Python Quick Reference v1.0

"Life is short; learn Python." ~ Leonhard Euler

Josh Parnell // josh@ltheory.com

BASICS

Comments

In Python, a pound sign (#) at the beginning of a line denotes a comment. You can use comments to add notes to your code. This is an especially good idea if you're doing something complicated! I'll use comments throughout this guide and other handouts to add notes to example code.

Operators

In most cases, you can use standard arithmetic operators like `+, -, *, /` and get what you expect. Exponentiation is a bit strange, and uses two asterisks, NOT a caret: `x**y` is x raised to the yth. Modulus is the percent sign: `x % y` is x mod y. When doing complicated computations, it's a good idea to use parentheses to make your code clearer.

ALTHOUGH EXPONENTIATION IS OFTEN DENOTED IN MATH USING A CARET (x^y), THE CARET USUALLY MEANS SOMETHING VERY DIFFERENT IN PROGRAMMING!
TAKE CARE TO USE ** , WHICH IS THE EXPONENTIATION OPERATOR IN PYTHON!

Math Library

Since we'll be doing so much math in Python, I will often use functions and constants from the math library, such as sqrt, exp, pi, etc. To use these in your own code, you must add the following line to the top:

```
from math import *
```

If you're working in the live interpreter (python.exe), you'll need to type this line once each time you start python.

EXAMPLES

```
# In example code, I may sometimes use the notation [statement] -> [result]
# To indicate what the result of a statement would be.
# The arrow and everything after it are not actually part of the code.
2 + 2 -> 4
3.14 * 2 + 1 -> 7.28
1 + 2**4 -> 17

# Notice that, without parens, ** will happen BEFORE other operators!
(5 * 5) ** 2 -> 625
5 * 5 ** 2 -> 125

# Even if the second line is what you intended, it's best to clarify with parens:
5 * (5 ** 2) -> 125

2 % 2 -> 0
3 % 2 -> 1
7.5 % 5 -> 2.5
```

FUNCTIONS

SYNTAX

```
function(argument1, argument2, ...)
functionWithNoArguments()
```

Function call syntax is fairly self-explanatory...it's basically the same as in math. Note that, unlike in math, we sometimes have functions that take no arguments. To call such a function, we simply put nothing in the parens. Also note that function calls can be nested as much as you like: `outerFunction(inner1(1, 5), inner2(9, inner1(4, 8)))`

EXAMPLES

```
print("Hello World")
abs(-6.283) -> 6.283
sum([abs(-pi), pi]) -> 6.283185307179586
```

VARIABLES

SYNTAX

```
x = 5
y = 7
z = x + y
print(z)
```

Variables are how we save and load things within a program. They're also a convenient way to label certain quantities so that it's easy to read the code. Choosing good variable names is actually an important part of programming — mostly so you don't confuse yourself or others reading your code! Good variables names are **short but descriptive**. For example:

BAD:

```
nums = [2, 3, 5, 7, 13, 17]
x = [2, 3, 5, 7, 13, 17]
pms = [2, 3, 5, 7, 13, 17]
listOfNumbersThatArePrime = [2, 3, 5, 7, 13, 17]
```

GOOD:

```
primes = [2, 3, 5, 7, 13, 17]
primeList = [2, 3, 5, 7, 13, 17]
```

'Strange' characters like `~, !, [, $, %` etc. can't be used in variable names. Usually a name should just be letters, sometimes with numbers mixed in (note that a variable name cannot begin with a number). Variable names cannot contain spaces, but some programmers use underscores (_) where a space would be, for example, `my_really_cool_variable`. Other programmers use capitalization where spaces would usually be, for example, `myReallyCoolVariable`. Which one is better has been debated by programmers with nothing better to do for decades. Personally I prefer using capitalization since underscores are ugly. `theChoiceIsYours = True`

All operators have an 'in place' version that ends with = and can be used instead of `var = var [operator] otherValue`. For example, to add 3 to the current value of x, instead of `x = x + 3` you can simply write `x += 3`.

> **DON'T GIVE YOUR VARIABLES NAMES THAT ALREADY HAVE A MEANING IN PYTHON (E.G. SUM, SQRT, FOR, DEF, PI, ETC.)!**

EXAMPLES

```
dontYouThink = "variables seem pretty easy"
myName = "Josh"
myFavoriteNumber = 17
why17 = ["It's prime", "It's a twin prime", "It has a 7.", "Just because."]
underscores_are_ugly = True

# Note that we can pass as many values to the print function as we like
# It will print them all, with a space in-between each one
print("My name is", myName, "and my favorite number is", myFavoriteNumber)

# Solve a quadratic system (will not work if det is negative!)
a = 3
b = -9
```

```
c = -27
det = b*b - 4*a*c

print((-b + sqrt(det)) / (2*a))
print((-b - sqrt(det)) / (2*a))
```

DEFINING NEW FUNCTIONS

SYNTAX

```
def myFunction(parameter1Name, parameter2Name, ...):
    # Do amazing things here
    return myResult
```

Certain pieces of code are so useful that we'll need to use them frequently, just as we'll use the built-in functions frequently. For such pieces of code, it makes sense for us to create our own, new functions. Creating functions allows us to 'package up' code so that we can invoke it over and over again when we need it, without rewriting it. This principle of **reusing code** is one of the most important principles in programming.

When we define a new function, we must define the function's parameters — the x, y, and z in `f(x, y, z)`. Defining parameters allows us to hand different values to the function, which it can then use as part of its computation. We also get to define what the function returns, i.e., the value that `y` will be given in the statement `y = myFunction(0)`

Return statements are optional, but for our purposes we will almost always return something. Usually we will write functions that take in some values as parameters, use them to compute some result, and then return that result.

DON'T FORGET THE COLON (:) AFTER THE PARAMETER LIST — IT'S EASY TO MISS!

IN PYTHON, ALL CODE WITHIN THE FUNCTION MUST BE INDENTED, WITH THE EXCEPTION OF SINGLE-LINE FUNCTIONS, WHICH MAY BE PLACED DIRECTLY AFTER THE COLON.

Examples

```
# If the function is a single line, we may place it directly after the :
# However, it's not usually a good idea to do this.
# It can make your code more difficult to read!
def determinant(a, b, c): return b*b - 4*a*c

# Compute a 6-term Taylor approximation, expanded about x = 0, for e^x
def expMaclaurin(x):
    y = 1 + x
    y += x**2 / factorial(2)
    y += x**3 / factorial(3)
    y += x**4 / factorial(4)
    y += x**5 / factorial(5)
    return y

def weirdFn():
    print("Hi!")
    print("I am a weird function.")
    print("I take no parameters, and return no value.")
```

FOR LOOPS

SYNTAX

```
for nameOfItem in sequenceOfItems:
    # Do something with nameOfItem
```

We will often need to run the same piece of code many times on different pieces of data. For example, we may want to compute the value of some `f(x)` for integer values of `x` from 0 to 99. The 'painful' way to do this would look like:

```
print(f(0))
print(f(1))
print(f(2))
print(f(3))
```

```
...
print(f(99))
```

But of course, there's a much better way! Here's a two-line piece of code that does the same thing as the 100-line version above:

```
for x in range(0, 100):
    print(f(x))
```

The 'for' loop has three pieces to it:

1. A loop variable (x in the example)
2. A sequence / 'list' of elements (`range(0, 100)` in the example)
3. Code that defines the loop's operation (`print(f(x))` in the example)

The way the loop works is simple: for each element in the sequence (2), the variable (1) will be set equal to said element, and then the code (3) will be executed. This means that the code can use the loop variable to access each element of the sequence, one at a time, and do something with them.

We'll often be looping over a range of numbers. To do this, we use the `range` function, which returns a sequence of numbers. Look at the `range` entry in the function appendix for details on how it works. We can also loop over the items in a list. In fact, there are many more things over which we can loop, but for now ranges and lists will be our primary concern.

> **JUST LIKE WITH FUNCTIONS, LOOPS REQUIRE A COLON AFTER THEM, AND THE CODE INSIDE MUST BE INDENTED (UNLESS THE LOOP CODE IS A SINGLE LINE, IN WHICH CASE IT MAY COME DIRECTLY AFTER THE COLON)**

EXAMPLES

```
for name in ["Josh", "Brooke", "Steven"]:
    print("Hello " + name + ", I'm very excited to learn to code!")

# Compute the sum of the harmonic series with the given number of terms
def harmonicSum(terms):
    y = 0
    for i in range(terms):
        y += 1 / (i + 1)
    return y

# Note that this is a very slow and lazy version of isPrime (and incorrect for 0, 1)
# We will build much better versions in the future!
# Also note the use of an if statement, which we'll discuss more in the future
def isPrime(x):
    for i in range(2, x):
        if x % i == 0:
            return False
    return True
```

APPENDIX: USEFUL PYTHON FUNCTIONS

Below I've detailed just a small subset of the functions Python offers. These are some of the most common and useful functions; we'll be using them frequently. If you want to see all of what the math library contains, have a look at https://docs.python.org/3.4/library/math.html . If you're even more ambitious and would like to see the beast that is the Python standard library, take a glance at https://docs.python.org/3.4/library/index.html ...quite a lot of toys for us to play with!

> **THERE ARE OFTEN SEVERAL WAYS TO WRITE THE SAME COMPUTATION, E.G. `sqrt(2) == 2**0.5 == pow(2, 0.5)`. IT'S USUALLY BEST TO USE THE MOST SPECIFIC WAY, AS SPECIFIC FUNCTIONS LIKE SQRT, EXP, ETC. ARE OFTEN OPTIMIZED TO BE FASTER THAN MORE GENERAL FUNCTIONS.**

abs(x)

```
# Returns the absolute value of an integer or float
abs(-2.7) -> 2.7
abs(10000) -> 10000
```

exp(x)

```
# Returns e^x; equivalent to pow(e, x) or e**x
exp(1) -> 2.718281828459045
exp(0) -> 1.0

# Note how Python will use scientific notation if numbers become very small or large
exp(-100.0) -> 3.720075976020836e-44
```

factorial(x)

```
# Returns x!
# Remember that the standard factorial function is only defined for non-zero integers
factorial(5) -> 120
factorial(0) -> 1
factorial(-1) -> ERROR: Undefined for x < 0
factorial(10.5) -> ERROR: Undefined for non-integers
```

len(sequence)

```
# Returns the length of the given sequence
len([True, False, True, True]) -> 4
len([]) -> 0
len("Gasp! A string is a sequence!?") -> 30
```

log(x), log(x, b)

```
# Returns the natural logarithm of x
# If the second argument is present, returns the log base b of x
log(e**3) -> 3.0
log(0.01) -> -4.605170185988091
log(512, 2) -> 9.0
log(0) -> ERROR: Undefined for x <= 0
```

max(sequence)

```
# Returns the maximum element of the given non-empty sequence
max([1, 2, 3, 2, 1, 0, -1]) -> 3
max(range(100)) -> 99

# Max is defined for a surprising number of types...we'll discuss this more later
max(["Well", "this", "is", "bizarre..."]) -> 'this'

# This function will also accept any number of values without using the list brackets:
max(1, 3) -> 3
max(sqrt(9), 8, 7 * 2, exp(5) - exp(6), 5**2) -> 25
```

min(sequence)

```
# Same as max(sequence), but returns the minimal element. See above.
min([1, 2, 3, 2, 1, 0, -1]) -> -1
min(range(100)) -> 0
```

pow(x, y)

```
# Returns x raised to the yth power; equivalent to (x**y)
# Note: don't use negative x and fractional y!
pow(5, 2) -> 25
pow(100, 0.5) -> 10.0
pow(pi, 2) -> 9.869604401089358
```

range(x), range(lower, upper)

```
# Returns a sequence of integers going from 0 to x - 1
# If the second argument is present, returns a sequence of integers going from lower to upper - 1
for n in range(101): print(n) -> 0 1 2 3 4 ... 101
for n in range(1, 11): print(n*n) -> 1 4 9 16 ... 100
```

NOTICE THAT NEITHER `range(x)` NOR `range(i, x)` ACTUALLY INCLUDE THE VALUE x!

reversed(sequence)

```
# Returns a new version of sequence in the reverse order
reversed([2, 3, 5, 7, 11]) -> [11, 7, 5, 3, 2]
```

round(x), round(x, digitCount)

```
# Returns x rounded to the nearest integer
# If the second argument is present, returns x rounded to digitCount digits
```

```
round(pi) -> 3
round(pi, 2) -> 3.14
round(-e, 1) -> -2.7
```

sin(x), cos(x), tan(x)

```
# Returns the sine, cosine, tangent of x, respectively
sin(pi / 2) -> 1.0
cos(pi / 4) -> 0.7071067811865476
sin(0.1)**2 + cos(0.1)**2 -> 1.0
tan(pi / 4) -> 1.0

# Since computers don't compute results with 100% accuracy, some results may be surprising!
# Should be infinity, but instead we get a very large number
tan(pi / 2) -> 1.633123935319537e+16
# Should be zero, but instead we get a very small number
cos(pi / 2) -> 6.123233995736766e-17
```

sorted(sequence)

```
# Returns a new version of sequence, with elements sorted from low to high
sorted([10, 5, 7, 1, 11]) -> [1, 5, 7, 10, 11]
sorted(["Ramanujan", "Gauss", "Cauchy", "Fermat"]) -> ['Cauchy', 'Fermat', 'Gauss', 'Ramanujan']
```

sqrt(x)

```
# Returns the square root of x; equivalent to pow(x, 0.5) or x**0.5
sqrt(2.0) -> 1.4142135623730951
sqrt(4.0) -> 2.0
sqrt(-1.0) -> ERROR: Undefined for x < 0
```

sum(sequence)

```
# Returns the sum of all elements in sequence
# We will usually use this function with lists, but it can be used with other sequences like range
sum([1, 99, 17]) -> 117
sum([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]) -> 55

# Sum of integers from 0 to 10 and then from 0 to 100
sum(range(0, 11)) -> 55
sum(range(101)) -> 5050
```

type(x)

```
# Returns the type of data in x
type([1, 2, 3]) -> <class 'list'>
type(1) -> <class 'int'>
type(1.0) -> <class 'float'>
type(True) -> <class 'bool'>
type("MathCircle") -> <class 'str'>
```