Algorithms and Data Structures

Python

Giacomo Fiumara
giacomo.fiumara@unime.it

2023-2024


python™
Donate
Search
GO
Socialize
About
Downloads
Documentation
Community
Success Stories
News
Events
Python is powerful... and fast;
plays well with others;
runs everywhere;
is friendly & easy to learn;
is Open.
These are some of the reasons people who use Python
would rather not use anything else.

Active Python Releases

For more information visit the Python Developer's Guide.

Python version	Maintenance status	First released	End of support	Release schedule
3.11	bugfix	2022-10-24	2027-10	PEP 664
3.10	bugfix	2021-10-04	2026-10	PEP 619
3.9	security	2020-10-05	2025-10	PEP 596
3.8	security	2019-10-14	2024-10	PEP 569
3.7	security	2018-06-27	2023-06-27	PEP 537

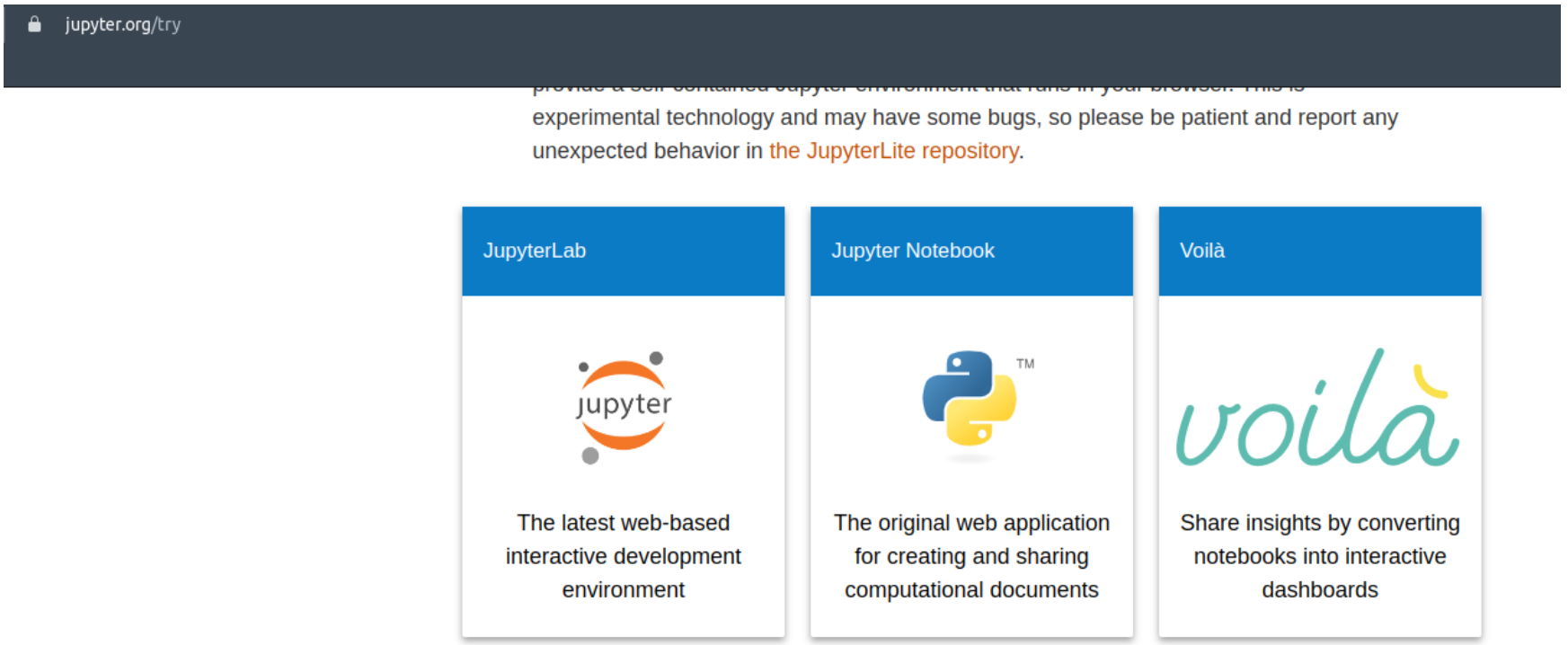
Anaconda

Individual Edition is now

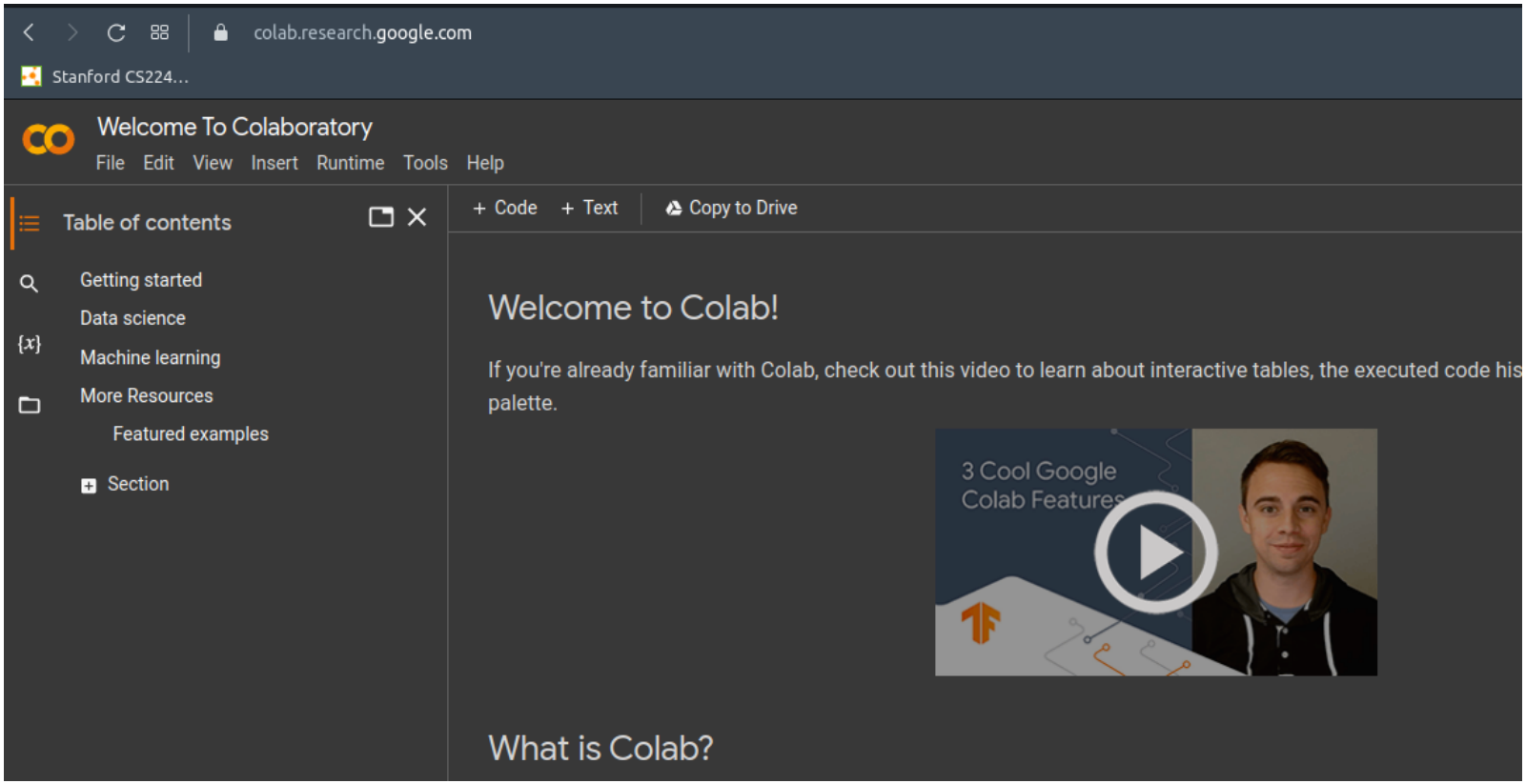
ANACONDA DISTRIBUTION

The world's most popular open-source Python distribution platform

Jupyter (online)



Google Colab



Variables, Expressions and Instructions

Assignments /1

- When a program carries out computations, values must be stored so to use them later. In Python programs **variables** are used to store values. A variable has a name and holds a value
- An **assignment instruction** allows to create a new variable, specifying its name and assigning a value

```
msg = 'Hello world!!!'
pippo = 4.0
pluto = 12
```

Variables, Expressions and Instructions

Assignments /2

- The names of variables can be as long as needed, and can be composed of letters and/or numbers. They cannot begin with a number
- It is possible to use capital letters, but according to a convention only regular letters are used
- Python interpreter is *case sensitive*
- Special characters cannot be employed (with the notable exception of *underscore*)

The following words are **reserved** and cannot be used as names of variables

False	None	True	and	as	assert
break	class	continue	def	del	elif
else	except	finally	for	from	global
if	import	in	is	lambda	nonlocal
not	or	pass	raise	return	try
while	with	yield			

Variables, Expressions and Instructions

Expressions

- An expression is a combination of values, variables and operators
- A value and an expression are two (small) examples of expressions
- When an expression is typed on the command prompt, the interpreter evaluated it and returns its value
- An instruction is a portion of code which the Python interpppreter can execute and produces some effect, such as creating a variable or returning a value
- When an instruction has been typed, the interpreter executes it

Variables, Expressions and Instructions

```
n = 12.3
print(n)
12.3
n
12.3
```

- If an expression contains some operators, the precedence of operators follows the rules used in mathematics
  - Parentheses
  - Powers
  - Multiplication
  - Division
  - Addition
  - Subtraction

For example:

$$\frac{360}{2\pi}$$

```
gradi = 360
pi = 3.14159
print(gradi / 2 * pi)
565.4861999999999
print(gradi / 2 / pi)
57.295827908797776
```

- Python provides two types of comments (text ignored by the interpreter)
- A comment is an explanation for human readers of a code
- Comments are opened by the # symbol and can extend to the end of the current line
- Multi line comments must be enclosed by triple quotes

Variables, Expressions and Instructions

Comments

```
#comment

n = 4 # setting n to 4
n
4
'''Multi line comment enclosed by triple (single)
    quotes
'''
'Multi line comment enclosed by triple quotes'

"""Multi line comment enclosed by triple (double)
    quotes
 """
'Multi line comment enclosed by triple (double) quotes'
```

A **function** is a series of instructions (to carry out a calculation) to which a name is associated

- Python provides a collection of functions that convert values from a type to another
- For example, function `int` converts, if possible, a given value to an integer. If the conversion cannot take place, a message error will be issued

Functions

Function call

```
int('32')
32
int(12.445)
12
int('c')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'c'

int(3.999)
3
```

- The string '32' is converted into an integer
- Function **int** can convert floating-point values into integers, but the decimal part is truncated (not rounded)

Functions

Function call

- Function **`float`** converts the argument into a floating point number

```
float(32)
32.0
float('123')
123.0
```

- Function **`str`** converts the argument into a string

```
str(32)
'32'
float('123.45')
'123.45'
```

Functions

Mathematical functions

- Python provides a mathematical module containing most common mathematical operations
- A **module** is a file containing a collection of correlated functions
- A module must be imported prior to its use (just once in a program)
- Functions (and variables) contained in a module must be invoked using the so-called **dot notation**

```
import math
math.cos(math.pi)
-1.0
```

- The **function definition** specifies the name of a new function and the sequence of instructions to be executed upon function calling

```
def printing():
    print(''First message'')
    print(''Second message'')
```

- `def` is a reserved word used to define a new function
- Function naming follows the same rules as variable naming

```
def printing():
    print(''First message'')
    print(''Second message'')
```

- Empty parentheses after the function name specify that (in this case) the function does not require any parameter
- The first row of a function is called **heading**, the remaining part is the **body**. The heading needs a trailing colon
- Strings contained in the `print` instructions are enclosed in quotes (single or double)
- Instructions contained in a function must be indented (4 spaces farther to the right)

- The definition of a new function produces a **function object**

```
print(printing())
first message
second message
None
print(printing)
<function printing at 0x7f091779ce18>
```

- Note the difference between the two calls: in the second case only some information about the object **`printing`** are requested

- Similar information can be obtained using the `type` built-in function

```
print(printing())
first message
second message
None
type(printing)
<class 'function'>
```

Functions

Execution flow

- The **execution flow** of a program is the order according to which instructions are executed
- Defining a function **does not** change the execution flow
- It is important to observe, however, that the instructions contained in a function are not executed unless the function is called
- A function call does represent a deviation in the execution flow as the next instruction will be the first instruction contained in the function

Functions

Parameters and arguments

- Some functions need arguments: for example, the function **math.pow** needs the base and the exponent

```
import math
print(math.pow(3,2))
```

- Within the function, arguments are assigned to variables called **parameters**

```
def printing2(argument1, argument2):
    print(argument1)
    print(argument2)

printing2('hello', 'everybody')
hello
everybody
```

Functions

Parameters and arguments

- Variables created in a function are **local** as they exist **only** within the function
- This means that these variables cannot be accessed/updated from outside the function

```
def printing3(argument1, argument2):
    stringx = argument1 + argument2
    print(stringx)

printing3('hello', ' ciao')
hello ciao
print(stringx)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'stringx' is not defined
```

- The local variable **stringx** will be destroyed when the function completes its execution. Therefore using it causes an error

```
print(stringx)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'stringx' is not defined
```

- Some functions, for example **`math.pi`**, yield results. They are called **fruitful functions**
- Other functions, for example **`printing3`**, perform an action but do not return a value. They are called **void functions**
- A fruitful function is called because its return value can be used for further operations:

```
x = math.cos(angle)
```

- A fruitful function such as

```
math.sqrt(4)
2.0
```

- should not be used this way in a program, because its return value will be lost

```
math.sqrt(4)
```

```
def printing():
    print('1st message')
    print('2nd message')

result = printing()
1st message
2nd message
print(result)
None
```

- Note that variable **result** does not contain any value (as expressed by **None**)

- The **modulus operator** % works on integers and yields the remainder when the first operand is divided by the second

```
>>> minutes = 105
>>> minutes / 60
1.75
>>>
>>> minutes = 105
>>> hours = minutes // 60
>>> hours
1
```

Conditionals and recursion

Modulus operator

- To obtain the remainder, it is possible to subtract from minutes the equivalent in hours

```
>>> remainder = minutes - hours * 60
>>> remainder
45
```

- The modulus operator allows to solve this problem in a more concise way

```
>>> remainder = minutes % 60
>>> remainder
45
```

Conditionals and recursion

Modulus operator

- The modulus operator is useful in a number of situations
- It is possible to check the divisibility of a number by another
- It is also possible, for example, to use it to extract the figures composing a number

```
>>> x % 3
>>> x % 10
>>> x % 100
```

- A **boolean expression** can be either true or false

```
>>> 5 == 5
True
>>> 5 == 6
False

>>> 5 = 5
  File "<stdin>", line 1
SyntaxError: can't assign to literal
>>> 5 == 6
False
>>>
```

- **`True`** and **`False`** are special values of the type **`bool`**, they are not strings

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
>>>
```

Other relational operators are:

x != y	x is not equal to y
x > y	x is greater than y
x < y	x is smaller than y
x >= y	x is greater than or equal to y
x <= y	x is smaller than or eqaul to y

Logical operators that can be used in conditionals are **`and`**, **`or`** and **`not`**

- The expression **`x > 0 and x < 10`** is true if and only if both conditions are true
- The expression **`n//2==0 or n//3==0`** is true if and only if at least one of the two conditions is true
- The operator **`not`** negates the value of a boolean expression. For example, **`not x>y`** is true if **`x>y`** is false

Conditionals and recursion

Logical operators

- In Python every nonzero number is regarded as **`True`** while zero is regarded as **`False`**

```
>>> 4 and True
True
```

- In an analogous way, every data structure (list, set, dictionary, string) is regarded as true if it is non empty

```
>>> stringx = ''
>>> if(stringx):
...     print('Non empty')
...
>>>
```

- Conditionals allow to evaluate conditions and to modify the behaviour of a program depending on the result of the evaluation

```
>>> x = 3
>>> if x > 0:
...     print('x positive')
...
x positive
```

- The instruction `if` is composed with an header and a body (indented with respect to the header) which contains the instructions to be executed if and only if the header is true

- There is no limit to the number of instructions that can be contained in the body of an `if` instruction. The bare minimum is one instruction (even a fictitious one is admitted)

```
if x < 0:
  pass
```

Conditionals and recursion

Alternatives

- The **if** instruction allows an alternative execution, meaning that there are two possible actions
- The value of the condition determines which of the two alternatives (branches) will be executed

```
if x % 2 == 0:
  print(''x even'')
else:
  print(''x odd'')
```

Conditionals and recursion

Series of conditionals

- In some cases more branches may be needed
- In these situations the **elif** clause can be used

```
if x < y:
  print(''x less than y'')
elif x > y:
  print(''x greater than y'')
else:
  print(''x equal to y'')
```

- Any number of **elif** clauses can be used. The **else** clause, which is not mandatory, must be the last
- The interpreter will control the conditions according to their appearance order. The first true condition will be executed and other possible true conditions will be ignored

Conditionals and recursion

Nested conditions

- Conditionals can be nested (awful programming style)

```
if x == y:
  print('' x is equal to y'')
else:
  if x < y:
    print(''x less than y'')
  else:
    print(''x greater than y'')
```

Conditionals and recursion

Nested conditions

- Logical operators allow to avoid nested conditions

```
if x > 0:
  if x < 10:
    print(''x positive, one figure'')
```

- This fragment can be rewritten as:

```
if x > 0 and x < 10:
   print(''x positive, one figure'')
```

Conditionals and recursion

Recursion

- Python functions can invoke themselves

```
def countdown(n):
  if n <= 0:
    print(''Go!!!'')
  else:
    print(n)
    countdown(n-1)
```

Conditionals and recursion

Infinite recursion

- If a recursive function never reaches the base case, the function call will be executed an infinite number of times
- In principle the program never ends
- This situations is called **infinite recursion**

```
def infinite_recursion():
  print("Here we are")
  infinite_recursion()
. . .
  File ``prova04.py'', line 2, in ricorsione\_infinita
    ricorsione\_infinita()
RecursionError: maximum recursion depth exceeded
```

- Python provides a built-in function called **`input`** that gets input from the keyboard
- When this function is called, the program stops and waits for the user to input something
- When the user presses `Enter` or `Return`, the program resumes

```
x1 = input()
print(x1)
x2 = input('Enter an integer number\n')
print(x2)
```

Conditionals and recursion

Keyboard input

```
datum = input('Enter an integer number\n')
str = datum + 'xxx'
print(str)
print(datum + 2)
```

- The value provided by the user is regarded as a string

```
Enter an integer number
4
4xxx
Traceback (most recent call last):
  File ``Examples/prova06.py'', line 4, in <module>
    print(datum + 2)
TypeError: Can't convert 'int' object to str implicitly
```

Conditionals and recursion

Recursion: factorial of a number

```
def factorial(n):
  if n== 0:
    return 1
  else:
    x = factorial(n-1)
    result  = n * x
    return result
```

Fruitful functions

Recursion

```
def factorial(n):
    spaces = ' ' * (4 * n)
    print(spaces, 'factorial', n)
    if n == 0:
        print(spaces, 'return 1')
        return 1
    else:
        ricors = factorial(n-1)
        result = n * ricors
        print(spaces, 'return ', result)
    return result
```

Fruitful functions

Return values

- **Fruitful functions** return some value

```
def area(radius):
  a = math.pi * radius**2
  return a
```

- Or, equivalently:

```
def area(radius):
  return math.pi * radius**2
```

Functions produttive

Return values

- Sometimes it is useful to have multiple return statements, one in each branch of a conditional:

```
def asbolute_value(x):
  if x < 0:
    return -x
  else:
    return x
```

- This function is wrong, because if x happens to be zero, neither condition is true and the function ends with returning any value

Fruitful functions

Boolean functions

- Functions may also return boolean values

```
def divisible(x, y):
  if x % y == 0:
    return True
  else:
    return False
```

- Or equivalently:

```
def divisible(x, y):
  return x % y == 0
```

Fruitful functions

Boolean functions: review 1

- Write the function **`between(x,y,z)`** which returns **`True`** if $x \leq y \leq z$, **`False`** otherwise

Fruitful functions

Boolean functions: review

- Write the function **`between(x,y,z)`** which returns **`True`** if $x \leq y \leq z$, **`False`** otherwise

```
def between(x,y,z):
    return x <= y <= z
```

Fruitful functions

Checking types

- If the **factorial** is invoked with a non integer argument, say 2.5, the result will be:

```
File "Examples/prova12bis.py", line 5, in
    factorial
     recursion = factorial(n-1)
  File "Examples/prova12bis.py", line 2, in
      factorial
     if n== 0:
RecursionError: maximum recursion depth exceeded
    in comparison
```

- If it happens, the case $n = 0$ never takes place and the recursion never stops (in principle)
- A possible solution consists in checking the type of argument passed to the function
- To this purpose the built-in function **`isinstance`** can be used

```
isinstance(object, classinfo)
```

- which returns **`True`** if **`object`** is a **`classinfo`** instance

Fruitful functions

Checking types

```
def factorial(n):
    if not isinstance(n, int):
        print("Enter an integer number")
        return None
    elif n < 0:
        print("Enter a positive integer number")
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)

n = int(input("Enter an integer number: "))
print(factorial(n))
```

A number a is power of b if it is divisible by b and if a/b is a power of b. Write a function named **`power(a,b)`** which requires two arguments (**`a`** and **`b`**) and returns **`True`** if **`a`** is a power of **`b`**.

Fruitful functions

Review 3

Write the recursive function **`product(n)`** which requires an integer number **`n`** as argument and returns its product with all natural numbers that precede it. For example, **`product(4)`** must return $24 = 4 \cdot 3 \cdot 2 \cdot 1$

Loops

The `while` loop

The `while` statement allows to implement a loop. The execution flow is as follows:

1. Determine whether the condition is true or false
2. If false, exit the `while` statement and continue execution at the next statement
3. If the condition is true, run the body and then go back to step 1

Loops

The `while` loop

```
def countdown(n):
    while n > 0:
        print(n)
        n -= 1
    print("Via!!!")
```

Loops

The `while` loop

- The body of the loop should change the value of one or more variables so that the condition becomes false eventually and the loop terminates
- Otherwise the loop will repeat forever, which is called **infinite loop**
- It is necessary to check in advance if the loop will end

Loops

The `while` loop

```
def sequence(n):
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = n * 3 + 1
    print(n)
```

Loops

The `while` loop: break

The **`break`** statement allows to terminate a loop without having to wait for the iterations to complete

```
while True:
    line = input("Enter a string: ")
    if line == 'done':
        break
    print(line)
print("Done!")
```

It is possible to approximate the value of π using the Gregory-Leibniz series:

$$\pi = 4 \sum_{i}^{\cdot} \frac{(-1)^i}{2i+1}$$

Implement the **greekpi** function that computes π with an error 1/1000 smaller than the "true" value provided by **math.pi**

Loops

The `while` loop: review

It is possible to approximate the value of π using the Gregory-Leibniz series:

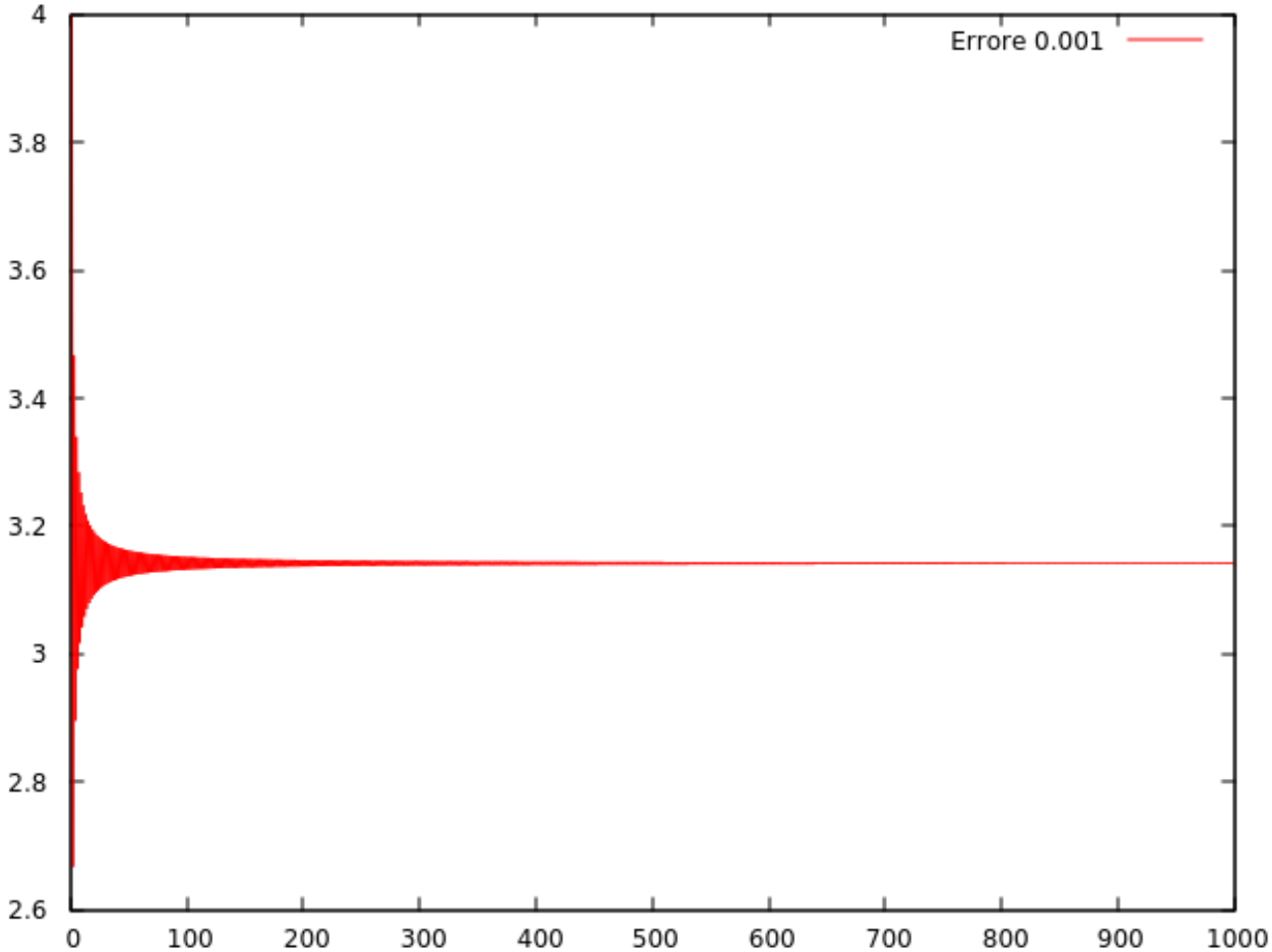
$$\pi = 4 \sum_{i}^{\cdot} \frac{(-1)^i}{2i+1}$$

Implement the **`greekpi`** function that computes π with an error 1/1000 smaller than the "true" value provided by **`math.pi`**

```
import math
def greekpi():
    gpi, i = 0, 0
    while abs(gpi - math.pi) > 0.0001:
        gpi += 4 * math.pow(-1,i)/(2 * i + 1)
        print(i, gpi, abs(gpi - math.pi))
        i += 1
```

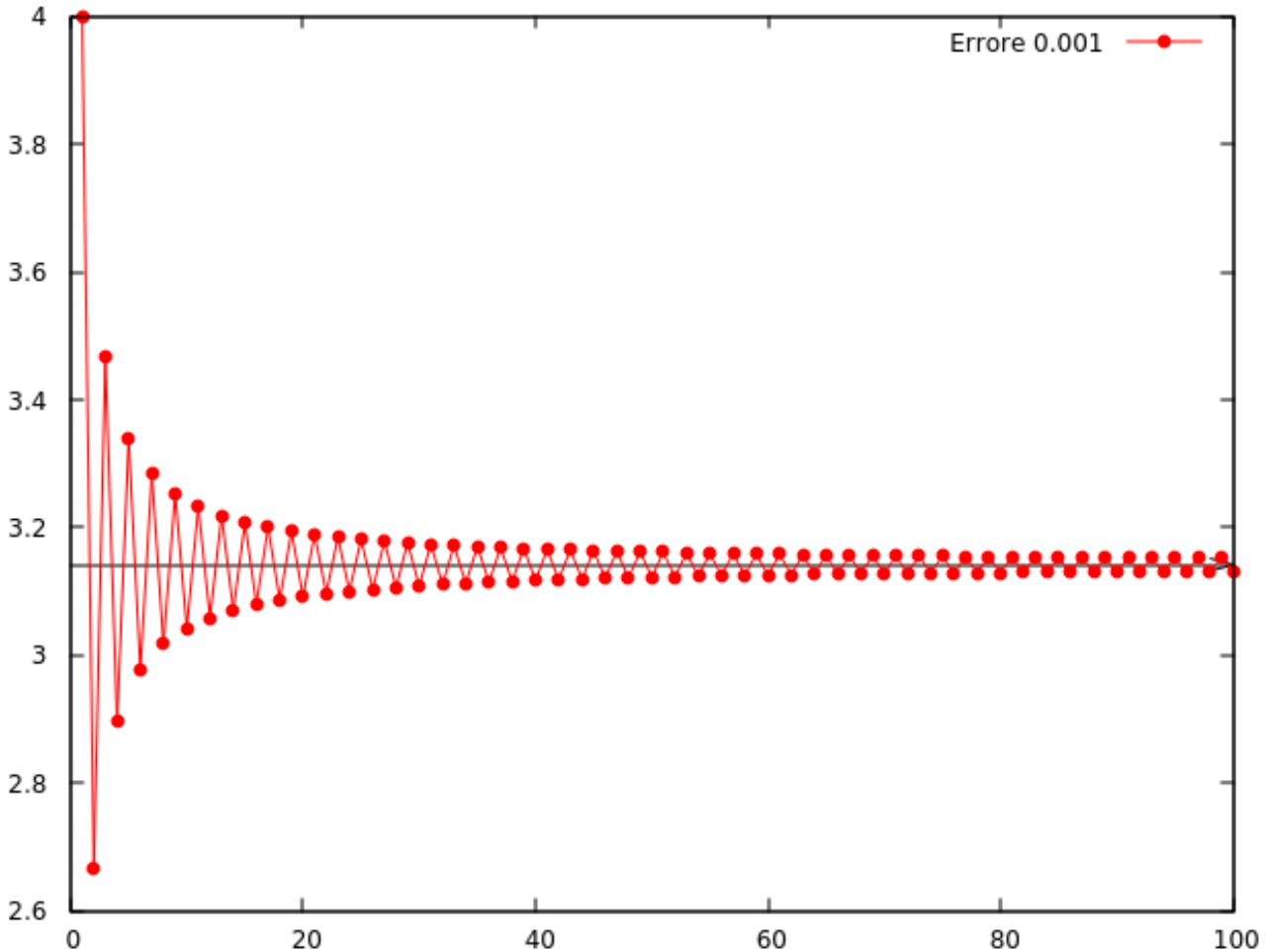
Loops

The `while` loop: review



Loops

The `while` loop: review



Strings

- A string is a sequence of characters
- It is possible to access the characters one by one with the bracket operator:

```
str1   = ``Hello everybody''
char01 = str1[4]
print(char01)
```

- The expression in brackets is called an **index**
- The index indicates which character in the sequence is requested

Strings

Function `len`

- **`len`** is a built-in function intended to return the number of characters in a string
- More generally, **`len`** returns the number of elements of a sequence data type
- If a sequence is not empty, it is possible to access to each element with the bracket operator

```
>>> str1   = ``hello everybody''
>>> len(str1)
12
>>>
>>> str1[len(str1)]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>>
>>> str1[len(str1)-1]
'y'
```

Strings

`for` loop

- It is possible to process a string one character at a time
- Usually this processing starts from the beginning of the string, each character is selected in turn, something is done to it and the process continues until the ened (**traversal**)

```
stringx = 'I wandered lonely as a cloud...'
i = 0
while i < len(stringx):
    letter = stringx[i]
    print(letter)
    i += 1
```

Strings

`for` loop

```
stringx = 'I wandered lonely as a cloud...'
for letter in stringx:
    print(letter)
```

Strings

Slicing

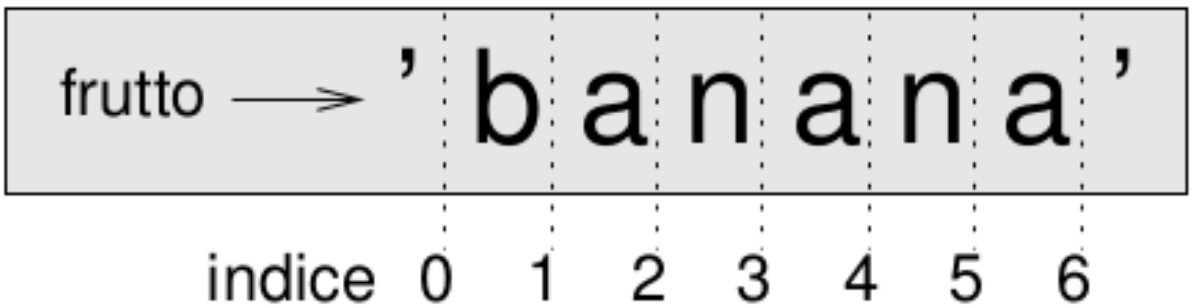
- A segment of a string is called a **slice**

```
>>> stringx = 'I wandered lonely as a cloud...'
>>> print(stringx[0:4])
'I wa'
```

Strings

Slicing

- The operator **`[n:m]`** returns the part of the string from the *n*-th character to the *m*-th character, including the first but excluding the last
- It might help to imagine the indices pointing *between* the characters



- If the first index is omitted, the slice starts at the beginning of the string
- If the second index is omitted, the slice goes to the end of the string
- If the first index is greater than or equal to the second the result is an **empty string**

Strings

Slicing

```
>>> stringx = 'I wandered lonely as a cloud...'
>>> stringx[:6]
'I wan'
>>> stringx[6:6]
''
>>> stringx[2:]
'wandered lonely as a cloud...'
>>> stringx[4:1]
''
>>> stringx[:]
'I wandered lonely as a cloud...'
>>>
```

Strings

Strings are immutable

```
>>> stringx[0] = 'i'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item
    assignment
>>>
```

- Strings are **immutable**
- This means that the operator `[]` cannot be used on the left side of an assignment

- A solution to this issue is:

```
>>> stringx = 'I wandered lonely as a cloud...'
>>> stringx = 'i' + stringx[1:]
>>> stringx
'i wandered lonely as a cloud...'
```

- Actually the original variable **stringx** has been replaced by another variable bearing the same name

Strings

Strings are immutable

- This can be shown this way

```
>>> stringx = 'I wandered lonely as a cloud...'
>>> id(stringx)
139892565942400
>>> stringx = 'i' + stringx[1:]
>>> print(stringx)
i wandered lonely as a cloud...
>>> id(stringx)
139892564734160
```

```
def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            return index
        index = index + 1
    return -1
```

```
def count(text, letter):
    count = 0
    for char in text:
        if char == letter:
            count = count + 1
    return count
```

Strings

Methods

capitalize	casefold	center	count	encode
endswith	expandtabs	find	format	format_map
index	isalnum	isalpha	isdecimal	isdigit
isidentifier	islower	isnumeric	isprintable	isspace
istitle	isupper	join	ljust	lower
lower	lstrip	maketrans	partition	replace
rfind	rindex	rjust	rpartition	rsplit
rstrip	rsplitlines	startwith	swapcase	title
translate	upper	zfill		

```
>>> stringx = 'Hello everybody'
>>> stringx.capitalize()
'Hello everybody'
>>> stringx.center(40)
'             Hello everybody              '
>>> stringx.center(40,'-')
'-------------Hello everybody-------------'
>>> stringx.count('e')
3
```

```
>>> 'goofy.exe'.endswith('exe')
True
>>> stringx.find('ry')
9
>>> stringx.find('yr')
-1
>>> stringx.ljust(40)
'Hello everybody                         '
>>> stringx.rjust(40)
'                         Hello everybody'
>>> stringx.replace('y','Y')
'Hello everYbodY'
```

Strings

The `in` operator

- The word **`in`** is a boolean operator that takes two strings and returns **`True`** if the first appears as a substring in the second:

```
>>> 'y' in 'Hello everybody'
True
>>> 'h' in 'Hello everybody'
False
```

Strings

The `in` operator

```
>>> stringx = 'ciao a tutti'
>>> for letter in stringx:
...     print(letterx)
...
c
i
a
o

a

t
u
t
t
i
>>>
```

Strings

The `in` operator

```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print(letter)
```

Strings

String comparison

- The relational operators work on strings
- Uppercase and lowercase letters are not interchangeable, and uppercase letters come first

---

```
>>> a='ciao'
>>> b = 'CIAO'
>>> a == b
False
>>> a > b
True
>>> b = 'Ciao'
>>> a > b
True
```

Write a program that reads a number between 1,000 and 999,999 from the user and prints it with a comma separating the thousands.

Lists

- Like a string, a list is a sequence of values
- In a list, the values can be any type
- The simplest way to create a new list is to enclose the elements in square brackets

---

```
>>> la = [1,2,3,4,5]
>>> lb = ['hello', 'everybody. ', 'How', 'are you
    ?']
>>> lc = [1,2,3,'hello everybody. How are you?',
    20, 30.24, True]
>>> lc
[1, 2, 3, 'hello everybody. How are you?', 20,
    30.24, True]
```

Lists

- A list that contains no elements is called an empty list, which can be created using empty brackets **[]**
- A list can contain another list. A list within another list is called **nested**

```
>>> ld = [lc, la]
>>> ld
[[1, 2, 3, 'hello everybody. How are you?', 20,
   30.24, True], [1, 2, 3, 4, 5]]
>>> le = []
>>> le
[]
```

Lists

Lists are mutable

- The syntax for accessing the elements of a list is the same for accessing the characters of a string, the bracket operator
- Unlike strings, lists are mutable

```
>>> lb = ['hello', 'everybody. ', 'How', 'are you
    ?']
>>> id(lb)
140451531110408
>>> lb[2]
'How'
>>> lb[2] = 'how'
>>> lb
['hello', 'everybody. ', 'how', 'are you?']
>>> id(lb)
140451531110408
```

Lists

Lists are mutable

- Lists indices can be any integer expression
- If an index is negative counts backward from the end of the list

```
>>> lb = ['hello.', 'How', 'are you?']
>>> lb[-1]
'are you?'
>>> i=0
>>> lb[i+1]
'How'
```

Lists

Lists are mutable

- Any access to non existing elements will produce an **IndexError**
- The **in** operator can be applied to lists too

```
>>> lb = ['hello.', 'How', 'are you?']
>>> lb[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

>>> 'a' in lb
False
>>> 'How' in lb
True
```

- A way to traverse a list is with a **for** loop

```
>>> for i in lb:
...     print(i)
...
hello.
How
are you?
>>>
```

- In some situations indices are necessary for writing or updating list elements
- In these cases it is possible to combine **range** and **len** built-in functions

```
>>> lb
['hello', 'everybody. ', 'How', 'are you?']
>>> for i in range(len(lb)):
...     print(i, "\t", lb[i])
...
0        hello
1        everybody.
2        How
3        are you?
```

Lists

Traversing a list

- If lists are nested, the inner list is regarded as a single element of the outer list

```
>>> lb.append([10,20,30])
>>> lb
['hello', 'everybody. ', 'How', 'are you?', [10,
    20, 30]]
>>> for i in range(len(lb)):
...     print(i,"\t", lb[i])
...
0        hello
1        everybody.
2        How
3        are you?
4        [10, 20, 30]
```

Lists

The `range` built-in function

- The **`range`** built-in function returns an immutable sequence data type
- The syntax is

```
range(stop)
range(start, stop [, step])
```

- Only **`stop`** is mandatory, as it expresses the stop value
- The argument **`start`** expresses the starting value
- The optional argument **`step`** expresses the increment (or decrement) value

Lists

The `range` built-in function

```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(2, 10))
[2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(2, 10,2))
[2, 4, 6, 8]
>>> list(range(2, 10,-2))
[]
>>> list(range(100, 0, -10))
[100, 90, 80, 70, 60, 50, 40, 30, 20, 10]
```

- The operator **+** concatenates two lists
- The operator ***** repeat a list a given (integer) number of times

```
>>> la
[1, 2, 3, 4, 5]
>>> lb
['ciao', 'a', 'TUTTI', 'voi']
>>> la + lb
[1, 2, 3, 4, 5, 'ciao', 'a', 'TUTTI', 'voi']
>>> (la+lb)[1]
2

>>> 3*la
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
```

```
>>> la = [1,2,3,4,5]
>>> la[1:3]
[2, 3]
>>> la[1:]
[2, 3, 4, 5]
>>> la[0:1] = [10,20]
>>> la
[10, 20, 2, 3, 4, 5]
```

Lists

List methods (1/2)

s[i] = x	The *i*-th element of **`s`** is replaced by **`x`**
s[i:j] = t	The slice of **`s`** (from *i*-th to *j* elements) is replaced by **`t`**
del s[i:j]	Equivalent to **`s[i:j] = []`**
s[i:j:k] = t	The elements of **`s[i:j:k]`** are replaced by the elements of **`t`**
del s[i:j:k]	Elements of **`s[i:j:k]`** are removed from **`s`**
s.append(x)	Append **`x`** to the end of the sequence
s.clear()	Delete all elements of **`s`**
s.copy()	Create a (shallow) copy of **`s`**

Lists

List methods (2/2)

s.extend(t)	Extend **s** with the content of **t**
s += t	Extend **s** with the content of **t**
s *= n	Update **s** repeating its content **n** times
s.insert(i, x)	Insert **x** in position **i**
s.pop([i])	Remove and return the *i*-th element of **s**
s.remove(x)	Remove the first element of **s** equal to **x**
s.reverse()	Invert the elements of **s**
s.sort()	Sort the elements of **s**

```
>>> la
[1, 2, 3, 4, 5, 6]
>>> lb
[1, 2, 3, 4, 5]
>>> la.insert(0,11)
>>> la
[11, 1, 2, 3, 4, 5, 6]
>>> la.insert(2,22)
>>> la
[11, 1, 22, 2, 3, 4, 5, 6]
```

```
>>> la
[11, 1, 22, 2, 3, 4, 5, 6]
>>> la.reverse()
>>> la
[6, 5, 4, 3, 2, 22, 1, 11]
>>> lb
[1, 2, 3, 4, 5]
>>> la.extend(lb)
>>> la
[6, 5, 4, 3, 2, 22, 1, 11, 1, 2, 3, 4, 5]
>>> la.sort()
>>> la
[1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 11, 22]
```

Lists

Common tasks

- **Reduce:** calculate the sum of the elements of a numeric list
- **Map:** apply the same operation to all elements of a list
- **Filter:** apply some operation to some elements of a list

- Implement the Python function **`myreduce(a)`** that returns the sum of the elements of the list **`a`**.

- Implement the Python function **`mymap(a)`** that returns a list containing the elements of the list **`a`** each multiplied by 2.

- Implement the Python function **`myfilter(a)`** that returns a list containing the odd elements of the list **`a`** each multiplied by 2.

Lists

Reduce /1

```
def sum_list(s):
    sum = 0
    for x in s:
        sum += x
    return sum
```

Lists

Reduce /2

```
def sum_list(s):
    sum = 0
    for x in s:
        sum += x
    return sum

a = [10,20,30,40]
print(sum_list(a))
print(sum(a))
```

```
def capitalize_all(s):
    result = []
    for x in s:
        result.append(x.capitalize())
    return result

a = ['hello', 'how', 'are', 'you?']
print(capitalize_all(a))
```

Lists

Filter

```
def only_upper(s):
    result = []
    for x in s:
        if x.isupper():
            result.append(x)
    return result

a = ['Hello', 'how', 'are', 'You?']
print(capitalize_all(a))
```

Lists

Deleting elements

```
>>> lista = [1,2,3,4,5]
>>> lista.pop()
5
>>> lista
[1, 2, 3, 4]
>>> lista.pop(2)
3
>>> lista
[1, 2, 4]
>>>
>>> x = lista.pop(0)
>>> x
1
```

Lists

Deleting elements

```
>>> lista = [1,2,3,4,5]
>>> del lista[1]
>>> lista
[1, 3, 4, 5]

>>> l2 = ['a', 'b', 'c', 'd']
>>> l2.remove('a')
>>> del lista[1:3]
>>> lista
[1, 5]
```

Lists

Lists and strings

- String: sequence of characters
- Lista: sequence of values
- Using `list()` it is possible to convert a string to a list of characters

```
>>> stringa = 'Hello world'
>>> s = list(stringa)
>>> s
['H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l'
    , 'd']
```

Lists

Lists and strings: function `split()`

```
>>> tlist = stringa.split()
>>> tlist
['ciao', 'a', 'tutti']
>>> str1 = 'username:password:group:home'
>>> str1.split(':')
['username', 'password', 'group', 'home']

>>> str1 = "ciao::a::tutti"
>>> str1.split('::')
['ciao', 'a', 'tutti']
```

Objects and values

```
>>> s1 = 'ciao'
>>> s2 = 'ciao'
>>> s1 is s2
True
>>> id(s1)
139857562372896
>>> id(s2)
139857562372896
```

The two strings are:

- **identical**: they refer to the same element
- **equivalent**: they contain the same elements

Objects and values

```
>>> s1 = 'ciao'
>>> s2 = 'ciao'
>>> id(s1)
139857562372336
>>> id(s2)
139857562373008
>>> s1 is s2
False
>>> if s1 == s2:
...     print("Equivalent")
...
Equivalent
>>> s1='ciao!'
>>> s2='ciao!'
>>> s1 is s2
False
```

Objects and values

```
>>> a = [1, 2, 3, 4, 5]
>>> b = [1, 2, 3, 4, 5]
>>> a is b
False
>>> if a == b:
...     print("Equivalent")
...
Equivalent
>>> id(a)
139857562454344
>>> id(b)
139857562375048
```

The two lists are:

- **not identical**: they do not refer to the same element
- **equivalent**: they contain the same elements

Objects and values

Aliasing

```
>>> a = [1, 2, 3, 4, 5]
>>> b = a
>>> b is a
True
>>> id(a)
139857562454984
>>> id(b)
139857562454984
```

- The two variables refer to the same object
- The association of a variable with an object is called a **reference**

Objects and values

Alias

- An object with more than one reference has more than one name
- The object is **aliased**

```
>>> b[0] = 44
>>> a
[44, 2, 3, 4, 5]
>>> b
[44, 2, 3, 4, 5]
>>> id(a)
139857562454984
>>> id(b)
139857562454984
```

- When a list is passed to a function, the function gets a reference to the list
- If the function modifies the list, the caller sees the change

```
>>> a
[44, 2, 3, 4, 5]
>>> id(a)
139857562454984
>>> def truncate(listx):
...     listx.pop(len(listx)-1)
...
>>> truncate(a)
>>> a
[44, 2, 3, 4]
>>> id(a)
139857562454984
```

- Some operations modify lists, some others create new lists
- For example, **append** modifies a list, the operator **+** creates a new list

```
>>> a
[44, 2, 3, 4]
>>> b = a.append(5)
>>> a
[44, 2, 3, 4, 5]
>>> b

>>> b = a + [11]
>>> a
[44, 2, 3, 4, 5]
>>> b
[44, 2, 3, 4, 5, 11]
```

- This difference is fundamental when writing functions that are supposed to modify lists

```
>>> def bad_truncate(listx):
...     listx = listx[1:]
...     print(listx)
...
>>> bad_truncate(a)
[2, 3, 4, 5]
>>> a
[44, 2, 3, 4, 5]
```

- The slice operator creates a new list

```
def bad_truncate(listx):
    listx = listx[1:]
    print("Within the function: ", listx)
    print(id(listx))

a = [100, 10, 20, 30, 40]
print(id(a))
bad_truncate(a)
print(id(a))
print("In the main: ", a)
```

```
def fine_truncate(listx):
    listx = listx[1:]
    print("Within the function: ", listx)
    print(id(listx))
    return listx

a = [100, 10, 20, 30, 40]
print(id(a))
fine_truncate(a)
print(id(a))
print("In the main: ", a)
```

Digression

Using the Montecarlo method to calculate π

```
import math
import random

def greekpi(eps):
    gpi = 0.0
    no_int_points = 0
    no_tot_points = 0
    while abs(gpi - math.pi) > eps:
        no_tot_points += 1
        x = random.random()
        y = random.random()
        if (x**2  + y**2) <= 1.0:
            no_int_points += 1
        gpi = 4 * no_int_points / no_tot_points

    print(no_int_points, no_tot_points)
    print(gpi)
    print(math.pi)


greekpi(0.000001)
```

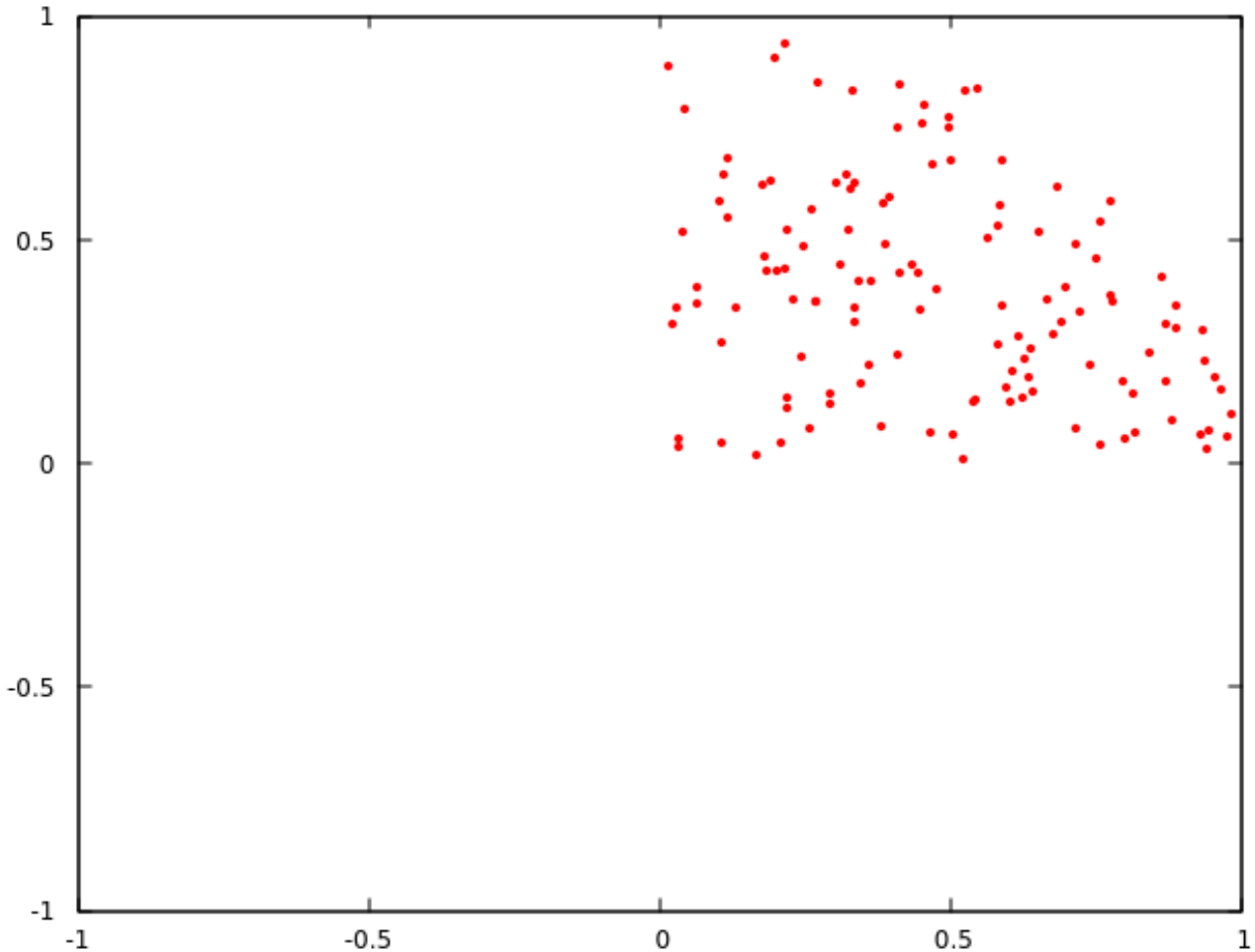
Digression

Using the Montecarlo method to calculate π

```
import math
import random

def greekpi(eps):
    gpi = 0.0
    no_int_points = 0
    no_tot_points = 0
    while abs(gpi - math.pi) > eps:
        no_tot_points += 1
        x = random.random()
        y = random.random()
        if (x**2  + y**2) <= 1.0:
            no_int_points += 1
        gpi = 4 * no_int_points / no_tot_points

    print(no_int_points, no_tot_points)
    print(gpi)
    print(math.pi)


greekpi(0.000001)
```

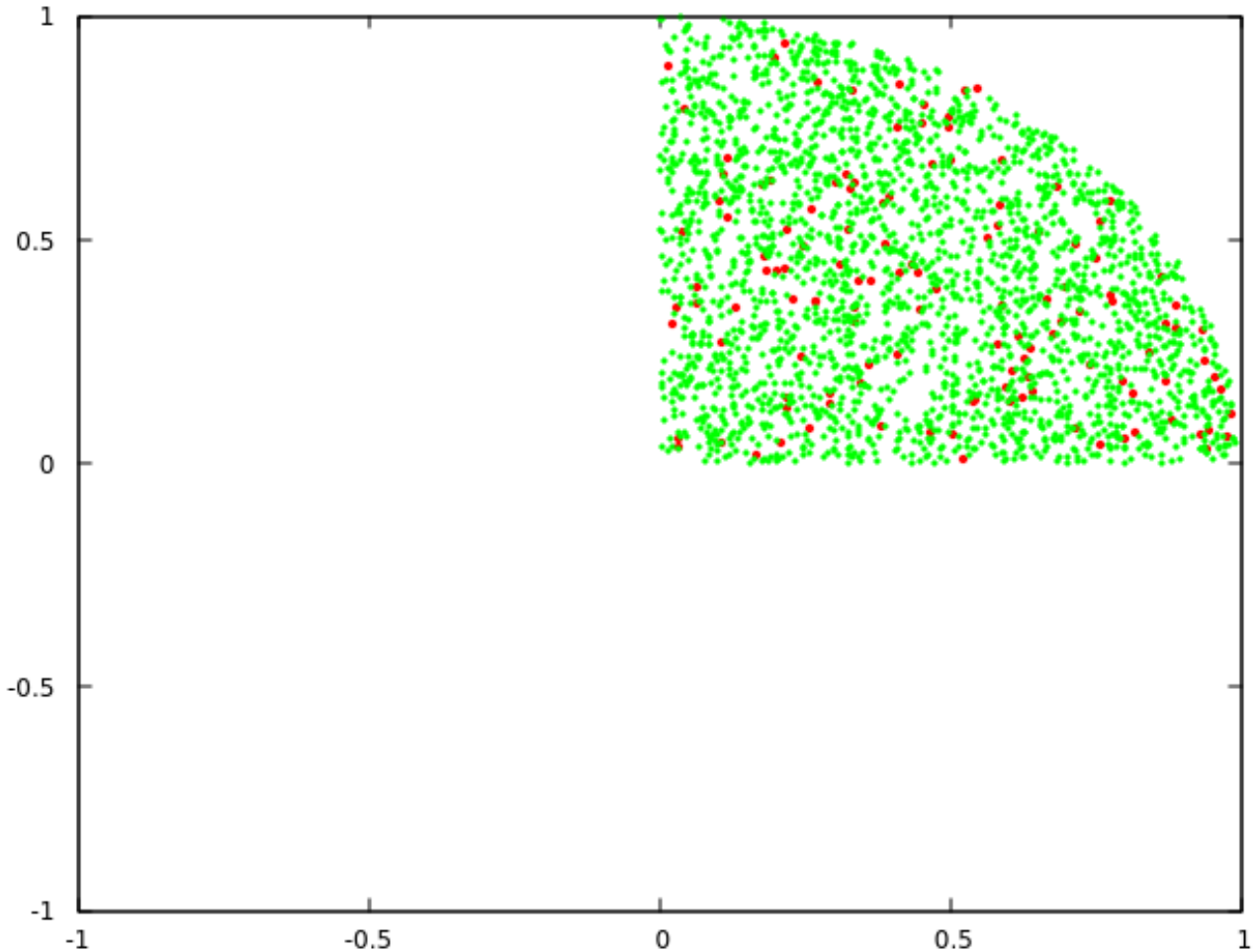
Digression

Using the Montecarlo method to calculate π



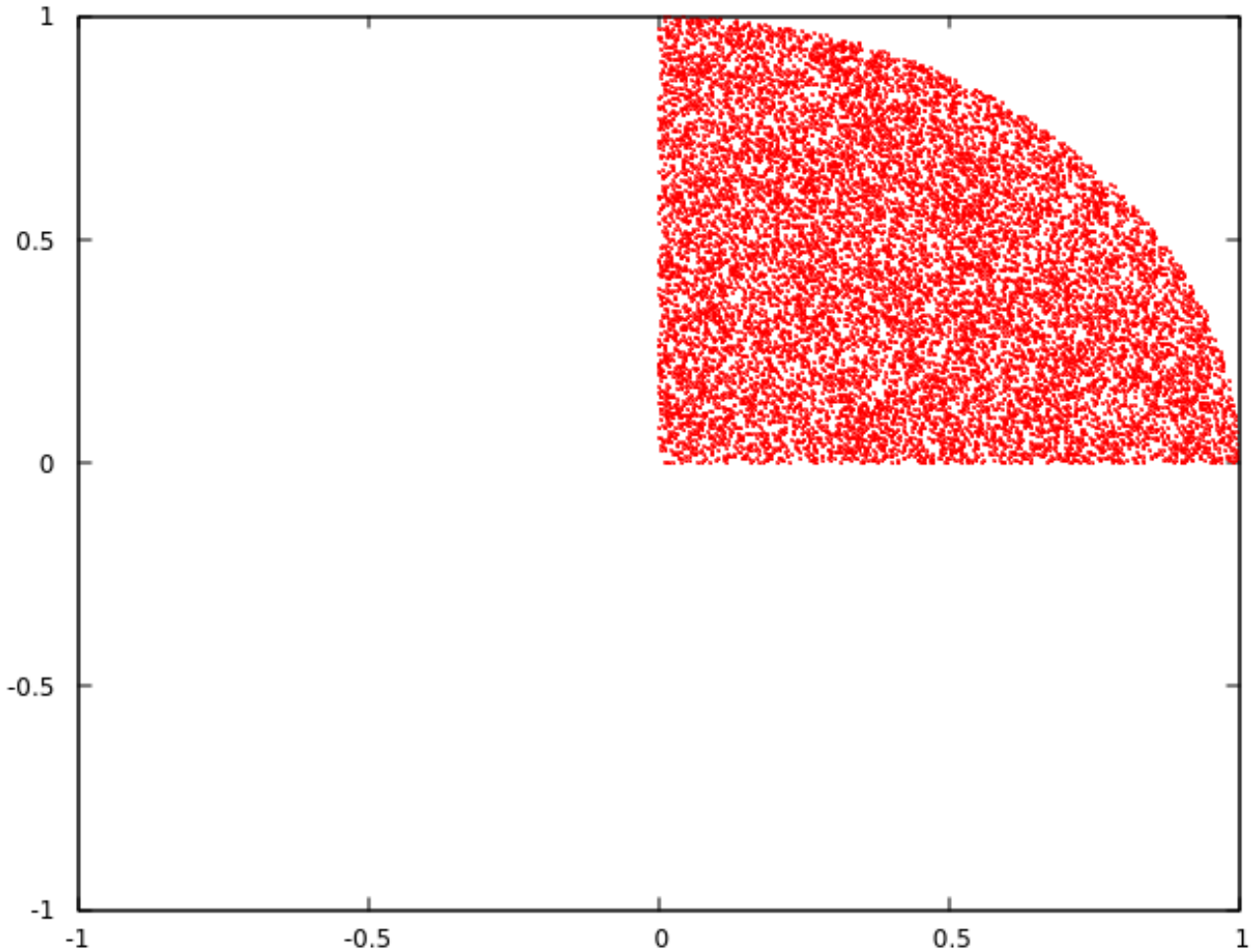
Digression

Using the Montecarlo method to calculate π



Digression

Using the Montecarlo method to calculate π



Dictionaries

- A **dictionary** represents the generalization of a list
- In a list indices are integer numbers, whereas in a dictionary indices can be (almost) every kind of datum
- A dictionary is composed of a collection of indices (**keys**) and a collection of values
- To every key corresponds one and only one value
- The elements of a dictionary are the pairs keys-value

```
phonebook = {}
print(phonebook)
{}
phonebook['joe'] = '090123456'
print(phonebook)
{'joe': '090123456'}
```

Dictionaries

```
>>> phonebook = {'joe': '090123', 'frank': '
    090456', 'ann': '090789'}
>>> print(phonebook)
{'joe': '090123', 'ann': '090789', 'frank': '
    090456'}
```

- The order of the pairs key-value cannot be easily predicted
- Actually this is not a major problem, as in dictionaries elements are not indexed using numeric indices
- A value is accessed using its corresponding key

```
>>> rubrica['peppe']
'090123'
```

Dictionaries

- If the key is not contained in the dictionary an **IndexError** is generated

```
>>> rubrica['franco']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'franco'
```

- The built-in function **len** and the boolean operator **in** can be used on dictiornaries

```
>>> len(rubrica)
3
>>> 'joe' in phonebook
True
>>> 'frank' in phonebook
False
```

Dictionaries

- Keys and values can be extracted from a dictionary:

```
>>> mykeys = phonebook.keys()
>>> mykeys
dict_keys(['joe', 'ann', 'frank'])
>>> myvalues = phonebook.values()
>>> myvalues
dict_values(['090123', '090789', '090456'])
```

- Values (and keys) returned are not iterable like lists or strings:

```
>>> mykeys[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'dict_keys' object does not support indexing
```

Dictionaries

Review

- Count how many times each letter is contained in a given string

Dictionaries

Review

- Count how many times each letter is contained in a given string
- Create 26 variables, each corresponding to a different letter of the alphabet. Traverse the string and increment the counter stored in the variable corresponding to the current letter. **(awful)**

Dictionaries

Review

- Count how many times each letter is contained in a given string
- Create a list containing 26 elements initially set to zero, each corresponding to a different letter of the alphabet. Traverse the string and increment the list element corresponding to the current letter. **(lousy)**

Dictionaries

Review

- Count how many times each letter is contained in a given string
- Create a dictionary whose keys are the letters and the corresponding values being the counters **(very good)**

Dictionaries

Review

```
def counting(text):
    d = dict()
    for c in text:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

Dictionaries

Method `get`

- Dictionaries support the method **`get`**, which requires a key and a predefined corresponding value
- If the key is contained in the dictionary, **`get`** returns the corresponding value, otherwise it will return the predefined value

```
dic = counting('hello everybody')
print(dic)
print(dic.get('a', 0))
print(dic.get('b', 0))
```

Dictionaries

Reverse lookup

- **Lookup:** given a dictionary and a key, the corresponding value is `v = diz[k]`
- **Reverse lookup:** given a dictionary and a value, find the corresponding key

Dictionaries

Reverse lookup

```
def reverse_lookup(dic, val):
    for k in dic:
        if dic[k] == val:
            return k
    print('Not found')
```

- Lists can appears as values in a dictionary
- For example, a dictionary that associates frequencies to letters can be reversed. That is, it is possible to create a dictionary that associates frequencies to letters

```
def invert_dic(dic):
    inverse = dict()
    for k in dic:
        val = dic[k]
        if val not in inverse:
            inverse[val] = [k]
        else:
            inverse[val].append(k)
    return inverse
```

Tuples

- A **tuple** is a sequence of values
- The values can be any type and they are indexed by integers (like lists)
- The main difference is that tuples are **immutable**

```
>>> t1 = 'a', 'b', 'c', 'd'
>>> t2 = ('a', 'b', 'c', 'd')
>>> t1
('a', 'b', 'c', 'd')
>>> t2
('a', 'b', 'c', 'd')

>>> type(t1)
<class 'tuple'>
>>> type(t2)
<class 'tuple'>
```

Tuples

- Synctactically, a tuple is a comma-separated list of values
- A tuple with a single element can be created adding a trailing comma

```
>>> t1 = 'a',
>>> type(t1)
<class 'tuple'>
>>> t1 = ('a'),
>>> type(t1)
<class 'tuple'>

>>> t = ('a')
>>> type(t)
<class 'str'>
```

Tuples

- A tuple can be created using the built-in function `tuple()`
- With no arguments, an empty tuple will be created

```
>>> t = tuple()
>>> t
()
>>> t = tuple('ciaociao')
>>> t
('c', 'i', 'a', 'o','c', 'i', 'a', 'o')
```

Tuples

- Most list operators also work on tuples

```
>>> t[0]
'c'
>>> t[1:4]
('i', 'a', 'o')
>>> t[0]  = 'C'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
    assignment

>>> t = ('C',) + t[1:]
>>> t
('C', 'i', 'a', 'o', 'c', 'i', 'a', 'o')
```

Tuples

- Relational operators work with tuples (and other sequences)
- The Python interpreter starts by comparing the first element from each sequence: if they are equal, it goes to the next elements, and so on, until it finds element that differ

```
>>> 'ciao' < 'Ciao'
False
>>> (0, 1, 2) < (0, 1, 3)
True
```

Tuples

Tuple assignment

- It is often useful to swap the values of two variables
- Tuple assignment allows to avoid using temporary variables

```
>>> temp = a
>>> a = b
>>> b = temp
>>>
>>> a = 11
>>> b = 22
>>> a, b = b, a
>>> a
22
>>> b
11
```

Tuples

Tuple assignment

```
>>> a, b = 1, 2, 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected
    2)


>>> account = 'gfiumara@unime.it'
>>> name, domain = account.split('@')
>>> name
'gfiumara'
>>> domain
'unime.it'
```

Tuples

Tuples as return values

- A function can only return one value, but if the value is a tuple the effect is the same as returning multiple values
- For example:

```
>>> t = divmod(11, 3)
>>> t
(3, 2)

>>> quotient, remainder = divmod(11, 3)
>>> quotient
3
>>> remainder
2
```

Tuples

Variable-length argument tuples

- Functions can take a variable number of arguments
- A parameter name that begins with * gathers arguments into a tuple
- For example:

```
>>> def printall(*args):
...     print(args)
...
>>> printall('a', 'b', 1.2, 12)
('a', 'b', 1.2, 12)
>>> printall(1)
(1,)
>>>
```

Tuples

Variable-length argument tuples

- The complement of gather is scatter
- The * operator can be used to pass to a function a tuple whose values must be scattered
- For example:

```
>>> t = (7,3)
>>> divmod(t)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: divmod expected 2 arguments, got 1

>>> divmod(*t)
(2, 1)
>>>
```

Tuples

Lists and tuples

- **zip** is a built-in function that takes two or more sequences and interleaves them
- The name refers to a zipper, which interleaves two rows of teeth

```
>>> a = 'abc'
>>> b = [1,2,3]
>>> zip(a,b)
<zip object at 0x7f323f37b848>
>>> for i in zip(a, b):
...   print(i)
...
('a', 1)
('b', 2)
('c', 3)
```

Tuples

Lists and tuples

- A list of tuples can be traversed using a **for** loop

```
>>> a = [('a', 1), ('b', 2), ('c', 3), ('d', 4)]
>>> for let, num in a:
...     print(let, num)
...
a 1
b 2
c 3
d 4
>>>
```

Tuples

Lists and tuples

- Using **`zip`** and **`for`** it is possible to traverse two lists at the same time
- For example:

```
def matches(t1, t2):
    for x,y in zip(t1,t2):
        if x == y:
            return True
    return False
```

Tuples

Lists and tuples

- The traversal of the elements of a sequence together with their indices can be carried out using the built-in function **enumerate**
- For example:

```
a = 'ciao a tutti'
for i, c in enumerate(a):
    print(i, "\t", c)

b = ['a', 'b', 'c', 'd', 'e']
for i, bchar in enumerate(b):
    print(i, "\t", bchar)

c = [('a', 1), ('b', 2), ('c', 3), ('d', 4)]
for i, t in enumerate (c):
    print(i, "\t", t)
```

Tuples

Dictionaries and tuples

- Dictionaries have a method called **items** that returns a sequence of tuples, each tuples being a key-value pair

```
>>> diz = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
>>> t = diz.items()
>>> t
dict_items([('a', 1), ('c', 3), ('b', 2), ('d', 4)])
>>> for k,v in diz.items():
...     print(k,v)
...
a 1
c 3
b 2
d 4
```

Tuples

Dictionaries and tuples

- It is possible to use a list of tuples to initialize a new dictionary

```
>>> t = [('a', 1), ('b', 2), ('c', 3), ('d', 4)]
>>> diz = dict(t)
>>> diz
{'a': 1, 'c': 3, 'b': 2, 'd': 4}

>>> t = [(1,2,3), (4,5,6), (7,8,9)]
>>> d = dict(t)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: dictionary update sequence element #0 has length
    3; 2 is required
```

- Combining **`dict`** and **`zip`** yields a concise way to create a dictionary:

```
>>> d = dict(zip('abcd', range(4)))
>>> d
{'a': 0, 'c': 2, 'b': 1, 'd': 3}
```

Tuples

Dictionaries and tuples

- Tuples can be used as keys of a dictionary

```
>>> nc = [('aa', 'bb'), ('cc', 'dd'), ('ee', 'ff')]
>>> nc
[('aa', 'bb'), ('cc', 'dd'), ('ee', 'ff')]
>>> tel = ['001', '002', '003']
>>> phonebook = dict(zip(nc, tel))
>>> phonebook
{('aa', 'bb'): '001', ('ee', 'ff'): '003', ('cc', 'dd'): '002
    '}
```

Sets

- A **set** is a container that stores a collection of unique values
- Unlike a list, the elements of a set are not stored in any particular order and cannot be accessed by position
- The operations available are the same that apply to mathematical sets
- Set operations are particularly faster than the equivalent list operations (due to the lack of a particular order)

Sets

- A set with initial elements can be created enclosing the elements in curly brackets

```
aset = {11, 4, 10, 2}
bset = {'d', 'a', 'e'}

numbers = [10, 10, 3, 4]
nset = set(numbers)
```

Sets

- An empty set cannot be created using the empty curly brackets
- Instead, the `set()` function with no arguments must be used

```
newset = set()
```

Sets

- The `len` function can be used
- To determine whether an element is contained in a set, the `in` operator can be used

```
print(len(nset))

if 10 in numbers:
    print("The element", 10, is contained in the
        set)
```

Sets

- Sets are unordered, therefore elements cannot be accessed by position
- Instead, a **for** can be used to iterate over the individual elements

```
for i in range(len(nset)):
    print(i,"\t", nset[i])
# TypeError: 'set' object is not subscriptable

for i in nset:
    print(i)
```

Sets

Adding elements

- Sets are mutable collections, so elements can be added or removed

---

```
nset.add(20)
```

- If the element being added is not already contained in the set, it will be added and the size of the container increased by one
- However, a set cannot contained duplicate elements
- Attempting to adding elements already in the set produces no effect (no addition, no error message)

- There are two methods to remove an element from a set

```
nset.discard(20)

nset.remove(5)
```

- **`discard`** removes an element, but has no effect if the element is not a member of the set
- **`remove`** raises an exception if the given element is not a member of the set

- A set A is a **subset** of the set B if and only if every element of A is also an element of B

```
ca = {'red', 'white'}
it = {'red', 'white', 'green'}
fr = {'red', 'white', blue'}

if ca.issubset(it):
    print("Subset")

if it == fr:
    print("The same colors")

if ca != fr:
    print("Not the same colors")
```

- Write the Python function **`mysubset(s,t)`** that returns **`True`** if set **`s`** is a subset of set **`t`**. Do not use the set function **`issubset()`**.

- Write the Python function **`mypropersubset(s,t)`** that returns **`True`** if set **`s`** is a proper subset of set **`t`**. A set is proper subset of another set if it is a subset, but the sets are not equal and the subset is not empty. Do not use the set function **`issubset()`**.

- The **union** of two sets contains all of the elements of both sets, with duplicates removed
- A new set is returned (neither of the sets is modified)

```
ca = {'red', 'white'}
it = {'red', 'white', 'green'}
fr = {'red', 'white', blue'}

it_u_fr = it.union(fr)
fr_u_it = fr.union(it)
```

- Write the Python function **`myunion(s,t)`** that returns the set union of the sets **`s`** and **`t`**. Do not use the set function **`union()`**.

Sets

Intersection

- The **intersection** of two sets contains all of the elements that are in both sets
- A new set is returned (neither of the sets is modified)

```
ca = {'red', 'white'}
it = {'red', 'white', 'green'}
fr = {'red', 'white', blue'}

it_i_fr = it.intersection(fr)
fr_i_it = fr.intersection(it)
```

- Write the Python function **`myintersection(s,t)`** that returns the set intersection of the sets **`s`** and **`t`**. Do not use the set function **`intersection()`**.

- The **difference** of two sets contains all of the elements of the first set that are not contained in the second
- A new set is returned (neither of the sets is modified)

```
ca = {'red', 'white'}
it = {'red', 'white', 'green'}
fr = {'red', 'white', blue'}

it_d_fr = it.difference(fr)
fr_d_it = fr.difference(it)
```

- Write the Python function **`frequency(text)`** that counts how often each word occurs in a text file.