Introductory Python 1

Values, Variables, and Control Flow

The basic constituents of a language are *values* and *variables*.

Python allows for different *types* of values, starting with the basic types:

- Integers, such as 1, 2, 3
- Floating point, such as 1.0, 2.53
- String, such as 'wassup' or "this is a longer string"

Note that you can use either single quotes or double quotes for a string.

In [1]:
```
"this is a string"
```

Out[1]:
```
'this is a string'
```

In [2]:
```
# A string with quotes inside it
"What's up?"
```

Out[2]:
```
"What's up?"
```

You can check that double-quotes versus single-quotes makes no difference.

In [3]:
```
"string quotes" == 'string quotes'
```

Out[3]:
```
True
```

The string "1" and the integer 1 are **not** the same

In [4]:
```
1 == "1"
```

Out[4]:
```
False
```

The "==" (double equal-to sign) we used is an *operator*. This particular operator checks for equality of the left-hand side and the right-hand side. The comparison is False in this case because an integer cannot equal a string.

You can check the types of these values using the *type* function.

In [5]:
```
type('1')
```

Out[5]:
```
str
```

In [6]:
```
type(1)
```

Out[6]:
```
int
```

There are many other operators. Let's see a few.

In [7]:
```
# addition
print(2+2)
```

```
4
```

In [8]:
```
2 + 2 != 4  # check for inequality
```

Out[8]:
```
False
```

In [9]:
```
2 < 3
```

Out[9]:
```
True
```

In [10]:
```
2 < 3 and 3 < 4  # Boolean operators include and, or, not
```

Out[10]:
```
True
```

In [11]:
```
(2 + 2 > 4) or (2 < 3)  # Use parentheses to fix the order of operatior
```

Out[11]:
```
True
```

In [12]:
```
print('Division converts everything to floating point! 4/2=', 4/2)
```

```
Division converts everything to floating point! 4/2= 2.0
```

Another strangeness...

In [13]:
```
#print(02313)

# SyntaxError: invalid token for 02313
```

What happened? Python doesn't expect you to start integers with 0, so it thinks there is an error.

How do we avoid it? Save the zipcode as a string, and use the int() function to convert it to an integer.

In [14]:
```
print('zipcode is ', '02313')
print('Extract to integer:', int('02313'))
```

```
zipcode is  02313
Extract to integer: 2313
```

int() is a conversion function; it converts from some other type to integer.

In [15]:
```
print(int(1.2))        # Convert from floating point (rounds the floati
print(int('12321'))    # Convert from string
```

```
1
12321
```

float() converts to floating point.

In [16]:
```
print(float(1))       # Convert from integer
print(float('1.23'))  # Convert from string
```

```
1.0
1.23
```

str() converts to string.

In [17]:
```
print(str(1))      # Convert from integer
print(str(1.54))   # Convert from floating point
```

```
1
1.54
```

In [18]:
```
type(str(1.54))
```

Out[18]:
```
str
```

 conversions

Variables

Variables hold information as values.

In [19]:
```
# variable assignment
x = 1.34
greeting = 'The value of variable x is'
print(greeting, x)
```

```
The value of variable x is 1.34
```

In [20]:
```
print('The type of x is', type(x))
print('The type of greeting is', type(greeting))
```

```
The type of x is <class 'float'>
The type of greeting is <class 'str'>
```

Remember: variables are case-sensitive!

In [21]:
```
x = 1.2
```

```
X = 3.4
print('x=', x, 'and X=', X, 'are different')
```

```
x= 1.2 and X= 3.4 are different
```

The values in variables can change (after, they are called "variable"). For instance:

In [22]:
```
print('Current value of x=', x, 'of type', type(x))
x = 'ABC'
print('Changed value of x=', x, 'of type', type(x))
```

```
Current value of x= 1.2 of type <class 'float'>
Changed value of x= ABC of type <class 'str'>
```

More interesting: do operations using variables.

In [23]:
```
year = 2015
price = 1000.0
print('In year', year, 'price =', price)

# The clock ticks
year = year + 1
price = price * 1.03  # inflation of 3%
print('In year', year, 'price =', price)
```

```
In year 2015 price = 1000.0
In year 2016 price = 1030.0
```

Doing simple operations and assigning to the same variable is so common, there's a shortcut.

In [24]:
```
# The clock ticks again
year += 1        # Shortcut for year = year + 1
price *= 1.03    # Shortcut for price = price * 1.03
print('In year', year, 'price =', price)
```

```
In year 2017 price = 1060.9
```

Getting user input

The quintessential variable is user input.

To get input, use the **input()** function

In [25]:
```
x = input('Please enter your name: ')
print('Hey', x, '!')
```

```
Please enter your name: deepayan
Hey deepayan !
```

- **input(string)** prints the string and waits for the user's input.
- This input is returned in the variable x
- What is the type of x?

In [26]:
```
type(x)
```

Out[26]:
```
str
```

Suppose I want to get the inflation estimate from the user.

How do I get it?

In [27]:
```
inflation = float(input('Please enter your inflation estimate (in perce
print('Inflation is', inflation, 'percent')
```

```
Please enter your inflation estimate (in percentage): -3
Inflation is -3.0 percent
```

In [28]:
```
print('inflation is a variable of type', type(inflation))
```

```
inflation is a variable of type <class 'float'>
```

One other thing to notice: we strung two functions together.

```
float(input('question to ask the user'))
```

Python executes this from the **inside-out**:

- First, do raw_input()
  - This returns a string

- Then, do float() on this string
  - This returns a floating-point number

Statements and Expressions

It's useful to differentiate between *statements* and *expressions*.

- An **expression** has a value; you can print it.
- A **statement**, on the other hand, does some work, but doesn't necessarily have a value.

In [29]:
```
x = 5.0  # statement
x * 3    # expression
print("value of the expression x * 3 is", x * 3)
```

```
value of the expression x * 3 is 15.0
```

String expressions

The usual operators '+' and '*' have different meanings for strings. '+' performs **concatenation**, i.e., it joins strings together end-to-end.

In [30]:
```
'abc' + 'def' # string concatenation using '+'
```

Out[30]:
```
'abcdef'
```

In [31]:
```
first_string = 'lady'
second_string = 'bird'
print(first_string + second_string)
```

```
ladybird
```

The '*' performs **repetition**, i.e., it repeats the same string several times over. Think of the analogy to numbers:

- 4 * 3 is the same as 4 + 4 + 4
- Similarly, 'abc' * 3 acts like 'abc' + 'abc' + 'abc'.

In [32]:
```
'abc' * 3
```

Out[32]:
```
'abcabcabc'
```

Control Flow

There are several standard ways of placing conditions on control flow:

- if-then-else
- for loops
- while loops

Let us see some examples of these.

If-then-else: The if-then-else paradigm goes as follows:

```
if some-condition:
    statements if the condition is true
else:
    statements if the condition is false
```

In [33]:
```
temperature = 30
if temperature <= 32:
    print('Freezing!')
else:
    print('Beach weather, dude.')
```

```
Freezing!
```

Two points are critical:

- At the end of each condition, you put the colon sign (the **:** sign)

- The statements are indented from the if/elif/else.

A more general version include the "elif" keyword, which is short for "else if".

```
if condition-1:
    statements if condition-1 holds
elif condition-2:
    statements if condition-2 holds but NOT condition-1
elif condition-3:
    statements if condition-3 holds but NEITHER condition-1 NOR
condition-2 hold
else:
    statements if none of the above conditions holds
```

Let's introduce the **len()** function:

- Given a string, it returns the length of the string
  - len('Advanced') is 8

In [34]:
```
name = 'Deepayan'
if len(name) <= 4:
    print("Short 'n sweet")
elif len(name) <= 10:
    print("Doable")
else:
    print("Tongue-twister")
```

```
Doable
```

for loops: For loops allow statements to be run over a list of items.

In [36]:
```
# Measure lengths of strings
words = ['cat', 'window', 'deforestation']  # This is the syntax for a
for w in words:
    print(w, len(w))   # len(w) is the length of string w
```

```
cat 3
window 6
deforestation 13
```

The statements within the for loop (here, just the print statement) are run with the variable w set to each list item in turn. This is the

same as if we'd written:

```
w = 'cat'
print w, len(w)
w = 'window'
print w, len(w)
w = 'deforestation'
print w, len(w)
```

Once again, note that

- the **for** statement ends with a colon (:), and
- the statements within the for loop are indented.

One particularly common use of a for loop is to run some statements a given number of times, say 5 times.

In [37]:
```
# Using the range() function in a for loop
for i in range(5):
    print("Iteration", i)
```

```
Iteration 0
Iteration 1
Iteration 2
Iteration 3
Iteration 4
```

Example: Inflation adjustment over the next 10 years.

In [38]:
```
inflation = 3.0
year = 2015
price = 1000.0
for iteration_number in range(10):
    print('In year', year, 'price will be', price)
    year += 1
    price *= 1.0 + inflation / 100.0
```

```
In year 2015 price will be 1000.0
In year 2016 price will be 1030.0
In year 2017 price will be 1060.9
In year 2018 price will be 1092.727
In year 2019 price will be 1125.50881
In year 2020 price will be 1159.2740743000002
```

```
In year 2021 price will be 1194.0522965290002
In year 2022 price will be 1229.8738654248702
In year 2023 price will be 1266.7700813876163
In year 2024 price will be 1304.7731838292448
```

while loop: This is useful when we don't know the number of iterations beforehand.

Example: Snail climbs a well

- The well is 10 feet deep
- The snail climbs 4 ft in each turn, but then slips a bit
- The slip is 1 ft in the 1st turn, 1.5 ft in the 2nd turn, 2 ft in the 3rd turn...
- Does the snail escape?

How do we code this up?

```
while (snail has not yet escaped) and (snail climbs more than it
slips):
   update the position of the snail
   update the slip rate
```

In [39]:

```
depth = 10.0              # depth of the well
climb_rate = 4.0          # climbing rate of snail
slip = 1.0                # initial slip rate
slip_increase_rate = 0.5

turn = 1
depth -= climb_rate               # snail starts climbing
while depth > 0 and slip < climb_rate:
    depth += slip                 # snail slips
    slip += slip_increase_rate    # the slip rate increases
    print('Snail is', depth, 'feet deep after turn', turn)
    turn += 1
    depth -= climb_rate

if depth <= 0:
    print('Snail escaped the well in', turn, 'turns!')
else:
    print('No escape!')
```

```
Snail is 7.0 feet deep after turn 1
```

```
Snail is 4.5 feet deep after turn 2
Snail is 2.5 feet deep after turn 3
Snail escaped the well in 4 turns!
```

Here's what happens in a while loop:

- Evaluate the conditional statement (n > 0), yielding True or False
- If the condition is False:
  - exit the while loop and skip to any statement after the while loop
- If the condition is True:
  - execute the statements in the body of the while loop, and
  - repeat the loop.

Functions

Functions operate on expressions, and generate new values. We've already seen some of them:

- the *type* function, which takes an expression and returns its type
- the *int, float,* and *str* type-conversion functions

There are many built-in functions, which often reside in *modules*. In fact, for much of our data analysis, we will use specialized modules such as pandas, numpy, and scikit-learn. As an example, consider the *math* module.

In [40]:
```
import math  # think of this as loading a module so its functions becom
print('The square-root of 2 is', math.sqrt(2))
```

```
The square-root of 2 is 1.4142135623730951
```

Note the function call format: modulename.function. This allows

you to access different functions with the same name from different modules.

In [41]:
```
import math
import numpy
x = math.sqrt(2)
y = numpy.sqrt(2)
absolute_diff = abs(x - y)  # Get the absolute value of the difference
print('The absolute difference between the two square roots is', absolu
```

```
The absolute difference between the two square roots is 0.0
```

What functions does a module provide?

In [42]:
```
help(math)
```

```
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module provides access to the mathematical functio
ns
    defined by the C standard.

FUNCTIONS
    acos(x, /)
        Return the arc cosine (measured in radians) of x.

    acosh(x, /)
        Return the inverse hyperbolic cosine of x.

    asin(x, /)
        Return the arc sine (measured in radians) of x.

    asinh(x, /)
        Return the inverse hyperbolic sine of x.

    atan(x, /)
        Return the arc tangent (measured in radians) of x.

    atan2(y, x, /)
```

```
        Return the arc tangent (measured in radians) of y/
x.

        Unlike atan(y/x), the signs of both x and y are con
sidered.

    atanh(x, /)
        Return the inverse hyperbolic tangent of x.

    ceil(x, /)
        Return the ceiling of x as an Integral.

        This is the smallest integer >= x.

    copysign(x, y, /)
        Return a float with the magnitude (absolute value)
of x but the sign of y.

        On platforms that support signed zeros, copysign(1.
0, -0.0)
        returns -1.0.

    cos(x, /)
        Return the cosine of x (measured in radians).

    cosh(x, /)
        Return the hyperbolic cosine of x.

    degrees(x, /)
        Convert angle x from radians to degrees.

    erf(x, /)
        Error function at x.

    erfc(x, /)
        Complementary error function at x.

    exp(x, /)
        Return e raised to the power of x.

    expm1(x, /)
        Return exp(x)-1.

        This function avoids the loss of precision involved
```

```
in the direct evaluation of exp(x)-1 for small x.

    fabs(x, /)
        Return the absolute value of the float x.

    factorial(x, /)
        Find x!.

        Raise a ValueError if x is negative or non-integra
l.

    floor(x, /)
        Return the floor of x as an Integral.

        This is the largest integer <= x.

    fmod(x, y, /)
        Return fmod(x, y), according to platform C.

        x % y may differ.

    frexp(x, /)
        Return the mantissa and exponent of x, as pair (m,
e).

        m is a float and e is an int, such that x = m * 2.*
*e.
        If x is 0, m and e are both 0.  Else 0.5 <= abs(m)
< 1.0.

    fsum(seq, /)
        Return an accurate floating point sum of values in
the iterable seq.

        Assumes IEEE-754 floating point arithmetic.

    gamma(x, /)
        Gamma function at x.

    gcd(x, y, /)
        greatest common divisor of x and y

    hypot(x, y, /)
        Return the Euclidean distance, sqrt(x*x + y*y).
```

```
    isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)
        Determine whether two floating point numbers are cl
ose in value.

          rel_tol
            maximum difference for being considered "clos
e", relative to the
            magnitude of the input values
          abs_tol
            maximum difference for being considered "clos
e", regardless of the
            magnitude of the input values

        Return True if a is close in value to b, and False
otherwise.

        For the values to be considered close, the differen
ce between them
        must be smaller than at least one of the tolerance
s.

        -inf, inf and NaN behave similarly to the IEEE 754
Standard.  That
        is, NaN is not close to anything, even itself.  inf
and -inf are
        only close to themselves.

    isfinite(x, /)
        Return True if x is neither an infinity nor a NaN,
and False otherwise.

    isinf(x, /)
        Return True if x is a positive or negative infinit
y, and False otherwise.

    isnan(x, /)
        Return True if x is a NaN (not a number), and False
otherwise.

    ldexp(x, i, /)
        Return x * (2**i).

        This is essentially the inverse of frexp().
```

```
    lgamma(x, /)
        Natural logarithm of absolute value of Gamma functi
on at x.

    log(...)
        log(x, [base=math.e])
        Return the logarithm of x to the given base.

        If the base not specified, returns the natural loga
rithm (base e) of x.

    log10(x, /)
        Return the base 10 logarithm of x.

    log1p(x, /)
        Return the natural logarithm of 1+x (base e).

        The result is computed in a way which is accurate f
or x near zero.

    log2(x, /)
        Return the base 2 logarithm of x.

    modf(x, /)
        Return the fractional and integer parts of x.

        Both results carry the sign of x and are floats.

    pow(x, y, /)
        Return x**y (x to the power of y).

    radians(x, /)
        Convert angle x from degrees to radians.

    remainder(x, y, /)
        Difference between x and the closest integer multip
le of y.

        Return x - n*y where n*y is the closest integer mul
tiple of y.
        In the case where x is exactly halfway between two
multiples of
        y, the nearest even value of n is used. The result
```

```
is always exact.

    sin(x, /)
        Return the sine of x (measured in radians).

    sinh(x, /)
        Return the hyperbolic sine of x.

    sqrt(x, /)
        Return the square root of x.

    tan(x, /)
        Return the tangent of x (measured in radians).

    tanh(x, /)
        Return the hyperbolic tangent of x.

    trunc(x, /)
        Truncates the Real x to the nearest Integral toward
0.

        Uses the __trunc__ magic method.

DATA
    e = 2.718281828459045
    inf = inf
    nan = nan
    pi = 3.141592653589793
    tau = 6.283185307179586

FILE
    (built-in)
```

To create our own functions:

We can define our own functions, and it is very useful for building re-usable code. If some code will need to be used twice or more, it is better to create a function.

```
def function_name(parameter_1, parameter_2, parameter_3):
    # Do
```

```
    # Some
    # Processing

    return result
```

There are several points to note:

- A function is created using the **def** keyword, again ending with a colon.
- All indented statements below the def statement form the body of the function.
- A function can accept **parameters**, so that the function can be called repeatedly with different values for its parameters.
- A function can return values to the caller using the **return** keyword.

Example: Define a function **integer_division** that

- takes two positive integers a and b, and
- returns the quotient and the remainder of a / b.

- integer_division(20, 3) should return 6, 2
  - since 20 / 3 gives 6 with remainder 2
- integer_division(16, 9) should return 1, 7
  - since 16 / 9 gives 1 with remainder 7

In [43]:

```
def integer_division(a, b):
    # The function has two parameters a and b
    quotient = int(a / b)
    remainder = a - quotient * b
    print(a, '=', quotient, '*', b, '+', remainder)

    # Finally, we can return results. Here, we return both quotient and
    return quotient, remainder

# All statements above were indented, so they form the function body.
# We now call the function with some parameter values.
integer_division(16, 3)
```

```
16 = 5 * 3 + 1
```

Out[43]:
```
(5, 1)
```

In [44]:
```
# Re-use the function with different parameter values
integer_division(23, 5)
```

```
23 = 4 * 5 + 3
```

Out[44]:
```
(4, 3)
```

In [45]:
```
# Another way of calling a function is by naming the parameters
integer_division(b=5, a=32)
```

```
32 = 6 * 5 + 2
```

Out[45]:
```
(6, 2)
```

In [46]:
```
# Get input from user, using the built-in raw_input() function
a = int(input("Enter first parameter:"))
b = int(input("Enter second parameter:"))

# The results of the function can be used by the function caller.
# Our function returns quotient and remainder.
q, r = integer_division(a, b)
print('quotient =', q, 'and remainder =', r)
```

```
Enter first parameter:35
Enter second parameter:6
35 = 5 * 6 + 5
quotient = 5 and remainder = 5
```

Example: calculator

Let us combine all we've learnt into a calculator program (adapted from here). We'll add one extra wrinkle:

- For each function, write a "docstring", within three double-quotes. This will be printed if the user says *help(function_name)*.

Docstrings are not necessary, but they are good practice.

What are the steps in building a calculator?

1. We need to prompt the user about whether he/she wants to add/subtract/multiply/divide, or exit.
2. We need to write functions that do each of these operations, given two numbers.
3. We need to loop over the first two steps until the user exits.

Step 1: Let us write the first function, which prompts the user to input his or her choice.

In [57]:
```
def get_number(prompt):
    """Prompt the user for a number"""
    return int(input(prompt))

def menu():
    """print what calculations this calculator can perform
       and ask the user to choose.
       This function takes no parameters as input.
       It returns the user's choice."""
    print("""
      Your options are:
      1) Addition
      2) Subtraction
      3) Multiplication
      4) Division
      5) Quit calculator
    """)
    return get_number("Choose your option: ")
```

Step 2: We will write each of the functions our calculator must support.

In [58]:
```
def add(a,b):
    """this adds two numbers given"""
    print(a, "+", b, "=", a + b)

def sub(a,b):
    """this subtracts two numbers given"""
    print(b, "-", a, "=", b - a)
```

```
def mul(a,b):
    """this multiplies two numbers given"""
    print(a, "*", b, "=", a * b)

def div(a,b):
    """this divides two numbers given"""
    print(a, "/", b, "=", a / b)
```

Step 3: Bring it all together by repeating the first two steps until the user exits.

In [49]:

```
def operate_calculator():
    """The glue function that calls the other functions"""

    choice = menu()
    while choice != 5:  # choice 5 was to quit the calculator

        # Process this choice
        if choice == 1:
            add(get_number("Add this: "), get_number("to this: "))
        elif choice == 2:
            sub(get_number("Subtract this: "), get_number("from this: "
        elif choice == 3:
            mul(get_number("Multiply this: "), get_number("by this: "))
        elif choice == 4:
            div(get_number("Divide this: "), get_number("by this: "))
        else:
            print("Invalid choice", choice)
            print("Please choose again")

        # Get a new choice
        choice = menu()

    print("Thank you for using the calculator!")
```

In [50]:

```
### Now all functions are declared, and we finally run the code
operate_calculator()
```

```
Your options are:
1) Addition
2) Subtraction
3) Multiplication
4) Division
5) Quit calculator

Choose your option: 1
```

```
Add this: 2
to this: 3
2 + 3 = 5

      Your options are:
      1) Addition
      2) Subtraction
      3) Multiplication
      4) Division
      5) Quit calculator

Choose your option: 5
Thank you for using the calculator!
```

In [51]:
```
help(menu)
```

```
Help on function menu in module __main__:

menu()
    print what calculations this calculator can perform
    and ask the user to choose.
    This function takes no parameters as input.
    It returns the user's choice.
```

Example: Find cube root

Given a positive integer a, find it's cube root.

How do we do it?

The bisection algorithm

- Suppose you knew the square root was in the interval [lower, upper]
- Take the midpoint m = (lower + upper) / 2.0
- Take its cube
  - m_cubed = m * m * m

- Is m_cubed greater than a?
  - The cube root of a is smaller than m
  - Reduce upper to m
- Is m_cubed less than a?
  - The cube root of a is greater than m
  - Increase lower to m
- Repeat

In [52]:
```
def our_cube_root(a):
    lower = 0
    upper = a

    for i in range(20):  # iterate 20 times
        m = (lower + upper) / 2.0
        m_cubed = m * m * m
        if m_cubed < a:
            lower = m
        else:
            upper = m
    print('Best estimate cube root of', a, 'is', m)
    return m
```

In [53]:
```
our_cube_root(8)
```

```
Best estimate cube root of 8 is 1.9999923706054688
```

Out[53]:
```
1.9999923706054688
```

In [54]:
```
our_cube_root(27)
```

```
Best estimate cube root of 27 is 3.000014305114746
```

Out[54]:
```
3.000014305114746
```

Summary

- Python allows three basic kinds of **values** (integer, floating-point, and string),

- on which we can run **operations** (addition, subtraction, less-than, greater-than, ==), and
- which can be stored and processed in **variables.**

- The statements of a program and run one after another
- but the flow can be changed by **if-then-else** statements, **for-loops**, and **while-loops**

- Define **functions** to reuse code