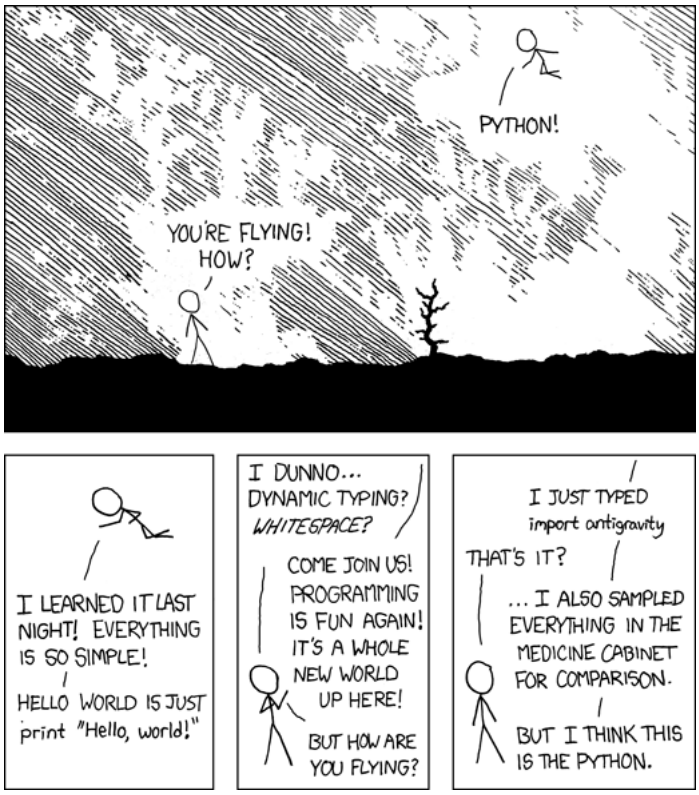
CS51 - A quick guide to Python syntax



https://xkcd.com/353/

1 Background

This is a quick guide on Python syntax to get you started with CSCI051. It is by no means exhaustive. If you are concurrently enrolled in CSCI050, do not worry, you will cover the same topics gradually but this document should still be a good reference.

2 Comments

Lines of code are commented out using the **#** (pound or hashtag sign). Use comments to document your code. Python will ignore them, but they are invaluable both for you and any reader of your

code.

3 Data types

Python supports multiple data types. For example:

- Integers: that is, whole numbers, such as `47` or `-23`.
- Floats: that is, decimal point numbers, such as `47.0` or -1.99.
- Strings: that is, sequences of characters, such as `'welcome to cs51!'`. Strings are enclosed in single quotes. They might also be enclosed in double quotes, e.g., `"Cecil"` but the Python convention is to use single quotes.
- Booleans: that is, `True` or `False` values.
- Lists: Indexable collections, e.g., `['cs50 ', 'cs51', 'cs54', 'cs62']`. More on lists later in the course.

4 Operators

Python supports the following operators:

- Arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, `**`, for addition, subtraction, multiplication, division, integer division, modulo, and exponentiation, respectively.
- Comparison operators: `==`, `!=`, `>`, `<`, `>=`, `<=`, for comparing equality, inequality, larger, smaller, larger or equal, or smaller or equal, respectively.
- Logical Operators: `and`, `or`, `not` for conjunction, disjunction, negation, respectively.
- Assignment operators: `=`, `+=`, `-=`, `*=`, `/=`, for assigning a value to a variable, or updating the existing value of a variable by increasing, decreasing, multiplying, or diving it, respectively.

  For example, `name = 'Cecil'` initializes the variable `name` with the value `'Cecil'`. Please note that in contrast with other languages, the type of the variable is implied to be string, and no special markings, such as `;`, are added at the end of the line.

  If you had a variable `age = 18`, that is an integer, the syntax `age += 1` would update the value of the variable `age` from 18 to 19.
- Membership Operators: `in`, `not in` for checking whether something belongs in a group, e.g., `'ec' in 'Cecil'` would return `True`, and `2 not in [1, 2, 3]` would return `False`

5 Identifiers

When naming variables (or later on, as we will see, functions), pick meaningful names, i.e., identifiers. For example, in the expression `x = 3`, `x` is not a great name. Instead, try to use variable names that communicate what the variable stands for, e.g., `number_of_classes = 3`. Note that when your identifier consists of multiple words, the convention is to use an underscore between those words, i.e., _.

6 Control Flow

Here is an example of how you would write an if-else statement in Python that prints different designations on the console based on one's age:

```
if age >=0 and age < 1:
    print('infant')
elif age >=1 and age < 6:
    print('toddler')
elif age >=6 and age < 14:
    print('child')
elif age >=14 and age < 18:
    print('teen')
else:
    print('adult')
```

A few important considerations:

- Indentation in Python matters! At the end of each of the `if`, `elif`, or `else` lines, you will notice that there is a colon (i.e., `:`). The next line needs to be indented. The official documentation says not to use tabs to indent code and instead add four spaces.
- The `elif` and `else` parts are optional. You could have a plain if statement (if you want to act only in one case), a statement that only has an if and else (if you want two possible options), or something like the code above with a lot more possible pathways.
- The ordering of the `if`, `elif`, and `else` parts matters! The first time that one of them gets satisfied, then the rest are ignored. For example, if the age were 6, the first `if` and `else` statements would be checked and not be satisfied. The third would be true, so the word `'child'` would be printed, and then the code would exit this if statement, not checking the cases for teens and adults.
- Don't confuse `if` with `elif` statements. Two independent `if` statements in a row are not the same with one `if` statement followed by an `elif` statement. For example:

```
n = 15
```

```
if n % 3 == 0:
    print(‘a’)
if n % 5 == 0:
    print(‘b’)

if n % 3 == 0:
    print(‘c’)
elif n % 5 == 0:
    print(‘d’)
```

The execution of the first two `if` statements is independent of each other, so both `‘a’` and `‘b’` would be printed. But the next would result only in `‘c’` being printed. This is because, as we saw, the first time that a case in an if-elif-else structure gets satisfied, the rest are skipped.

7 Loops

7.1 `for` loops

There are largely two types of loops in Python. The first one type is `for` loops.

With `for` loops, you can iterate through sequences with a well-defined length. For example:

```
for i in range(3):
    print(i)
```

would print the numbers 0, 1, 2. Note that the function `range` starts at 0 and is exclusive of the argument, that is, it counts up to but excludes 3.

You can also iterate through lists and strings. For example:

```
cs_courses = [‘cs50’, ‘cs51’, ‘cs54’, ‘cs62’]

for cs_course in cs_courses:
    print(cs_course)
```

would print one by one the courses in the `cs_courses` list.

7.2 `while` loops

The second type of loop that Python supports is `while` loops, which can be handy when we know that we want to repeat an action as long as a condition holds true, but we don’t know in advance how long that will take. You can also write a `for` loop as a `while loop`. For example, instead of a `for` loop enumerating the first three non-negative integers, we could have a while loop:

```
i = 0
while i<3:
    print(i)
    i += 1
```

8 Functions

We like writing modular and reusable code because it avoids unnecessary copying-pasting and mistakes (bugs), and makes testing and reading of our code easier. We do so by **declaring** functions that we can **invoke** whenever we need them in other places in our code.

Here is a simple function in Python:

```
def greet(name, age):
    print('Hello, ' + name + '! I see you are ' + str(age) + 'old.')
```

The name of this function is `greet`. It takes two **parameters**, `name` and `age`, and prints a greeting message to the console. Note that we declare a function by writing `def` followed by its identifier, the parameters in parentheses, and a colon after the right parenthesis. The body of the function is indented.

If we invoke this function, we should pass two **arguments** that will be associated with the two parameters. For example, we could invoke it by writing `greet('Cecil', 47)`, which would print `'Hello, Cecil!  I see you are 47 years old.'`. What happened within the `print` function is that multiple strings were **concatenated** into one new string. Note that the parameter `age` was associated with an integer number, `47`. In Python, we cannot concatenate a number with a string; we need to convert it to its string representation by calling the `str` function. In our case, that was done by calling `str(age)`

We can keep invoking the same function by passing different arguments, e.g., `greet('world', 4543000000)`, which would print 'Hello, world! I see you are 4543000000 years old.'

When we declare a function, we need to decide how many parameters it expects and give them good names. We could declare a function without any parameters, but we would still need the parentheses and colon after the identifier, e.g.,

```
def good_morning():
    print('Good morning!')
```

If we want to invoke this function, we would write `good_morning()` and, in turn, that would print `'Good morning!'`.

More often than not, we want our functions to do some work and give back to us the results of that work so we can act on them. We can use that by adding a `return` statement as the **last** line in our function body.

For example, we could write a simple function that adds two numbers:

```
def add(number1, number2):
    return number1 + number2
```

If we invoke the `add` function as `add(2, 3)`, it would return to us the number 5. Note that it has not printed 5 in the console. It is up to us to choose what we want to do with the result we got back. For example, we could have written `sum = add(2,3)` and after `add` completed its work, `sum` would now contain the number 5. We can now use `sum` in the rest of our code.

Or we could have directly printed the result to the console, by passing the result of the `add` function into a `print` function, e.g., `print('The sum is ' + str(add(2,3))`, which would print `'The sum is 5'`. Note again how we concatenated two strings, with the second one needing to be converted from an integer. This time, `add` did the work of summing up the two numbers, we printed its result, but now the result is gone because we didn't 'save' it in any variable.

It is very important to understand that the `return` statement has to be the last line in the body of a declared function. Any code after that would be inaccessible because Python halts the execution of the function and returns to whomever invoked it.

9 Style

In terms of boolean expressions, you should **NOT** have anything like:

```
if boolean_expression == True:
```

or

```
if boolean_expression == False:
```

Instead, use:

```
if boolean_expression:
```

or

```
if not (boolean_expression): # or some other way of negating the expression
```

If `function(x, y)` returns a boolean value, you should NOT have anything like:

```
if function(x, y):
    return True
else:
    return False
```

instead use:

```
return function(x, y)
```

Sometimes people use nested if statements to compute a result that is based on one or more conditions. But this can often be done more simply and clearly with a Boolean expression.

For example:

```
if (class_open and num_enrolled <= capacity):
    return True
else:
    return False
```

could be much more simply written as

```
return class_open and num_enrolled <= capacity
```