Programming with Python

Stefan Güttel, guettel.com (http://guettel.com)

Contents:

Strings

Strings are usually described as character sequences. However, in Python, they are better described as tuples of characters, in a sense that they function very much like immutable lists.

A string literal can be defined in the following ways:

In [1]:

```
tring1 = 'This string is enclosed in single quotes, so we can use "double quote
" easily.'
tring2 = "This string is enclosed in double quotes, so we can use 'single quote
' easily."
tring3 = '''Triple quotes make it easy to put anything (but those same triple qu
tes) in a string,
ven a new line character!'''
tring4 = """Triple quotes make it easy to put anything (but those same triple qu
tes) in a string,
ven a new line character!"""
```

Character sequences starting with a backslash (\) have a special meaning. Some of them can be seen here:

In [2]:

```
rint("Literal backslash: '\\'")
rint("Quotes: \"...\"")
rint("Quotes: '...'")
rint('Quotes: "..."')
rint('Quotes: \'...\'')
rint("New line: \"new\nline\"")
```

```
Literal backslash: '\'
Quotes: "..."
Quotes: '...'
Quotes: "..."
Quotes: '...'
New line: "new
line"
```

So, `\\` inside a string means a single backslash (\), '\n' means new-line character, `\"` and `\'` mean quotation marks (instead of the end of the string), etc.

More about string literals can be read in the official reference (https://docs.python.org/3/reference/lexical_analysis.html#literals).

If we need to put many backslashes in a string (required by some applications, for example, regular expressions), we can also define so called *raw strings*, by prepending a string literal with `r`:

In [3]:

```
rint(r"Double backslash: '\\'")
rint(r"Quotes prepended with backslashes: \"...\"")
rint(r"Quotes: '...'")
rint(r'Quotes: "..."')
rint(r'Quotes prepended with backslashes: \'...\'')
rint(r"No new line, just a bunch of backslashes: \"new\nline\"")
```

```
Double backslash: '\\'
Quotes prepended with backslashes: \"...\"
Quotes: '...'
Quotes: "..."
Quotes prepended with backslashes: \'...\'
No new line, just a bunch of backslashes: \"new\nline\"
```

As we have already seen, `print` can be used to print a string:

In [4]:

```
rint(string1)
rint(string2)
rint(string3)
rint(string4)
```

```
his string is enclosed in single quotes, so we can use "double quot
s" easily.
his string is enclosed in double quotes, so we can use 'single quot
s' easily.
riple quotes make it easy to put anything (but those same triple qu
tes) in a string,
ven a new line character!
riple quotes make it easy to put anything (but those same triple qu
tes) in a string,
ven a new line character!
```

In fact, we have been using this since we first encountered `print`:

In [5]:

```
rint("Hello, World!")
```

```
Hello, World!
```

We have also seen how to load a string:

In [6]:

```
 = input("Type a string: ")
rint("Here is your string:", x)
```

```
Type a string: This is a string
Here is your string: This is a string
```

and how to convert it to an integer or a "real" number:

In [7]:

```
  = " -17 "
  = "  -1.719e1  "
  = 17.19
 rint("(x, y) =", (x, y))
 rint("x =", '"' + x + '"')
 rint("y =", '"' + y + '"')
 rint("int(x) =", int(x))
 rint("float(y) =", float(y))
 rint("str(z) =", '"' + str(z) + '"')
```

```
(x, y) = (' -17 ', '  -1.719e1  ')
x = " -17 "
y = "  -1.719e1  "
int(x) = -17
float(y) = -17.19
str(z) = "17.19"
```

Of course, if the string contains something other than an integer or a "real" number, an error occurs. For example:

In [8]:

```
 rint(float("17.19+11.13"))
```

```
---------------------------------------------------------------------
-------
 alueError                                 Traceback (most recent cal
  last)
 ipython-input-8-8d38ce842cec> in <module>()
----> 1 print(float("17.19+11.13"))

 alueError: could not convert string to float: '17.19+11.13'
```

Standard list/tuple operations work as we've seen with lists:

In [9]:

```
  = "Python is hard."
  = "Or, is it?"
 rint("The string:                    ", x)
 rint("The first character:           ", x[0])
 rint("The first 6 characters:        ", x[:6])
 rint("Characters with indices 7-8:   ", x[7:9])
 rint("The last 5 characters:         ", x[-5:])
  = x[:9] + " not" + x[9:]
 rint("The new string:                ", x)
 rint("17 dots:                       ", "." * 17)
 rint("Concatenation of strings:      ", x + " " + y)
 rint("Concatenation of string slices:", x[:-1] + ", " + y[4:])
```

```
The string:                     Python is hard.
The first character:            P
The first 6 characters:         Python
Characters with indices 7-8:    is
The last 5 characters:          hard.
The new string:                 Python is not hard.
17 dots:                        .................
Concatenation of strings:       Python is not hard. Or, is it?
Concatenation of string slices: Python is not hard, is it?
```

As we said before, strings are immutable. This means they cannot be changed:

In [10]:

```
 [1] = "y"
```

```
---------------------------------------------------------------------
-------
 ypeError                                 Traceback (most recent cal
  last)
 ipython-input-10-2c9b60f0481f> in <module>()
----> 1 x[1] = "y"

 ypeError: 'str' object does not support item assignment
```

Neither do strings support in-place sorting or reversal, nor can they be extended or appended to:

In [11]:

```
 .sort()
```

```
---------------------------------------------------------------------
-------
 ttributeError                            Traceback (most recent cal
  last)
 ipython-input-11-42dad5a67ac3> in <module>()
----> 1 x.sort()

 ttributeError: 'str' object has no attribute 'sort'
```

However, these operations can be done by creating new strings. Something like this:

In [12]:

```
  = "Python is not hard"
  = sorted(x)
 rint(x)
```

```
[' ', ' ', ' ', 'P', 'a', 'd', 'h', 'h', 'i', 'n', 'n', 'o', 'o',
'r', 's', 't', 't', 'y']
```

Notice that the sorting has created a list (of characters) and not a new string.

In order to fix this (so, to get a string), we have to merge the elements of a list into a string. This is done by `join`:

In [13]:

```
  = "Python is not hard"
 st = sorted(x)
  = "".join(lst)
 rint('"' + y + '"')
  = ",".join(lst)
 rint('"' + z + '"')
```

```
"   Padhhinnoorstty"
" , , ,P,a,d,h,h,i,n,n,o,o,r,s,t,t,y"
```

We can also do the opposite: `split` a string into a list of substrings, by some separator. For example,

In [14]:

```
 _string = "17;19;23"
 _list = a_string.split(";")
 rint(a_list)
```

```
['17', '19', '23']
```

To just convert a string to the list of its characters, we use the function `list()` which we have seen while working with iterables:

In [15]:

```
 rint(list(a_string))
```

```
['1', '7', ';', '1', '9', ';', '2', '3']
```

Python has a truly rich support for strings. Some of the available operations are:

In [16]:

```
  = "this is a string with several words"
 rint("x:                                ", x)
 rint("Length of x:                      ", len(x))
 rint("Capitalized x:                    ", x.capitalize())
 rint("Does x end with \"ords\"?           ", x.endswith("ords"))
 rint("Does x end with \"orDs\"?           ", x.endswith("orDs"))
 rint("Does x end with \"word\"?           ", x.endswith("word"))
 rint("x with \"a string\" replaced by \"x\":", x.replace("a string", "x"))
 rint("Does x contain a \"ring\"?          ", "ring" in x)
 rint("The index of the first character of the first \"ring\" in x:", x.find("rin
 "))
 rint("Does x have any cased characters and are they all lowercase?", x.islower
 ))
```

```
 :                                this is a string with several wor
 s
 ength of x:                      35
 apitalized x:                    This is a string with several wor
 s
 oes x end with "ords"?           True
 oes x end with "orDs"?           False
 oes x end with "word"?           False
  with "a string" replaced by "x": this is x with several words
 oes x contain a "ring"?          True
 he index of the first character of the first "ring" in x: 12
 oes x have any cased characters and are they all lowercase? True
```

See the full list of the supported string operations in the official documentation (https://docs.python.org/3/library/stdtypes.html#string-methods).

Regular expressions

For a truly powerful (but sometimes complex) way to analyze strings, read about *regular expressions* in A.M. Kuchling's HOWTO (https://docs.python.org/3/howto/regex.html). Those already familiar with regular expressions can read Python `re` reference (https://docs.python.org/3/library/re.html).

There are numerous practical applications to regular expressions: data analysis, syntax highlighting, system maintenance and configuration (for example ModRewrite in Apache), etc. Most of the modern languages support the same regular expressions syntax (so called Perl compatible), which makes them quite portable. While often fairly simple, they can also get downright scary (http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html).

Regular expressions are beyond the scope of this course. You may use them if you want to, but it is not assumed that you are familiar with them.

String formats

Forming a string from several values can be tricky. For example, if we had `x = 17` and `y = 19`, and wanted to print that information in the form

```
      = 17, y = 19
```

we have so far used the following syntax:

In [17]:

```
  = 17
  = 19
 rint("Wrong:   x =", x, ", y =", y)
 rint("Correct: x = " + str(x) + ", y =", y)
```

```
Wrong:   x = 17 , y = 19
Correct: x = 17, y = 19
```

We had to convert the number `x` to a string using `str(x)` and then "glue" it with the string literals `"x = "` and `", y ="` because otherwise `print` would add spaces.

Even with such conversions, we are limited with what we can do. For example,

- how would we print the value of `pi` up to only 3 decimal places?
- how to make it easy to customize messages in several languages, i.e., to tell Python to write something like

> Student <student's name> has enrolled in <name of the university> where {he/she} studied <what did they study>.

This is where *formats* come in handy.

There are three wide-spread ways to format strings in Python:

1. C-like formatting,
2. `format` function,
3. various templating engines (https://wiki.python.org/moin/Templating#Templating_Engines).

The `format` function is very easy for a basic use, and it has a powerful support for advanced formatting (like named arguments, embedded formats, etc.).

The C-like formatting is somewhat shorter to invoke and it is common to many widely used languages (C/C++, PHP, Perl,...).

The third way, templating, is more powerful, usually with support for complex data structures. This is often used to produce whole documents, instead of just short strings.

`format` function

The basic use of the `format` function is like this:

```
   "some string containing {}, possibly more than one".format(x)
```

In this case, `{}` will be replaced by the value in `x` and the new string will be returned.

In [18]:

```
  = "the pairs of braces"
  = "some string containing {}, possibly more than one"
rint("Unformated:\n", s)
rint("Formated:\n", s.format(x))
```

```
Unformated:
 some string containing {}, possibly more than one
Formated:
 some string containing the pairs of braces, possibly more than one
```

The function accepts multiple arguments, usually one for each of the braces pairs. It then replaces the first pair of braces with the first argument, the second one with the second pair, and so on.

In [19]:

```
rint("Student {} was enrolled in {} where {} studied {}.".format(
   "Benedict Timothy Carlton Cumberbatch",
   "The University of Manchester",
   "he",
   "Drama"
)
```

```
tudent Benedict Timothy Carlton Cumberbatch was enrolled in The Uni
ersity of Manchester where he studied Drama.
```

Using the *unpacking* operator * for sequences, we can also do it like this:

In [20]:

```
rgs = [
   "Benedict Timothy Carlton Cumberbatch",
   "The University of Manchester",
   "he",
   "Drama"

rint("Student {} was enrolled in {} where {} studied {}.".format(*args))
```

```
tudent Benedict Timothy Carlton Cumberbatch was enrolled in The Uni
ersity of Manchester where he studied Drama.
```

However, if we change the sentence to imply a different order of the parameters, we're in trouble:

In [21]:

```
rgs = [
   "Benedict Timothy Carlton Cumberbatch",
   "The University of Manchester",
   "he",
   "Drama"

rint("{} studied {} at {}.".format(*args))
```

```
enedict Timothy Carlton Cumberbatch studied The University of Manch
ster at he.
```

There is an easy fix for this:

In [22]:

```
rgs = [
   "Benedict Timothy Carlton Cumberbatch",
   "The University of Manchester",
   "he",
   "Drama"

rint("{0} studied {3} at {1}.".format(*args))
```

```
Benedict Timothy Carlton Cumberbatch studied Drama at The University
of Manchester.
```

Named arguments can also come in handy:

In [23]:

```
rint("Student {name} was enrolled in {uni} where {heshe} studied {studies}.".for
at(
   name="Benedict Timothy Carlton Cumberbatch",
   uni="The University of Manchester",
   heshe="he",
   studies="Drama"
)
rint("{name} studied {studies} at {uni}.".format(
   name="Benedict Timothy Carlton Cumberbatch",
   uni="The University of Manchester",
   heshe="he",
   studies="Drama"
)
```

```
tudent Benedict Timothy Carlton Cumberbatch was enrolled in The Uni
ersity of Manchester where he studied Drama.
enedict Timothy Carlton Cumberbatch studied Drama at The University
f Manchester.
```

or, using the *unpacking operator* `**` for dictionaries:

In [24]:

```
rgs = {
   "name": "Benedict Timothy Carlton Cumberbatch",
   "uni": "The University of Manchester",
   "heshe": "he",
   "studies": "Drama"

rint("Student {name} was enrolled in {uni} where {heshe} studied {studies}.".for
at(**args))
rint("{name} studied {studies} at {uni}.".format(**args))
```

```
tudent Benedict Timothy Carlton Cumberbatch was enrolled in The Uni
ersity of Manchester where he studied Drama.
enedict Timothy Carlton Cumberbatch studied Drama at The University
f Manchester.
```

Notice that `format` with named arguments doesn't give an error even if some of the parameters are not used (`"heshe"`, in our example).

The use that we have just seen makes it easy to create localised messages:

In [25]:

```
essages = {
   "language": {
       "en": "English",
       "cy": "Cymraeg",
   },
   "study-info": {
       "en": "{name} studied {studies} at {uni}.",
       "cy": "{name} Astudiodd {studies} am {uni}.",  # Google Translated to Wel
h
   },
   "score": {
       "en": "{name} had {score} in the course \"{course}\".",
       "cy": "{name} wedi {score} yn y cwrs \"{course}\".",  # Google Translated
o Welsh
   },
   # ...

 ...
rgs = {
   "course": "Programming with Python",
   "name": "Benedict Timothy Carlton Cumberbatch",
   "score": 17,
   "studies": "Drama",
   "uni": "The University of Manchester",

or lang, name in messages["language"].items():
   print("Language: {} ({})".format(name, lang))
   print(" ", messages["study-info"][lang].format(**args))
   print(" ", messages["score"][lang].format(**args))
rint("""Disclaimer: All the characters appearing in this work are fictitious.
ny resemblance to the real persons', living or dead, exam scores are purely coin
idental.""")
```

```
anguage: English (en)
 Benedict Timothy Carlton Cumberbatch studied Drama at The Universi
y of Manchester.
 Benedict Timothy Carlton Cumberbatch had 17 in the course "Program
ing with Python".
anguage: Cymraeg (cy)
 Benedict Timothy Carlton Cumberbatch Astudiodd Drama am The Univer
ity of Manchester.
 Benedict Timothy Carlton Cumberbatch wedi 17 yn y cwrs "Programmin
 with Python".
isclaimer: All the characters appearing in this work are fictitiou
.
ny resemblance to the real persons', living or dead, exam scores ar
 purely coincidental.
```

We can also align values on the screen:

In [26]:

```
 = """Gaudeamus igitur
uvenes dum sumus.
ost iucundam iuventutem
ost molestam senectutem
os habebit humus."""

rint("The original string:")
rint(x)

rint("\n|" + "".join(str(i%10) for i in range(1,11)) * 4 + "|")

rint("|{:<40}|".format("Centered in 40 characters:"))
or line in x.splitlines():
   print("|{:^40}|".format(line))

rint("|{:<40}|".format("Aligned to the right in 40 characters:"))
or line in x.splitlines():
   print("|{:>40}|".format(line))
```

```
The original string:
Gaudeamus igitur
Iuvenes dum sumus.
Post iucundam iuventutem
Post molestam senectutem
Nos habebit humus.

|1234567890123456789012345678901234567890|
|Centered in 40 characters:              |
|            Gaudeamus igitur            |
|           Iuvenes dum sumus.           |
|        Post iucundam iuventutem        |
|        Post molestam senectutem        |
|           Nos habebit humus.           |
|Aligned to the right in 40 characters:  |
|                         Gaudeamus igitur|
|                       Iuvenes dum sumus.|
|                Post iucundam iuventutem|
|                Post molestam senectutem|
|                      Nos habebit humus.|
```

Numbers can be formatted easily:

In [27]:

```
rom math import pi
rint("Pi printed without formatting:", pi)
rint("Pi to the 5th decimal: {:.5f}".format(pi))
rint("Pi in 17 characters, to the 5th decimal, prepended with spaces: {:17.5f}".
ormat(pi))
rint("Pi in 17 characters, to the 5th decimal, prepended with zeros: {:017.5f}".
ormat(pi))
rint("Pi in 17 characters, to the 5th decimal, with sign: {:+17.5f}".format(pi))
rint("Minus Pi in 11 characters, to the 5th decimal, with sign: {:+11.5f}".forma
(-pi))
```

```
i printed without formatting: 3.141592653589793
i to the 5th decimal: 3.14159
i in 17 characters, to the 5th decimal, prepended with spaces:
.14159
i in 17 characters, to the 5th decimal, prepended with zeros: 00000
00003.14159
i in 17 characters, to the 5th decimal, with sign:          +3.1415

inus Pi in 11 characters, to the 5th decimal, with sign:    -3.1415
```

Any names or positional indices are given before ":". For example, the list of the tallest (http://en.wikipedia.org/wiki/List_of_tallest_people) and the shortest (http://en.wikipedia.org/wiki/List_of_shortest_people) people in the World:

In [28]:

```
eople = [
   {
       "name": "Robert Wadlow",
       "height": 272,
       "status": "deceased",
   },
   {
       "name": "Sultan Kösen",
       "height": 251,
       "status": "alive",
   },
   {
       "name": "Chandra Bahadur Dangi",
       "height": 54.6,
       "status": "alive",
   }

rint("The tallest and the shortest people in the world:\n")
rint("{0:<21} | {1} | {2:^8}".format("Name", "Height (cm)", "Status"))
rint("-" * 22 + "+" + "-" * 13 + "+" + "-" * 9)
or person in people:
   print("{name:<21} | {height:11.1f} | {status:^8}".format(**person))
```

```
The tallest and the shortest people in the world:

Name                  | Height (cm) |  Status
----------------------+-------------+---------
Robert Wadlow         |       272.0 | deceased
Sultan Kösen          |       251.0 |  alive
Chandra Bahadur Dangi |        54.6 |  alive
```

There is much more that `format` can do. For the complete reference, read the official documentation (https://docs.python.org/3/library/string.html#formatstrings).

C-like formatting

C-like formatting is somewhat less powerfull, but easier to invoke, which makes it ideal for simple formattings (for example, of numbers). The basic syntaxt is:

```
    ormat_string % value
```

or, if there is more than one value to include,

```
    ormat_string % a_tuple_of_values
```

For example:

In [29]:

```
rom math import pi
rint("Pi printed without formatting:", pi)
rint("Pi to the 5th decimal: %.5f" % pi)
rint("Pi in 17 characters, to the 5th decimal, prepended with spaces: %17.5f" %
i)
rint("Pi in 17 characters, to the 5th decimal, prepended with zeros: %017.5f" %
i)
rint("Pi in 17 characters, to the 5th decimal, with sign: %+17.5f" % pi)
rint("Minus Pi in 11 characters, to the 5th decimal, with sign: %+11.5f" % pi)
 = 17
 = 19
rint("x = %d, y = %d, x + y = %d + %d = %d" % (x, y, x, y, x+y))
```

```
i printed without formatting: 3.141592653589793
i to the 5th decimal: 3.14159
i in 17 characters, to the 5th decimal, prepended with spaces:
.14159
i in 17 characters, to the 5th decimal, prepended with zeros: 00000
00003.14159
i in 17 characters, to the 5th decimal, with sign:          +3.1415

inus Pi in 11 characters, to the 5th decimal, with sign:    +3.1415

 = 17, y = 19, x + y = 17 + 19 = 36
```

More about C-like formatting can be read in the official reference (https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting).

Feel free to use it if you're already familiar with it. Otherwise, such a formatting is considered old and out of date, and it is advised to use the `format` function instead.

Generators and generator expressions

In this section we address some very useful, albeit Python-specific concepts:

- An *iterator* (https://docs.python.org/3/glossary.html#term-iterator) is an object representing a stream of data, i.e., capable of providing the first element, as well as traversing thorught all the elements from the beginning to the end.
  Note: Iterators also exist in C++, but it is a different concept and they have a different purpose there.
- A *generator* (https://docs.python.org/3/glossary.html#term-generator) is a function that returns an iterator.
- A *generator expression* (https://docs.python.org/3/glossary.html#term-generator-expression) is an expression that returns an iterator.

Iterators are a powerful thing to use, but quite complex in all their generality, and they go well beyond the scope of this course.

We shall cover generator expressions, as they are a very compact, efficient, and often used way to deal with list-like data, and then we will briefly see how generators work, as a bit more complex but quite powerful mechanism.

We have seen the function `range` before and we said that it

> *pretends* to return a list of numbers (it returns something more complex, we can consider it a list for now)

Let us see *what exactly* `range` returns:

In [30]:

```
ng = range(3,7)
st = list(rng)
rint("rng =", rng)
rint("type(rng) =", type(rng))
rint("Elements of rng:")
or x in rng:
   print("  %d" % x)
rint("\nlst =", lst)
rint("type(lst) =", type(lst))
rint("Elements of lst:")
or x in lst:
   print("  %d" % x)
```

```
ng = range(3, 7)
ype(rng) = <class 'range'>
lements of rng:
 3
 4
 5
 6

st = [3, 4, 5, 6]
ype(lst) = <class 'list'>
lements of lst:
 3
 4
 5
 6
```

As we can see:

- A range and a list are not the same (they print differently).
- They have different types.
- However, they "contain" the same elements.

So, while acting the same way as lists do in a `for` loop, range is actually a generator, which is a function that returns an iterator, which is what *rng* is.

The main difference between a list and an iterator is that a list actually contains all the data and allows us to access each of its elements.

An iterator, on the other hand, may contain the elements, but it almost never will. It can somehow get the first element, keep track of what the current element is, and "jump" to the next one. So, basically, it can provide everything a `for` loop needs, without any restrictions on where it gets its data.

For example, `range(3, 7)` will return an iterator that holds three values:

- starting value: 3,
- increment: 1 (default, since we didn't provide the third argument),
- end value: 7 (exclusive).

So, when we execute

```
for x in range(3, 7):
    print(x)
```

the following will happen:

1. `for` will "ask" `range` for its first value and obtain 3. It will then assign that 3 to `x` and execute its body (`print(x)`).
2. In the next step, `for` will "ask" `range` for its next value. Now, `range` remembers that its current value was 3, so it will compute the next one, and return 4, which will get assigned to `x` and the loop's body will once again be executed.
3. In the next step, `for` will "ask" `range` for its next value, and get 5.
4. In the next step, `for` will "ask" `range` for its next value, and get 6.
5. In the next step, `for` will "ask" `range` for its next value. However, `range` will see that the next value, 7, is too big, and it will gently explain `for` that it has no more values. `for` will then end, without further executions of its body.

The benefits

1. No extra memory is required, no matter how long the range is. For example, `range(10**9)` will not take memory for one billion integers.
2. There will be no time delay before the `for`-loop starts, which would otherwise be needed to create such a long list of numbers in the computers' memory.
3. It is not uncommon that the program stops before going through all the elements of a list or an iterator. Looping (either with loops or some other mechanisms) can be stopped prematurely. In these cases, an iterator will not even compute the unneeded values, potentially saving a lot of memory and processing time.

`range` here is just an example. Iterators can be used for all kinds of streaming data. For example, they are also used for reading files (which can easily be a few gigabytes big, which would put a real strain on the system if actually loaded at once).

Generators

So, how do we make our own generator (a function that returns an iterator)?

The key thing here is `yield`: a statement that acts like `return`, but it doesn't terminate a function (it merely "pauses" it).

Here is an example of a generator:

In [31]:

```
 ef is_prime(n):
    """Returns true if an integer `n` is a prime."""
    if n < 2:
        return False
    for p in range(2,n):
        if n % p == 0:
            return False
    return True

 ef primes(n):
    """Returns a generator for the prime factors of `n`."""
    n = abs(n)
    for p in range(2,n+1):
        # if n is divisable by p and p is prime
        if n % p == 0 and is_prime(p):
            yield p

  = int(input("n = "))
 nt = 0  # Counter
 or prime in primes(n):
    cnt += 1
    print("Prime factor #%d of %d: %d" % (cnt, n, prime))
 rint("Number %d has %d distinct prime factors." % (n, cnt))
```

```
n = 12345
Prime factor #1 of 12345: 3
Prime factor #2 of 12345: 5
Prime factor #3 of 12345: 823
Number 12345 has 3 distinct prime factors.
```

Note: This is a highly **inefficient** algorithm to find prime factors of a number. We shall see an improvement in several weeks. We use this one here because it closely follows the mathematical definition and, as such, it is easy to understand it.

So, how does the above work? Let us assume that $n = 84 = 2^2 \cdot 3 \cdot 7$.

When the `for`-loop is first encountered, `prime(84)` is called. It executes as any other function would, until it reaches `yield p`, which will first happen for `p = 2`. At this moment, the function returns `p` (i.e., 2) and stops executing, but **it is not terminated**, as it would have been if `return` was in place of `yield`.

So, the `for`-loop has now obtained its first value, 2, and it assigns it to `prime`. The body of the `for`-loop is now executed, i.e., `cnt` is increased from zero to one, and the message

> Prime factor #1 of 84: 2

is printed.

After the body is executed, we return to the head of the `for`-loop. Recall that the function was not terminated, so it continues where it stopped when a `yield` statement was last invoked. The next prime factor that the `for`-loop in `primes` function finds is `p = 3`. Again `yield p` is invoked, the main `for`-loop gets its second value, `prime = 3` is assigned, and the message

> Prime factor #2 of 84: 3

is printed.

Back to the header of the `for`-loop, we reenter the still uninterrupted function `primes`, and continue after its last executed `yield` (generally, a generator can have many `yield` statements, not just one as in our example). The next `p` it will find is 7, which is again returned to the main `for`-loop and

> Prime factor #3 of 84: 7

is printed.

We return to the header of the `for`-loop once again, reentering `primes`. However, this time, no new value of `p` will be found, and the function's own `for`-loop will end. Since there is no code after it, the `primes` function will finally come to an end (i.e., it will stop and not just "pause"). Now, the main `for`-loop is notified that there are no further values to be assigned to the `prime` variable, so it too quits.

The last command in our program is now executed, printing:

> Number 84 has 3 distinct prime factors.

Iterators are easily converted to lists:

In [32]:

```
  = int(input("n = "))
rint("The list of the prime factors of %d:" % n, list(primes(n)))
```

```
n = 12345
The list of the prime factors of 12345: [3, 5, 823]
```

Generators are a fairly advanced concept and they will not be part of any examinations. However, the use of generators is encouraged. You can find more about them here (http://www.jeffknupp.com/blog/2013/04/07/improve-your-python-yield-and-generators-explained/).

Generator expressions

Generator expressions are expressions that return iterators. They behave in the same manner as generators, but their syntax is more compact and usually easier to understand as it is very similar to `for` loops.

Here is a simple example of a generator of the squares of the numbers from 0 to `n-1`:

In [33]:

```
  = int(input("n = "))
en = (x*x for x in range(n))
rint("gen =", gen)
rint("type(gen) =", type(gen))
rint("list(gen) =", list(gen))
```

```
n = 17
gen = <generator object <genexpr> at 0x7f536a78f0f0>
type(gen) = <class 'generator'>
list(gen) = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169,
196, 225, 256]
```

Notice that the syntax of a generator is very much like what one would write in a mathematical document:

> gen contains values of x^2 for all integer x in $[0, n\rangle$.

Even if we want to add some filtering condition, it is done in the same manner. For example,

> gen contains values of x^2 for all integer x in $[0, n\rangle$ for which x is a prime number.

is written like this:

In [34]:

```
  = int(input("n = "))
en = (x*x for x in range(n) if is_prime(x))
rint("gen =", gen)
rint("type(gen) =", type(gen))
rint("list(gen) =", list(gen))
```

```
n = 17
gen = <generator object <genexpr> at 0x7f536a78fd20>
type(gen) = <class 'generator'>
list(gen) = [4, 9, 25, 49, 121, 169]
```

As we can see, generator expressions look like neatly packed `for`-loops, which makes them rarely useful in `for`-loops themselves. For example,

In [35]:

```
en = (x*x for x in range(n) if is_prime(x))
or y in gen:
   print(y)
```

```
4
9
25
49
121
169
```

or, without the auxiliary variable `gen`,

In [36]:

```
or y in (x*x for x in range(n) if is_prime(x)):
   print(y)
```

```
4
9
25
49
121
169
```

is better written as:

In [37]:

```
or x in range(n):
   if is_prime(x):
       print(x*x)
```

```
5
9
21
69
```

However, there are various other uses. The first of them is creating a list of easily computable values (complex operations don't fit in generator expressions):

In [38]:

```
 The traditional way:
st = list()
or x in range(n):
   if is_prime(x):
       lst.append(x*x)
rint(lst)

 Using a list comprehension
st = [x*x for x in range(n) if is_prime(x)]
rint(lst)
```

```
[4, 9, 25, 49, 121, 169]
[4, 9, 25, 49, 121, 169]
```

A generator expression that creates a list is often called a *list comprehension*.

Additionally, there are various functions that take iterators as arguments. One of them is `sum`, which returns the sum of all the elements:

In [39]:

```
 The traditional way:
um_ = 0
or x in range(n):
   if is_prime(x):
       sum_ += x*x
rint(sum_)

 Using a generator
rint(sum(x*x for x in range(n) if is_prime(x)))
```

```
377
377
```

In these two examples, we merely wrote less code, but there was no gain in memory consumption nor the processing speed.

However, let us check if **any** of the above squares has the form `p+2` for some prime number `p`. In other words, we want to check if `x*x-2` is a prime for some `x` which is a prime itself:

In [40]:

```
umbers = (x*x for x in range(n) if is_prime(x))
f any(is_prime(number-2) for number in numbers):
   print("Yes")
lse:
   print("No")
```

```
es
```

In the above example, `is_prime(number-2)` gets computed until the first `number` for which it is true is encountered. This is 4, which happens for `x = 2` when generating `numbers`:

In [41]:

```
umbers = (x*x for x in range(n) if is_prime(x))
rint("Yes:", list(number for number in numbers if is_prime(number-2)))
umbers = (x*x for x in range(n) if is_prime(x))
rint("No: ", list(number for number in numbers if not is_prime(number-2)))
```

```
es: [4, 9, 25, 49, 169]
o:  [121]
```

Obviously, not all numbers satisfy this new condition (121 does not, because $121 - 2 = 119 = 7 \cdot 17$).

If we wanted to just check if **all** of the `numbers` have the form `p+2` for some prime number `p` (i.e., if all `x*x-2` are primes for some `x` which is a prime itself), we would have used the function `all`:

In [42]:

```
umbers = (x*x for x in range(n) if is_prime(x))
f all(is_prime(number-2) for number in numbers):
   print("Yes")
lse:
   print("No")
```

```
No
```

Notice that the following wouldn't work properly:

In [43]:

```
 = 17

 Try one:
umbers = (x*x for x in range(n) if is_prime(x))
rint("Yes:", [number for number in numbers if is_prime(number-2)])
rint("No: ", [number for number in numbers if not is_prime(number-2)])

 Try two:
umbers = (x*x for x in range(n) if is_prime(x))
rint("No: ", [number for number in numbers if not is_prime(number-2)])
rint("Yes:", [number for number in numbers if is_prime(number-2)])
```

```
es: [4, 9, 25, 49, 169]
o:  []
o:  [121]
es: []
```

Generator expressions are meant for a single use. They know how to do "next", but there is no way to "reset" them (in the above example, the problem is that `numbers` is not reset). Depending on the exact generator that is used, there are various ways to deal with this, but this goes beyond the scope of this course.

More than one `for` clause

Generators and list comprehensions can have more than one `for` clause.

For example, here we create a list of all fully simplified fractions (memorized as strings) between zero and one (exclusively) with numerator and denominator between `1` and `m`.

In [44]:

```
 = 5

 List comprehension
racts = ["%d/%d" % (n, d)
         for n in range(1, m+1)
         for d in range(n, m+1)
         if all(n % k != 0 or d % k != 0 for k in range(2, n+1))
        ]
rint(fracts)

 Equivalent loops
racts = list()
or n in range(1, m+1):
   for d in range(n, m+1):
       if all(n % k != 0 or d % k != 0 for k in range(2, n+1)):
           fracts.append("%d/%d" % (n, d))
rint(fracts)
```

```
'1/1', '1/2', '1/3', '1/4', '1/5', '2/3', '2/5', '3/4', '3/5', '4/
']
'1/1', '1/2', '1/3', '1/4', '1/5', '2/3', '2/5', '3/4', '3/5', '4/
']
```

Of course, there are more efficient ways to check if a fraction is fully simplified (hint: Euclidean algorithm), but we shall cover that later.

As always, all of these `for` statements could've looped over iterables other than iterators created by `range`. We could use lists, tuples, sets, etc.

For example, to limit numerators to the set $\{1, 3, 5, 7, 11\}$ and denominators to the set $\{17, 19, 23\}$ we can do this:

In [45]:

```
 = 5

 List comprehension
racts = ["%d/%d" % (n, d)
         for n in [1,3,5,7,11]
         for d in (17,19,23)
         if all(n % k != 0 or d % k != 0 for k in range(2, n+1))
        ]
rint(fracts)

 Equivalent loops
racts = list()
or n in [1,3,5,7,11]:
   for d in (17,19,23):
       if all(n % k != 0 or d % k != 0 for k in range(2, n+1)):
           fracts.append("%d/%d" % (n, d))
rint(fracts)
```

```
'1/17', '1/19', '1/23', '3/17', '3/19', '3/23', '5/17', '5/19', '5/
3', '7/17', '7/19', '7/23', '11/17', '11/19', '11/23']
'1/17', '1/19', '1/23', '3/17', '3/19', '3/23', '5/17', '5/19', '5/
3', '7/17', '7/19', '7/23', '11/17', '11/19', '11/23']
```

Examples

We are now ready to make some of the old pieces of code simpler.

In [46]:

```
 = int(input("n = "))
rint("a) Sum of digits of %d:" % n, sum(int(x) for x in str(n)))
rint("b) Prime digits of %d:" % n, [int(x) for x in str(n) if x in "2357"])
rint("c) Sum of prime digits of %d:" % n, sum(int(x) for x in str(n) if x in "23
7"))
rint("d) Prime factors of %d:" % n, [x for x in range(2,abs(n)) if n % x == 0 an
 is_prime(x)])
```

```
n = 1234567
a) Sum of digits of 1234567: 28
b) Prime digits of 1234567: [2, 3, 5, 7]
c) Sum of prime digits of 1234567: 17
d) Prime factors of 1234567: [127, 9721]
```

Notes:

1. Generator expressions don't always speed things up.
   For example, the above code in d) is still very slow, as it tests all the numbers between 2 and n-1.
2. Repeating the same generator multiple times, like we did in b) and c), can actually make things slower (computing the same thing twice) and the code redundant (we copied the same code twice).
   To overcome these disadvantages, we can:

   - create a list of items (using a single generator) if speed is more of an issue than memory, or
   - clone the generator in some cases (read about itertools.tee (https://docs.python.org/3/library/itertools.html#itertools.tee)), or a function that returns an iterator in other cases, or use a specific function for resetting that some iterators have (for example, those that read files).