function definitions with two blank lines.
6. Imports should be on separate lines.
7. Imports are always put at the top of the file.
8. Always use double quote characters for triple-quoted strings.
9. Avoid whitespace in the following situations:
    - Immediately inside parentheses;
    - immediately before a comma, semicolon, or colon;
    - immediately before the open parenthesis that starts the argument list of a function call;
    - immediately before the open parenthesis that starts an indexing or slicing.
10. Always surround these binary operators with a single space on either side:
    - assignment (=),
    - augmented assignment (+= , -= etc.),
    - comparisons (== , < , > , != , <> , <= , >= , in , not in , is , is not),
    - Boolean operators (and , or , not).
11. Comments should be complete sentences.
12. Block comments should start with '#' and a single space.
13. Modules should have short, all-lowercase names. Underscores can be used in the module name if it improves readability.
14. Function names should be lowercase, with words separated by underscores as necessary to improve readability.
15. Keep variable, function, and method names lowercase.
16. Capitalize names of user-defined classes.

You may also read the entire PEP 8 document at

`https://www.python.org/dev/peps/pep-0008/`

PEP8 Is Required

For this assignment and all future assignments, you are required to follow a subset of the PEP 8 style guide. Read it on page 10.

7 Exercises

Exercise 7.1 (shapes.py).
Write a program that prompts the user for either "circle" or "rectangle" or "square" then reads in either the radius or width and height or the side length of the shape as floating point numbers. If a radius is given, the program should print the shape's circumference and area. For a rectangle, print the perimeter and area. Here are some usage examples:

```
$ python3 shapes.py
Please enter a shape: circle
Please input the radius of the circle:3
The circumference of the circle is
18.84955592153876
The area of the circle is
28.274333882308138
$ python3 shapes.py
Please enter a shape: rectangle
Please enter the width of the rectangle: 10
Please enter the height of the rectangle: 3
The perimeter of the rectangle is
26
The area of the rectangle is
30
$ python3 shapes.py
Please enter a shape: square
Please enter the side length of the square: 10
The perimeter of the square is
40
The area of the square is
100
```

Exercise 7.2 (calculator.py).
Write a small calculator that can compute arcsin, arccos, arctan and square root of a number. Use `math.sqrt()`, `math.asin()`, `math.acos()`, and `math.atan()`. Remember to import `math`.

Make sure to check for each function that the input is valid.

Function	Valid Input Should be
`math.sqrt()`	non-negative
`math.asin()`	between -1 and 1
`math.acos()`	between -1 and 1
`math.atan()`	any number

```
$ python3 calculator.py
Enter a number to use: 16
Which operation? sqrt (s), arcsin (a), arccos (c), arctan (t): s
The square root of the input is
4.0
$ python3 calculator.py
Enter a number to use: 1.1
Which operation? sqrt (s), arcsin (a), arccos (c), arctan (t): a
Input should be between -1 and 1
$ python3 calculator.py
Enter a number to use: 0.5
Which operation? sqrt (s), arcsin (a), arccos (c), arctan (t): a
The arcsine of the input is
0.5235987755982989
$ python3 calculator.py
Enter a number to use: 1000
Which operation? sqrt (s), arcsin (a), arccos (c), arctan (t): t
The arctangent of the input is
1.5697963271282298
```

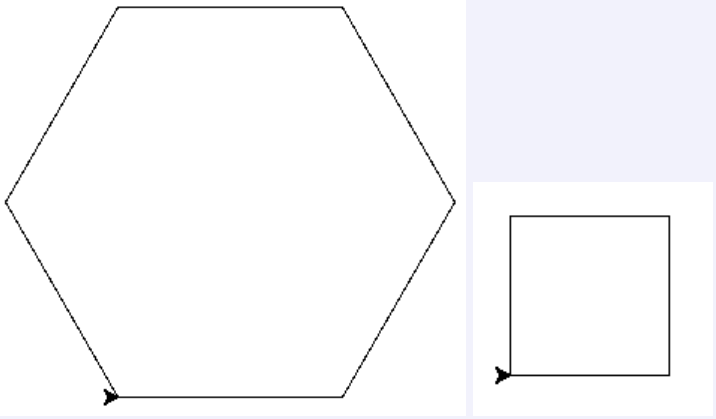
Exercise 7.3 (polygons2.py).
Write a program that takes in an integer. If the integer is less than 3, print a message and exit. Otherwise, draw a shape with as many sides as the user's input. For example, an input of "3 <ENTER> 100" would draw a triangle with sides of length 100. The angle between sides in a shape with n-sides is

$$\frac{360}{\text{number of sides}}$$

Here is a sample:

```
$ python3 polygons2.py
How many sides? 0
Invalid input.
$ python3 polygons2.py
Please enter the number of sides: 6
Please enter the side-length: 150
$ python3 polygons2.py
Please enter the number of sides: 4
Please enter the side-length: 100
```

The last two commands with inputs 6<ENTER>150 and 4<ENTER>100 should draw these two images:



Index of New Functions and Methods

Submitting

You should submit your code as a tarball that contains all the exercise files for this lab. The submitted file should be named

`cse107_firstname_lastname_lab2.tar.gz`

Upload your tarball to Canvas.

List of Files to Submit

Exercises start on page 12.