Data Science and Advanced Programming — Lecture 3

Python Fundamentals I

Simon Scheidegger
Department of Economics, University of Lausanne, Switzerland

September 29th, 2025 | 12:30 - 16:00 | Internef 263

Roadmap

- Let's get started with Python (*Introduction to Computation and Programming Using Python, J. Guttag*)
  1. Python Basics
  2. Control flow: Branching and Loops
  3. String Manipulation
  4. Examples

  Covered in the TA session/Videos:
- Version Control (git)
- Coding Style
- Enabling Collaboration, Replicability, and Code Enhancement

Recall — What does a Computer do?

- Fundamentally:
  - performs **calculations**:
    several billion calculations per second even on a Laptop!
  - **remembers results**:
    $\sim$ terabytes of storage even on a Laptop!
- What kinds of calculations?
  - **built-in** to the language (basic types: additions, substraction,....)
  - Some calculations that **you define** as the programmer (built on top of the basic operations).

> computers only know what you tell them.
> $\rightarrow$ They only do what you tell them to do (REALLY FAST)!

Algorithm: A Basic Example – compute sqrt(x)

- Find the square root of a number x is g such that $g^* g = x$
- Algorithm (= recipe) for deducing square root of the number $x = 16$
  1. Start with a guess, g
  2. If **g*****g** is close enough to x, stop and say g is the answer
  3. Otherwise make a new guess by averaging g and x/g
  4. Using the new guess, repeat process until close enough

g	g*g	x/g	(g+x/g)/2
3	9	16/3	4.17
4.17	17.36	3.837	4.0035
4.0035	16.0277	3.997	4.000002

Recall — Algorithms

- sequence of simple steps.
- flow of control process that specifies when each step is executed.
- a means of determining when to stop (finite compute time).

$\rightarrow 1 + 2 + 3 =$ an algorithm!

Comuters => Machines

- **fixed program** computer
  - Pocket calculator (very limited in terms of capabilities).
- **stored program** computer
  - machine stores and executes instructions.
  - the computers we know nowadays.

Recall — a computer

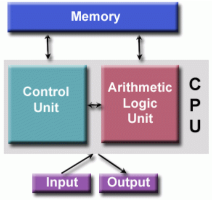
A computer is a machine that can:

- **Accept input**. Input could be entered by a human typing at a keyboard, received over a network, or provided automatically by sensors attached to the computer.
- **Execute a (mechanical) procedure**, that is, a procedure where each step can be executed without any thought.
- **Produce output.** Output could be data displayed to a human, but it could also be anything that effects the world outside the computer such as electrical signals that control how a device operates.

Basics: von Neumann Architecture

https://computing.llnl.gov/tutorials/parallel_comp

- Virtually all computers have followed this basic design. Comprised of **four main components**: Memory, Control Unit, Arithmetic Logic Unit, Input/Output.
- **Read/write, random access memory** is used to store both program instructions and data:
  - Program instructions are coded data which tell the computer to do something.
  - Data is simply information to be used by the program.
- **Control unit**
  - fetches instructions/data from memory, decodes the instructions and then sequentially coordinates operations to accomplish the programmed task.
- **Arithmetic unit**
  - performs basic arithmetic operations.
- **Input/Output**
  - interface to the human operator.



A (Python) Program

- A **program** is a sequence of definitions and commands
  - definitions evaluated.
  - commands executed by Python interpreter in a shell.
- **Commands** (statements) instruct interpreter to do something.
- can be typed directly in a shell or stored in a file that is read into the shell and evaluated.

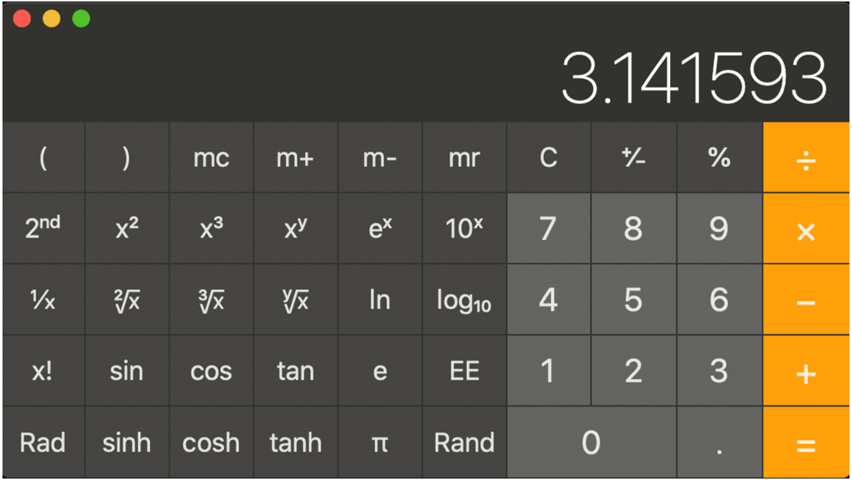
Recall — Some features

- Python is a high level language suitable for rapid development.
- It has a relatively small core language **supported by many libraries.**
- A multi-paradigm language, in that multiple programming styles are supported (procedural, object-oriented, functional,...).
- **Interpreted rather than compiled.**

Recall — Syntax and Design

- One nice feature of Python is its **elegant syntax** — we'll see many examples later on.
- **Elegant code** might sound superfluous but in fact it's highly beneficial because it makes the syntax **easy to read and easy to remember.**
- **Remembering** how to read from files, sort dictionaries and other such routine tasks means that you don't need to break your flow in order to hunt down correct syntax.
- Closely related to **elegant syntax** is **elegant design.**
- Features like iterators, generators, list comprehensions, etc. make Python highly expressive, **allowing you to get more done with less code.**

1. Python Basics



Python Setup

A bare-bones development environment consists of:

- A text editor (e.g., gedit, emacs, vim)
- The Python interpreter (it is installed by default on Ubuntu and almost any other Linux distribution)
- A terminal application to run the interpreter in.

See http://wiki.python.org/moin/IntegratedDevelopmentEnvironments for a commented list of IDEs with Python support.

Python Basics I

Python is an *interpreted* language.

It also features an interactive “shell” for evaluating expressions and statements immediately.

The Python shell is started by invoking the command `python` in a terminal window.

Python Basics II

Expressions can be entered at the Python shell prompt (the '>>>' at the start of a line); they are evaluated and the result is printed:

```
>>> 2+2
4
```

A line can be continued onto the next by ending it with the character '\'; for example:

```
>>> "hello" + \
... " world!"
'hello world!'
```

The prompt changes to '. . .' on continuation lines.

Reference:

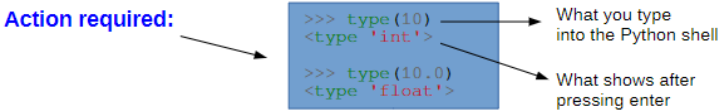
http://docs.python.org/reference/lexical_analysis.html#line-structure

Objects

- Programs manipulate **data objects**.
- objects have a **type** that defines the kinds of operations $(^*, +, -, \ldots)$ programs can do to them:
  - $\rightarrow$ *Simon is a human: he can walk, speak French with an accent Fédérale, etc.*
  - $\rightarrow$ 3 *is an integer, so we can* $+, -, ^*, /, ^{**} \ldots$
- **Objects** are
  - scalar (cannot be subdivided into smaller "sub-items", e.g., $a = 1$).
  - non-scalar (have internal structure that can be accessed, e.g., $b = [1, 2, 3]$).

Scalar Objects

- `int` - represent integers, for example `666`
- `float` - represent real numbers, eg., `6.66`
- `bool` - represent Boolean values `True` and `False`
- `NoneType` - special and has one value: `None` (absence of a type)
- use `type()` to see the type of an object



Action required: Type conversions

- Python can convert object of one type to another
- `float(10)` converts integer 10 to float 10.0
- `int(10.9)` truncates float 10.9 to integer 10 (rounding towards zero)

```
>>> float(10)
10.0
>>> int(10.9)
10
```

Print to the Terminal

demo/example1a.py

- In order to show output from code to a user, use print command

```
>>> 10 + 2
12
>>> print(10+2)
12
```

- You see the output only in an interactive shell.
- **If you use a *.py file, you explicitly need to enforce output.**
- Try this in a file `print.py`

```
a = 3+4
print('no printout so far')
print('here we go')
print(a)
```

Action Required

- Run demo/example1a.py:
  `python example1a.py`
- Run demo/example1b.py:
  `python example1b.py`

```python
a = 3
b = 2

sum = a + b
print('no printout of the result so far')
```

```python
a = 3
b = 2

sum = a + b
print('no printout of the result so far')
print("here we go with the result")
print(sum)
```

Create Expressions

- Combine objects and operators to form expressions
- An expression has a value, which has a type
- syntax for a simple expression:

```
<object> <operator> <object>
```

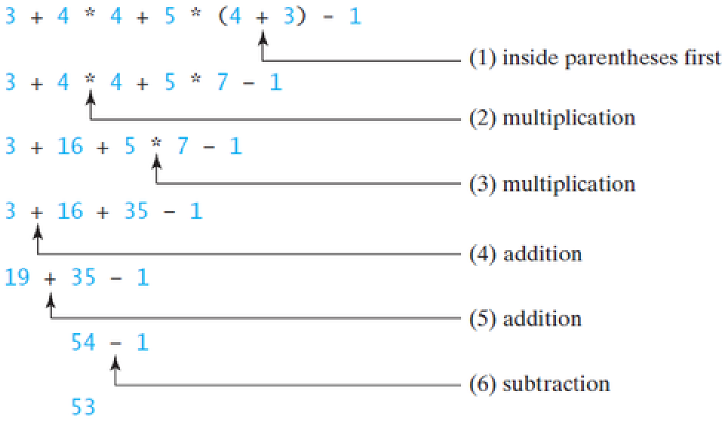
- `i + j` a sum
- `i - j` the difference
- `i * j` the product
- `i / j` division (be careful with result: `type(1 / 2)`, `type(1/2.0)`)
  - Note: if both `i, j` are `int`, the result is `int`, if either or both are floats, result is a float
- `i % j` the remainder when `i` is divided by `j`
- `i ** j` → `i` to the power of `j`

Summary: Numeric Operators

- Parentheses are used to tell Python to do these operations first
- Operator precedence without parentheses
- "+" and "-" are executed **left to right**, as appear in expression

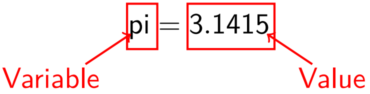
Name	Meaning	Example	Result
+	Addition	34 + 1	35
-	Subtraction	34.0 - 0.1	33.9
*	Multiplication	300 * 30	9000
/	Float Division	1 / 2	0.5
//	Integer Division	1 // 2	0
**	Exponentiation	4 ** 0.5	2.0
%	Remainder	20 % 3	2

Order of Expressions



Assignments

- The equal sign (=) is an **assignment** of a value to a variable name



- Its value is stored in the computer memory
- An assignment **binds a name to value.**
- You can retrieve the value associated with name or variable by invoking the name, by typing pi.

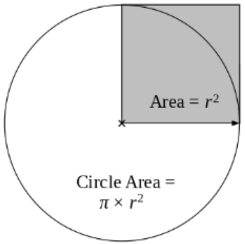
```
>>> pi = 3.1415
>>> pi
3.1415
```

Expressions

demo/example2.py

- Why should you give names to values of expressions?
  - $\rightarrow$ to reuse names instead of values
  - $\rightarrow$ code is easier to read
  - $\rightarrow$ easier to change code later

```
>>> pi = 3.14159
>>> radius = 2.0
>>> area = pi * (radius**2)
```



Programming logic vs. math logic

In programming, we do not "solve for x"

```
>>> pi = 3.14159
>>> radius = 2.0
>>> area = pi * (radius**2)
>>> radius = radius + 1
>>> radius
3.0
```

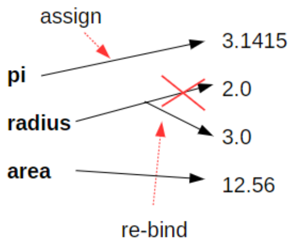
- $\rightarrow$ "=" is an assignment.
- $\rightarrow$ It strictly means "the expression on the right evaluated to a value on the left."

Change Assignments

demo/example2b.py

- You can re-assign (re-bind) variable names using new assignment statements.
- Previous value may still stored in memory but lost the handle for it.
- The value for area does not change until you tell the computer to do the calculation again.

```
>>> pi = 3.14159
>>> radius = 2.0
>>> area = pi * (radius**2)
>>> radius = radius + 1
```



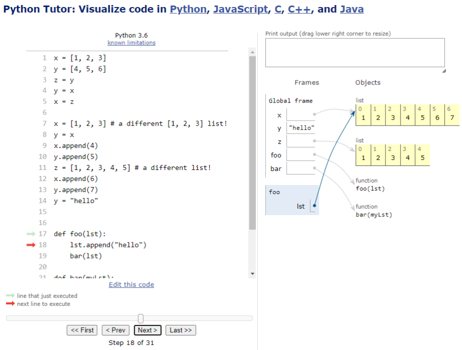
Augmented Assignments

demo/example2c.py

Operator	*Example*	*Equivalent*
`+=`	`i += 8`	`i = i + 8`
`-=`	`f -= 8.0`	`f = f - 8.0`
`*=`	`i *= 8`	`i = i * 8`
`/=`	`i /= 8`	`i = i / 8`
`%=`	`i %= 8`	`i = i % 8`

Another helpful online tool

The Online Python Tutor is a free tool to visualize the execution of programs step-by-step.



Feel free to use it for the course exercises and your own code:
http://pythontutor.com/visualize.html

Object: Characters, Strings

demo/example3.py

- Sequences of Characters: Letters, special characters, spaces, digits
- Enclose in quotation marks or single quotes (be consistent!)

```
>>> hi = "hello there"
```

- To concatenate strings, use +

```
>>> name = "Mickey"
>>> greet = hi + name
>>> greeting = hi + " " + name
```

- Do some operations (*) on a string as defined in Python docs

```
>>> nonsense = "howdy" + " " + name * 5
```

Input and Output

demo/example4.py

- "`print`" is used to output (text messages, numbers,...) to the console.

```
x = 666
print(x)
x_str = str(x)   #Cast the number to a string
print("my favorite  number is", x, ".", "x =", x)  # use commas -> different objects
print("my favorite number is " + x_str + ". " + "x = " + x_str) # one big string object
```

Interactive Program: Input and Output (II)

demo/example5.py

- `Input(" ")` reads whatever is inside the quotation marks.
- The user types in something and presses enter.
- `Input(" ")` binds that value to a variable

```
>>> text = input("Type some meaningful text string... ")
>>> print(5*text)
```

- The input gives you a string so must cast if working with numbers

```
>>> num = int(input("Type a number... "))
>>> print(5*num)
```

Add tests to the Code: Comparison operators

- Assume i and j are variable names
- Comparisons shown below evaluate to a Boolean (logical)
  - `i > j`
  - `i >= j`
  - `i < j`
  - `i <= j`
  - `i == j` → equality test, True if `i` is the same as `j`
  - `i != j` → inequality test, True if `i` is NOT the same as `j`
- These tests work on int, float, string

```
>>> i = 3
>>> j = 5

>>> j < i
False
```

Testing: Logical operations

- Assume `a` and `b` are variable names (with Boolean values)
- `not a` — True if `a` is `False`, False if `a` is True
- `a and b` — `True` if both are `True`
- `a or b` — `True` if either or both are `True`

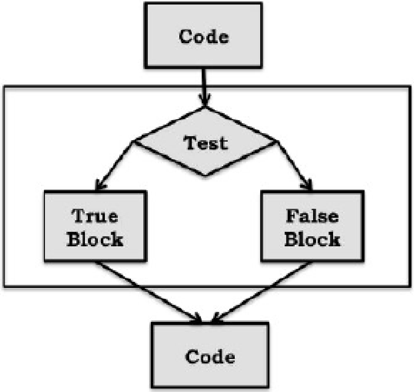
```
>>> a='true'
>>> b='true'
>>>a==b
Out[17]: True
>>>a!=b
Out[18]: False
```

2. Control flow: Branching and Loops

Control flow in the real world

```
         ⎧ print("Programming is fun!")
         ⎪ print("Programming is fun!")
         ⎪ print("Programming is fun!")
         ⎪ print("Programming is fun!")
         ⎪ print("Programming is fun!")
         ⎪ print("Programming is fun!")
100      ⎪
times    ⎨ ...
         ⎪
         ⎪ ...
         ⎪
         ⎪ ...
         ⎪ print("Programming is fun!")
         ⎪ print("Programming is fun!")
         ⎩ print("Programming is fun!")
```

Branching



Branching: if-else statements

`if-elif-else` statements:

```
if x < 0:
  print("x is less than zero")

if x < 0:
  print("x is less than zero")
else:
  print("x is greater or equal zero")

if x < 0:
  print("x is less than zero")
elif x > 0:
  print ("x is greater than zero")
else:
  print ("x is zero")
```

Indentation matters in Python to denote blocks of code

demo/example6.py

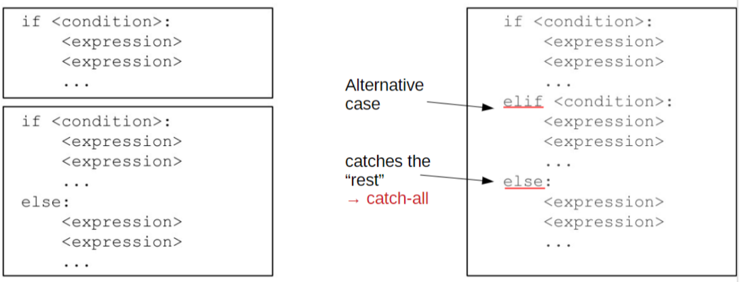
```
if x < 0:          ← Colon required!
    print("x is less than zero")
```

```
if x < 0:
    print("x is less than zero")
else:
    print("x is greater or equal zero")
```

```
if x < 0:
    print("x is less than zero")
elif x > 0:
    print("x is greater than zero")
else:
    print("x is zero")
```

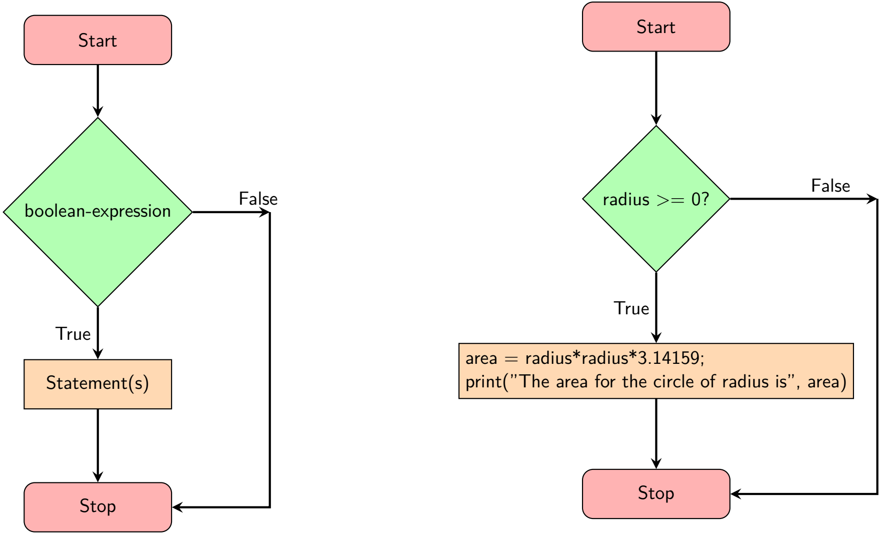
PLEASE: 4 spaces indentation

`if-else` branching in general



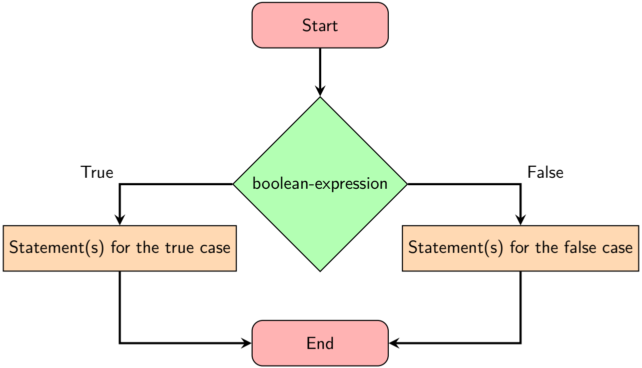
- `<condition>` has a value `True` or `False`
- evaluate expressions in that code block if `<condition>` is `True`

Visualized: One-way if-statements



Visualized: Two-way if-statements

```
if boolean-expression:
  statement(s)-for-the-true-case
else:
  statement(s)-for-the-false-case
```



Multiple alternative if-statements

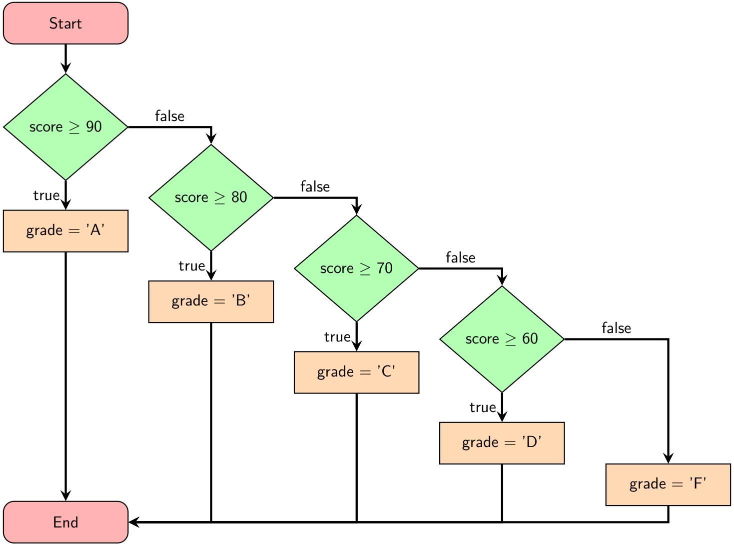
```
if score >= 90.0:
    grade = 'A'
else:
    if score >= 80.0:
        grade = 'B'
  else:
        if score >= 70.0:
            grade = 'C'
        else:
            if score >= 60.0:
                grade = 'D'
            else:
                grade = 'F'
```

Equivalent

This is better

```
if score >= 90.0:
    grade = 'A'
elif score >= 80.0:
    grade = 'B'
elif score >= 70.0:
    grade = 'C'
elif score >= 60.0:
    grade = 'D'
else:
    grade = 'F'
```

Flowchart


Start
score ≥ 90
false
true
grade = 'A'
score ≥ 80
false
true
grade = 'B'
score ≥ 70
false
true
grade = 'C'
score ≥ 60
false
true
grade = 'D'
grade = 'F'
End

Another Example

demo/example6a.py

```
x = float(input("Enter a number for x: "))
y = float(input("Enter a number for y: "))
if x == y:
   print("x and y are equal")
   if y != 0:
       print("therefore, x / y is", x/y)
elif x < y:
   print("x is smaller")
elif x > y:
   print("y is smaller")
```

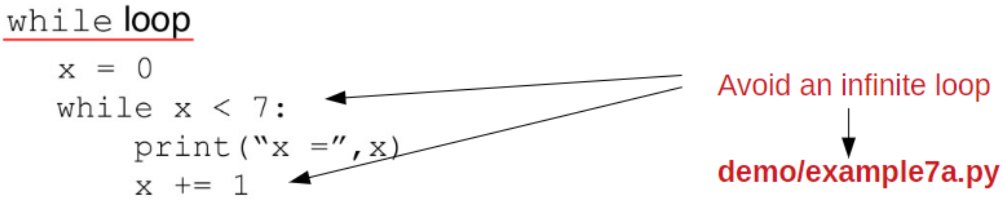
Another Example

demo/example6c.py

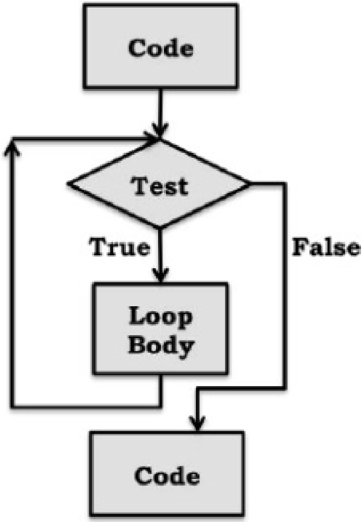
```
x = 12/3 - 2        # this is a comment
y = "Hola"
z = 3.14            # another comment

if (y == "Hola" or z >= 3):
    x = x + 2
    y = y + " mundo!" # string concatenation
    print(y)
    print(x)

year, month , day  = 1943, 6, 15
hour, minute, second = 23, 6, 54
if 1900 < year < 2100 and 1 <= month <= 12  \
   and 1 <= day <= 31 and 0 <= hour < 24    \
   and 0 <= minute < 60 and 0 <= second < 60:
      print("Looks like a valid date!")
```

Control flow: while loop

demo/example7.py



While loop — iteration



Control flow: while loop in general

```
while <condition>:
    <expression>
    <expression>
    ...
```

- `<condition>` evaluates to a Boolean
- if `<condition>` is `True`, do all the steps inside the while code block
- check `<condition>` again
- repeat until `<condition>` is `False`

Loops — the Motivation

- Suppose that you need to print a string (e.g., "Programming is fun!") a hundred times.
- It would be tedious to have to write the following statement a hundred times:

```
print("Programming is fun!")
```

- So, how do you solve this problem?

Opening Problem

100 times

```
print("Programming is fun!")
print("Programming is fun!")
print("Programming is fun!")
print("Programming is fun!")
print("Programming is fun!")
print("Programming is fun!")

...

...

...
print("Programming is fun!")
print("Programming is fun!")
print("Programming is fun!")
```

Control flow: for loops I

demo/example8.py

- To iterate through numbers in a sequence, use "for" loop

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

- Shortcut for the for loop

```
for i in range(10):
  print(i)
```

Control flow: `for` loops II

`demo/example8.py`

- In general, we use: `range (start, stop, step)`
- The default values `are start=0` and `step=1` and are optional
- **The loop continues until the counter value is `stop-1`**

```
sum = 0
for i in range(5,7):
  sum += i
print(sum)
```

```
sum = 0
for i in range(40,50,2):
  sum += i
print(sum)
```

`demo/example8a.py`
`demo/example8b.py`

Control flow: `for` loops III

`demo/example8.py`

for loop: in python only “for-each” form of loops

```
for <item> in <collection>:
  <statements>

for item in [0, "a", 7, 1j]:
  print(item)

for letter in "StRinG":
  print(letter)
```

Example

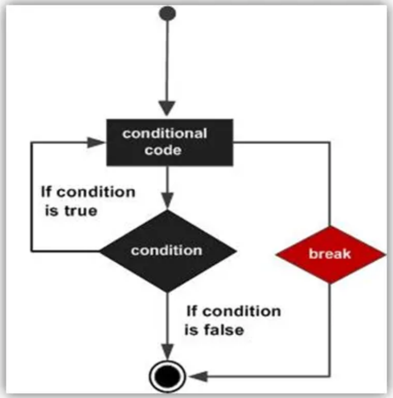
demo/example9.py

```
for item in [0,"a",7,1,j]:
    print(item)

for letter in "StRiNg":
    print(letter)

for i in range(5):
    print(i)

lst = ["Suzuki","Kawasaki","Aprilia","Ducati"]
# use enumerate below!!!
# for i in range(len(lst)):
#     print(i,lst[i])
for (i,item) in enumerate(lst):
    print(i,item)
```

Stop within a loop — break statement

- If you want to immediately exit a loop → break
- It skips remaining expressions in the code block.
- Note: it exits only innermost loop.

```
while <condition_1>:
  while <condition_2>:
    <expression_a>
    break
    <expression_b>
  <expression_c>
```



break — an example

demo/example10.py

```
var = 10
while var > 0:
    print('Current variable value :', var)
    var = var -1
    if var == 5:
        break

print ("Test done!")
```

3. String Manipulation

Strings → Sequences of Characters

Square brackets are used to perform indexing into a string to get the value at a certain index/position.

```
s = ''abcd''
```

indices: 012 → **indexing always starts at 0**
indices: $-3-2-1$ → last element **always at index -1**

- `s[0]` evaluates to "a"
- `s[1]` evaluates to "b"
- `s[2]` evaluates to "c"
- `s[3]` evaluates to "d"
- `s[4]` trying to index out of bounds, error

- `s[-1]` evaluates to "d"
- `s[-2]` evaluates to "c"
- `s[-3]` evaluates to "b"
- `s[-4]` evaluates to "a"

Mutable vs. Immutable types

- **Mutable types**
  - Can change their contents / members
  - `lists`, `dicts`, user-defined types
- **Immutable types**
  - Cannot change their contents / members
  - most built-in types (`int`, `float`, `bool`, `str`, `tuple`)

Recap: Strings and Loops

`demo/example11.py`

The two code snippets below do the same thing: they loop over the characters in the string.

```
s = "mickey"
for index in range(len(s)):
    if s[index] == 'i' or s[index] == 'y':
        print("There is an i or y")
```

```
for char in s:
    if char == 'i' or char == 'y':
        print("There is an i or y")
```

4. Examples

Example: Guess and Check

demo/example12.py

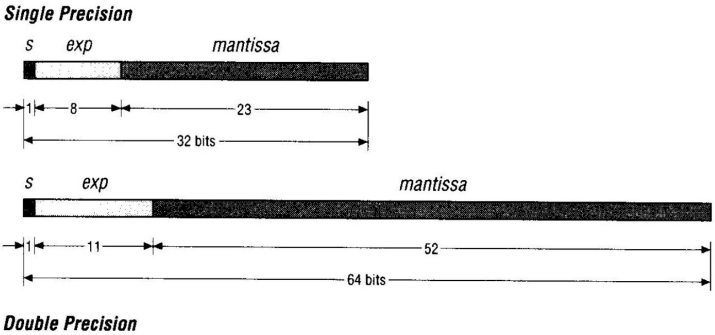
We want to guess the cube root:

```
cube = 8
for guess in range(abs(cube)+1):
    if guess**3 >= abs(cube):
        break
if guess**3 != abs(cube):
    print(cube, 'is not a perfect cube')

else:
    if cube < 0:
        guess = -guess
    print('Cube root of '+str(cube)+' is '+str(guess))
```

Approximate Solutions

- We are performing finite-precision arithmetic on a computer.
- We need to define a good enough solution.
  - start with a guess and increment by some small value.
  - keep guessing if $\left|\text{guess}^3 - \text{cube}\right| >= \varepsilon$ for some small ε.
  - decreasing increment size $\implies$ slower program.
  - increasing $\varepsilon \implies$ less accurate answer.

Recall — IEEE Floating Point Representation



Type	Exponent	Mantissa	Smallest	Largest	Base 10 accuracy
float	8	23	1.2E-38	3.4E+38	6-9
double	11	52	2.2E-308	1.8E+308	15-17

An approximate solution

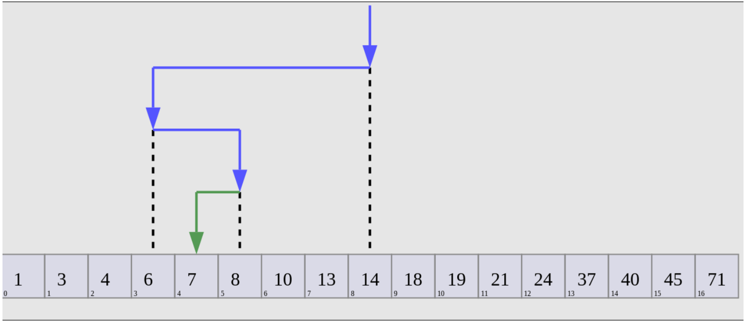
demo/example13.py

```
cube = 27.8
epsilon = 0.1
guess = 0.0
increment = 0.01
num_guesses = 0
# look for close enough answer and make sure
# didn't accidentally skip the close enough bound and thus overshoot
while abs(guess**3 - cube) >= epsilon and guess <= cube:
    guess += increment
    num_guesses += 1
print('num_guesses =', num_guesses)
if abs(guess**3 - cube) >= epsilon:
    print('Failed on cube root of', cube, "with these parameters.")
else:
    print(guess, 'is close to the cube root of', cube)
```

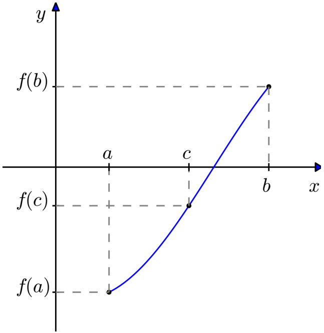
Another example: Bisection

Binary search algorithm - https://en.wikipedia.org/wiki/Binary_search_algorithm

- The idea of Bisection:
- half interval each iteration
- new guess is halfway in between



Visualization of the binary search algorithm where 7 is the target value.



Bisection applied to the cube root

demo/example14.py

```
cube = 27.8
# won't work with x < 1 because initial upper bound is less than ans
#cube = 0.25
epsilon = 0.01
num_guesses = 0
low = 0
high = cube
guess = (high + low)/2.0
while abs(guess**3 - cube) >= epsilon:
   if guess**3 < cube:
       # look only in upper half search space
       low = guess
   else:
       # look only in lower half search space
       high = guess
   # next guess is halfway in search space
   guess = (high + low)/2.0
   num_guesses += 1
print('num_guesses =', num_guesses)
print(guess, 'is close to the cube root of', cube)
```

Bisection applied to the cube root

- search space of size N
  - first guess: N/2
  - second guess: N/4
  - k-th guess: $N/2^k$
- The guess converges on the order of $\log_2 N$ steps
- The Bisection search works when the value of function varies monotonically with input.
- The code as shown only works for positive cubes > 1 - why?

Summary → GIT next