Python Fundamentals I

Experis KickStart
ManpowerGroup

Variables

- Variables are used to store data in our programs
- To create a variable in Python we use an assignment statement:

```
variable_name = expression
```

  - variable_name is the name of the variable, (=) is the assignment operator, and expression is just a combination of values, variables and operators
- Example for creating a variable:

```
num = 5
```

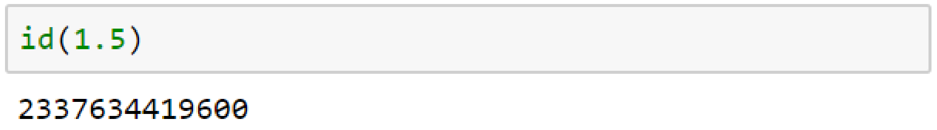
```
num
```

```
5
```

- Python is a dynamically-typed language - we don't need to declare the type of the variable ahead of time
  - The variable’s type is inferred from its definition, e.g., the number 5 is assumed to be an int

Objects

- Objects are combinations of variables, functions, and data structures
- Objects represent the entities in your program
- Everything in Python is an object (including basic types such as integers, strings, etc.)
- When an object is initiated, it is assigned a unique **object id**
- The built-in function **id**() returns the identity of an object as an integer
  - The id typically corresponds to the object's location in memory, but this is platform-dependent
- For example:

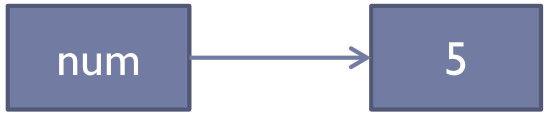


```
id(1.5)
```

```
2337634419600
```

Variables and Objects

- A variable name can be assigned ("bound") to any object and used to identify that object in future calculations
- For example, when Python encounters the statement **num = 5**, it does the following:
  - Creates an int object with the value 5
  - Assign the variable name **num** to this object
    - num contains the memory location where the object 5 is stored



```
num = 5
id(num)
```

```
1831890528
```

Variables and Objects

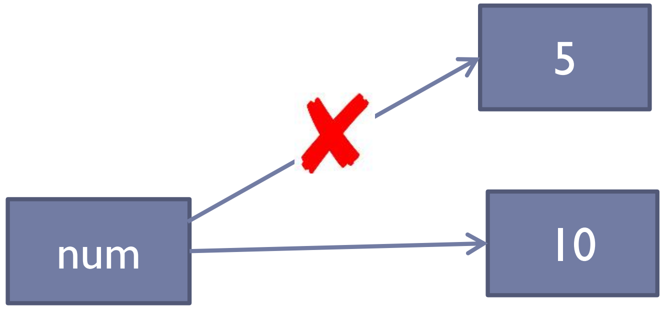
- When we assign a new value to a variable the reference to the old value is lost
- For example, when "num" is assigned to the value 10, the reference to value 5 is lost
- At this point, no variable is pointing to the memory location of the object 5
- When this happens, Python Interpreter automatically removes the object from the memory through a process known as **garbage collection**

```
num = 5
id(num)
```

```
1831890528
```

```
num = 10
id(num)
```

```
1831890688
```



Printing Variables

- You can print the value of a variable using the **print()** function

```
num = 12
```

```
print(num)
```

```
12
```

- Trying to access a variable before assigning any value to it results in a NameError:

```
age
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-23-d075b0315d53> in <module>()
----> 1 age

NameError: name 'age' is not defined
```

Variable Names

- In Python, we have the following rules to create a valid variable name:
  - Only use letters (a-z, A-Z), underscore (_) and numbers (0-9)
  - It must begin with a underscore (_) or a letter
  - You can't use reserved keywords to create variables names
  - Variable name can be of any length
- Examples for valid names: home4you, after_you, _thatsall, all10, python_101
- Examples for invalid names: $money, hello pi, 2001, break
- Python is case-sensitive language, e.g., HOME and Home are two different variables
- By convention, variable names should be lowercase, with words separated by underscores as necessary to improve readability, e.g., grades_average

Python Keywords

- Python Keywords are words that have a specific meaning in the Python language
- That's why, we are not allowed to use them as variables names
- Here is the list of Python keywords:

and	assert	break	class	continue
def	del	elif	else	except
finally	for	from	global	if
import	in	is	lambda	nonlocal
not	or	pass	print	raise
return	try	while	yield	

Comments

- Comments are used to add notes to a program and help other understand your code
- In Python, everything from # to end of the line is considered a comment

```
# This is a comment on a separate line
print("Testing comments")  # This is a comment after print statement
```

```
Testing comments
```

  - Generally, prefer to place comments on their own lines rather than “inline” with code
- Explain *why* your code does what, don’t simply explain *what* it does
- A bad comment:

```
# Increase i by 10
i += 10
```

- A good comment:

```
# Skip the next 10 data points
i += 10
```

Comments

- Some programmers advocate aiming to minimize the number of comments by carefully choosing meaningful identifier names
- For example, if we rename our index, we might even do away with the comment altogether:

```
DATA_SKIP = 10

data_index += DATA_SKIP
```

- Keep comments up-to-date with the code they explain
- It is all too easy to change code without synchronizing the corresponding comments:

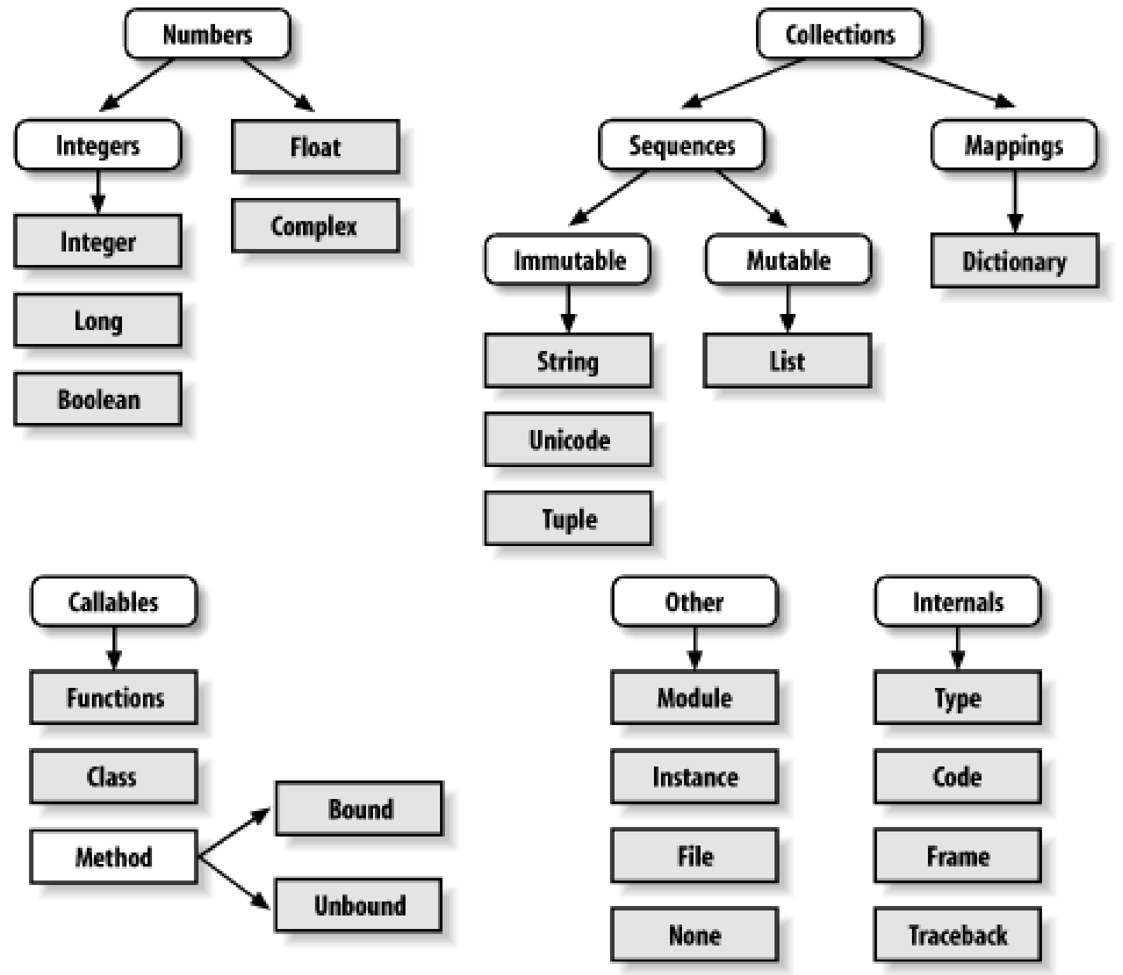
```
# Skip the next 10 data points
i += 20
```

Data Types

- A data type defines a set of values along with operations that can be performed on those values
- Each variable and value we use in our programs has a type associated with it
- Python has 5 basic data types:

Type	Description
int	Whole numbers of unlimited range (limited only by the available memory) Older versions of Python had separate types for int and long
float	Double precision floating-point numbers, which provide approximately 15-16 digits of precision (like double in C)
complex	Complex numbers are represented as a pair of floating-point numbers The real and imaginary parts of a complex number z are available in z.real and z.imag.
bool	boolean (True or False)
str	Sequences of characters

Python Built-In Type Hierarchy



type()

- The built-in function **type()** can be used to determine the data type of the object referred by a variable:

```
num = 5
type(num)
```

```
int
```

```
s = "hello"
type(s)
```

```
str
```

```
price = 2.3
type(price)
```

```
float
```

```
allowed = True
type(allowed)
```

```
bool
```

Literals

- Explicit values we use in our programs is known as **literal**
  - For example, 10, 88.22, 'test' are called literals
- Literals also have types associated with them

```
type(54)
```

```
int
```

```
type("a string")
```

```
str
```

```
type(3.14)
```

```
float
```

```
type("3.14")
```

```
str
```

Types of Numbers

- Numbers in Python come in three types*:*
  - integers (type: int)
  - floating point numbers (type: float)
  - complex numbers (type: complex)
- Any single number containing a period (.) is considered by Python to specify a floating point number
- Scientific notation is supported using ‘e’ or ‘E’ to separate the significand (mantissa) from the exponent
  - For example, 1.67263e-7 represents the number 1.67263×10^{-7}
- Complex numbers such as 4+3j consist of a real and an imaginary part (denoted by j in Python), each of which is itself represented as a floating point number (even if specified without a period)

Types of Numbers

```
5
```

```
5
```

```
5.
```

```
5.0
```

```
0.100
```

```
0.1
```

```
0.0001
```

```
0.0001
```

```
# Numbers smaller than 0.0001 are displayed in scientific
# notation
0.00009999
```

```
9.999e-05
```

```
5.123e8
```

```
512300000.0
```

```
3j
```

```
3j
```

```
2 + 3j
```

```
(2+3j)
```

```
# A complex number may be specified by separating the real
# and imaginary # parts in a call to complex()
complex(2.3, 1.2)
```

```
(2.3+1.2j)
```

```
complex(5)
```

```
(5+0j)
```

Dynamic Typing

- A variable in Python can contain any data
- A variable can at one moment be a string and later receive a numeric value:

```
foo = 42      # foo is now a number
print(type(foo))

foo = 'bar'   # foo is now a string
print(type(foo))

foo = True    # foo is now a boolean
print(type(foo))
```

```
<class 'int'>
<class 'str'>
<class 'bool'>
```

- Python automatically detects the type of the variable and operations that can be performed on it based on the type of the value it contains

Dynamic Typing

- Python is a **strongly**, **dynamically** typed language
- **Dynamic typing** means that there are data types, but variables are not bound to any of them
- **Strong typing** means that the type of a value doesn't suddenly change
  - A string containing only digits doesn't magically become a number, as may happen in JavaScript
  - Every change of type requires an explicit conversion

Named Constants

- Constants are variables whose values don’t change during the lifetime of the program
- Python doesn't have a special syntax to create constants
- We create constants just like ordinary variables
- However, to separate them from an ordinary variable, we use all uppercase letters

```
DOLLAR_TO_EURO = 0.848
DOLLAR_TO_POUND = 0.748

price = 100 * DOLLAR_TO_EURO
print(price)
```

```
84.8
```

Displaying Multiple Items with print()

- We can use the **print()** function to print multiple items in a single call by separating each item with a comma (,)
- The items will be printed to the console separated by spaces

```
age = 25
print("Your age is:", age)
```

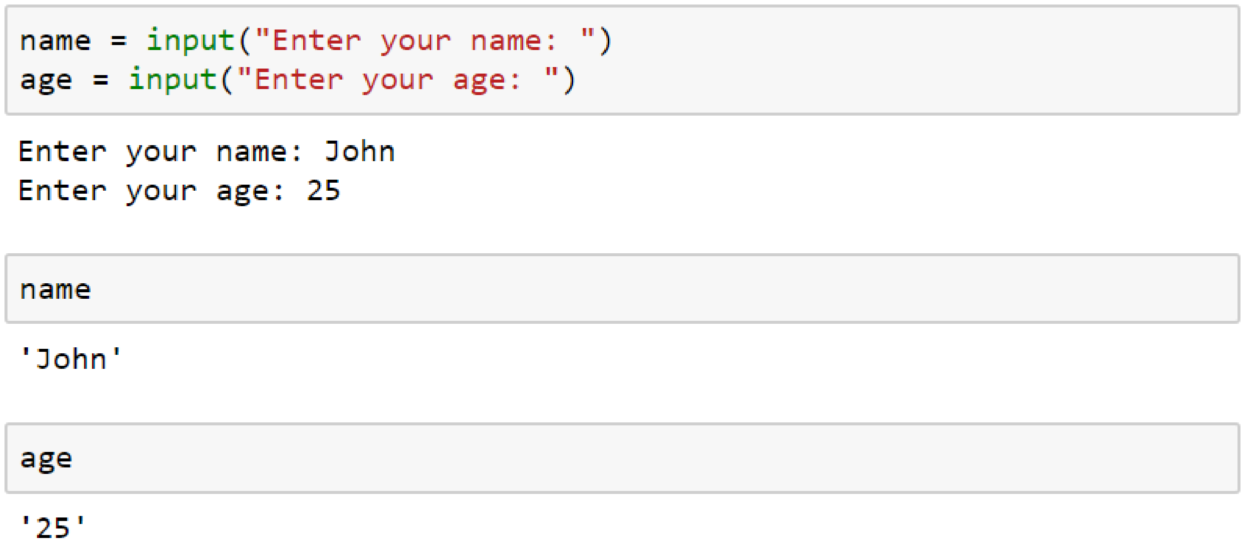
```
Your age is: 25
```

Reading Input from Keyboard

- The function **input()** is used to read input from the keyboard. The syntax is:

```
var = input(prompt)
```

  - prompt is an optional string which instructs the user to enter the input
  - The input() function reads the input data from the keyboard and returns it as a string
  - The entered data is then assigned to a variable named var for further processing



```
name = input("Enter your name: ")
age = input("Enter your age: ")
```

```
Enter your name: John
Enter your age: 25
```

```
name
```

```
'John'
```

```
age
```

```
'25'
```

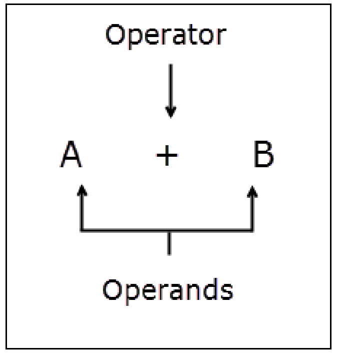
Exercise (1)

- Write a Python program which accepts the user's first and last name
- Then it prints a welcome message that starts with the word “Welcome,” followed by the user’s first and last name in reverse order with a space between them
- For example:

```
Please enter your first name: Roi
Please enter your last name: Yehoshua
Welcome, Yehoshua Roi
```

Operators

- **Operator**: An operator is a symbol which specifies a specific action
- **Operand**: An operand is a data item on which operator acts



- Some operators require two operands (binary operators) while others require only one (unary operators)
- **Expression**: An expression is a combination of operators, variables, constants and function calls that results in a value
- For example:

```
1 + 8
(3 * 9) / 5
a * b + c * 3
a + b * math.pi
d + e * math.sqrt(441)
```

Arithmetic Operators

- Arithmetic operators are commonly used to perform numeric calculations
- Python has the following arithmetic operators:

Operator	Description	Example
+	Addition	100 + 45 = 145
-	Subtraction	500 – 65 = 435
*	Multiplication	25 * 4 = 100
/	Float division	10 / 2 = 5.0
//	Integer division	10 / 2 = 5
**	Exponentiation	5 ** 3 = 125
%	Remainder	10 % 3 = 1

- Note that + and - operators can be binary as well as unary
  - For example: in -5, the - operator is acting as a unary operator, whereas in 100 - 40, it is acting as a binary operator

Float Division Operator (/)

- The / operator performs a floating point division
- Always returns a floating point number result, even if it acts on integers

```
6 / 3
```

```
2.0
```

```
2 / 3
```

```
0.6666666666666666
```

```
50 / 2.5
```

```
20.0
```

```
-5 / 2.1
```

```
-2.380952380952381
```

Integer Division Operator (//)

- The // operator always **rounds down** the result to the nearest integer
- The type of the result is int only if both operands are int, otherwise it returns a float

```
6 // 3
```

```
2
```

```
9 // 2
```

```
4
```

```
2.7 // 2
```

```
1.0
```

```
-5 // 2
```

```
-3
```

Exponentiation Operator (**)

- We use a**b operator to calculate a^b

```
15 ** 2
```

```
225
```

```
3 ** 4
```

```
81
```

```
5 ** 1.2
```

```
6.898648307306074
```

```
9 ** 0.5
```

```
3.0
```

Remainder Operator (%)

- The % operator returns the remainder after dividing left operand by the right operand
- Again the number returned is an int only if both operands are int

```
5 % 2       # 5 = (2 * 2) + 1
```

```
1
```

```
13 % 5      # 13 = (2 * 5) + 3
```

```
3
```

```
-13 % 5     # -13 = (-3 * 5) + 2
```

```
2
```

- The remainder operator % is a very useful operator in programming
  - One common use of % operator is to determine whether a number is even or not

Operator Precedence and Associativity

Operator	Description	Associativity
[v1, …], { v1, …}, { k1: v1, …}, (…)	List/set/dict/generator creation or comprehension, parenthesized expression	left to right
seq [n], seq [n : m], func (args…), obj.attr	Indexing, slicing, function call, attribute reference	left to right
**	Exponentiation	right to left
+x, -x, ~x	Positive, negative, bitwise not	left to right
*, /, //, %	Multiplication, float division, integer division, remainder	left to right
+, -	Addition, subtraction	left to right
<<, >>	Bitwise left, right shifts	left to right
&	Bitwise and	left to right
|	Bitwise or	left to right
in, not in, is, is not, <, <=, >, >=, !=, ==	Comparision, membership and identity tests	left to right
not x	Boolean NOT	left to right
and	Boolean AND	left to right
or	Boolean OR	left to right
if-else	Conditional expression	left to right
lambda	lambda expression	left to right

Operator precedence goes from higher to lower

Operator Precedence

- Whenever we have an expression where operators involved are of different precedence, the operator with a higher precedence is evaluated first:

```
10 + 5 * 3
```

```
25
```

```
3 * 5 ** 2
```

```
75
```

- We can change the operator precedence by wrapping parentheses around the expression which we want to evaluate first:

```
(10 + 5) * 3
```

```
45
```

```
(3 * 5) ** 2
```

```
225
```

Operator Associativity

- Operator associativity defines the direction in which operators of the same precedence are evaluated
- The associativity of all arithmetic operators is from left to right
- Except for the exponentiation operator ** which is evaluated from right to left
  - Since a**b**c = $a^{b^c} = a^{(b^c)}$ = a**(b**c)

```
5 + 12 / 2 * 4    # 5 + ((12 / 2) * 4) == 5 + 6.0 * 4
```

```
29.0
```

```
2 ** 2 ** 3    # 2**(2**3) == 2**8
```

```
256
```

```
(2 ** 2) ** 3    # 4**3
```

```
64
```

Type Conversions

- When a calculation involves data of different types Python has the following rules:
  - When both operands are int, the result will be an int
  - When both operands are float, the result will be a float
  - When one operand is of float type and the other is of type int then the result will be a float
- Sometimes, we need to convert data from one type to a different type
  - For example, when reading numeric inputs from the console
- Python provides us the following conversion functions:

Function	Description	Examples
int()	Accepts a string or a number and returns a value of type int. A floating point number is rounded down in casting it into an int.	int(2.7) returns 2 int(“25”) returns 25
float()	Accepts a string or a number and returns a value of type float	float(42) return 42.0 float(“1.6”) returns 1.6
str()	Accepts any value and returns a value of type str	str(12) returns “12” str(3.4) returns “3.4”

Type Conversions

- A program that asks the user to enter two numbers and prints their sum:

```
num1 = int(input("Enter first number: ")) # int() is used to convert the input to a number
num2 = int(input("Enter second number: "))

sum = num1 + num2
print("Their sum is:", sum)
print("Their sum is: " + str(sum)) # another way to print a string together with a number
```

```
Enter first number: 5
Enter second number: 7
Their sum is: 12
Their sum is: 12
```

Exercise (2)

- Predict the results of the following expressions and check them in Jupyter Notebook:

```
2.7 / 2
2 / 4 - 1
2 // 4 - 1
(2 + 5) % 3
2 + 5 % 3
3 * 4 // 6
3 * (4 // 6)
3 * 2 ** 2
3 ** 2 * 2
-2 ** 2
2 ** -2
(-2) ** 2
-2 ** 3 ** 2
(-2) ** 3 ** 2
```

Exercise (3)

- Write a program that asks the user to enter the length and width of a rectangle, and prints its area and perimeter

```python
length = int(input("Enter the length of the rectangle: "))
width = int(input("Enter the width of the rectangle: "))

area = length * width
perimeter = 2 * (length + width)

print("The rectangle's area is:", area)
print("The rectangle's perimeter is:", perimeter)
```

```
Enter the length of the rectangle: 5
Enter the width of the rectangle: 7
The rectangle's area is: 35
The rectangle's perimeter is: 24
```

Breaking Statements into Multiple Lines

- Python doesn't place any restriction on the length of a line,
- However, for ease of reading, it is usually a good idea to keep the lines of your program to a fixed maximum length (80 characters is recommended)
- There are two ways in Python to break long statements into multiple lines:
- Using parenthesis:

```
(1111100 + 45 - (88 / 43) + 783 +
10 - 33 * 1000 +
88 + 3772)
```

```
1082795.953488372
```

- Using the line continuation symbol (\ ):

```
1111100 + 45 - (88 / 43) + 783 + \
10 - 33 * 1000 + \
88 + 3772
```

```
1082795.953488372
```

Compound Assignment Operator

- In programming it is very common to change the value of a variable and then reassign the value back to the same variable, i.e., a = a + 5
- Such reassignments have useful shorthand notation: the **compound assignment operator**
- The following table lists the compound assignment operators available in Python:

Operator	Example	Equivalent to
+=	x += 5	x = x + 5
-=	x -= 5	x = x – 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
//=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
**=	x **= 5	x = x ** 5

Compound Assignment Operators

- Example:

```
x = 5
x *= 2
x
```

```
10
```

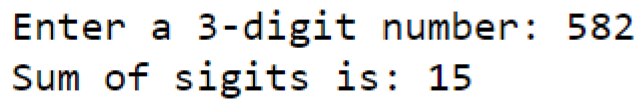
```
x += 1
x
```

```
11
```

- C-style increment and decrement operations such as a++ for a += 1 are *not supported* in Python

Exercise (4)

- Write a program that asks the user to enter a 3-digit number and prints the sum of its digits
- Use the compound assignment operators
- Sample run:



```
Enter a 3-digit number: 582
Sum of sigits is: 15
```

Methods and Attributes of Numbers

- Python numbers are **objects** (everything in Python is an object)
- Thus, they have **attributes**, accessed using the “dot” notation `<object>.<attribute>`
- For example, complex number objects have the attributes **real** and **imag**, which are the real and imaginary (floating point) parts of the number:

```
(4+5j).real
```

```
4.0
```

```
(4+5j).imag
```

```
5.0
```

- Numbers also have **methods**: functions that belong to objects
- Methods are called on an object using the following notation:

```
object.method_name(arg1, arg2,…, argN)
```

Methods and Attributes of Numbers

- For example, complex numbers have a method, **conjugate()**, which returns the complex conjugate:

```
(4+5j).conjugate()
```

```
(4-5j)
```

- Integers and floating point numbers don't actually have many useful attributes
- But if you're curious you can find out how many bits an integer takes up in memory by calling its **bit_length()** method:

```
(5733382412312391233243).bit_length()
```

```
73
```

Mathematical Functions

- Python provides the following built-in mathematical functions:

Function	**Description**	**Example**
abs(*number*)	Returns the absolute value of *number*	abs(-12) = 12 abs(112.21) = 112.21
round(*number*)	Rounds *number* to the nearest integer	round(17.3) = 17 round(8.6) = 9
round(*number*, *ndigits*)	Rounds *number* to *ndigits* after decimal point	round(3.14159, 2) = 3.14 round(2.71828, 2) = 2.72
min(*arg1*, *arg2*, ..., *argN*)	Returns the smallest item among *arg1*, *arg2*, ..., *argN*	min(12, 2, 44, 199) = 2
max(*arg1*, *arg2*, ..., *argN*)	Returns the largest item among *arg1*, *arg2*, ..., *argN*	max(991, 22, 19) = 991

Mathematical Functions

- Examples for the **abs()** function:

```
abs(-5.2)
```

```
5.2
```

```
abs(-2)
```

```
2
```

```
abs(3+4j)
```

```
5.0
```

- This is an example of *polymorphism*: the same function, abs, does different things to different objects:
  - If passed a real number, *x*, it returns |*x*|, the non-negative magnitude of that number
  - If passed a complex number, *z* = *x* + i*y*, it returns the modulus $z = \sqrt{x^2 + y^2}$

Mathematical Functions

- Examples for the **round()** function:

```
round(-9.62)
```

```
-10
```

```
round(7.5)
```

```
8
```

```
round(4.5)
```

```
4
```

- Note that in Python 3, this function employs *Banker's rounding*: if a number ismidway between two integers, then the even integer is returned

Modules in Python

- Python uses modules to group related functions, classes, and variables
  - For example, the **math** module contains various mathematical functions
  - **datetime** module contains various classes and functions for working with dates and times
- To use functions, constants or classes defined inside a module, we first have to **import** it using the import statement
- This statement finds and loads the module into memory
- The syntax of the import statement is as follows:

```
import module_name
```

- For example, to import the math module in a Jupyter Notebook write:

```
import math
```

Modules in Python

- To use a method or a constant from a module, you type the module name followed by the dot (.) operator, and the name of the method or constant that you need
- For example, to use the **sqrt()** function from the math module, type:

```
math.sqrt(225)
```

```
15.0
```

- It is possible to import the math module with ‘from math import *’ and access its functions directly:

```
from math import *
cos(pi)
```

```
-1.0
```

  - However, this is not recommended in Python programs, since there is a danger of name conflicts (particularly if many modules are imported in this way)

The math Module (1)

- Some standard functions and constants provided by the math module:

Function/Constant	Description	Example
math.pi	The value of π	
math.e	The value of e	
math.ceil(*x*)	Returns the smallest integer greater than or equal to *x*	math.ceil(3.621) = 4
math.floor(*x*)	Returns the largest integer smaller than or equal to *x*	math.floor(3.621) = 3
math.sqrt(*x*)	Returns the square root of *x* as float	math.sqrt(144) = 12.0
math.exp(*x*)	Returns the exponent of *x* (e^x)	math.exp(2) = 7.3891
math.log(*x*)	Returns the natural log of *x* to the base e	math.log(2) = 0.6931
math.log10(*x*)	Returns the log of x to the base 10	math.log10(999) = 2.9996
math.log(*x*, *b*)	Returns the log of *x* to the given base *b*	math.log(2, 2) = 1.0
math.sin(*x*)	Returns the sine of *x* radians	math.sin(math.pi/2) = 1.0
math.cos(*x*)	Returns the cosine of *x* radians	math.cos(0) = 1.0
math.tan(*x*)	Returns the tangent of *x* radians	math.tan(45) = 1.61

The math Module (2)

- More math functions:

Function/Constant	Description	Example
math.degrees(*x*)	Converts the angle from radians to degress	math.degrees(math.pi/2) = 90
math.radians(*x*)	Converts the angle from degrees to radians	math.radians(90) = 1.5707
math.hypot(*x*, *y*)	The Eculidean norm $\sqrt{x^2 + y^2}$	math.hypot(3,5) = 5.38095
math.factorial(*x*)	*x*!	math.factorial(5) = 120

- A complete list of the functions and constants provided by the math module can be found at https://docs.python.org/3/library/math.html

Math Functions – Example

```
import math
```

```
math.ceil(3.5)    # the smallest integer greater than or equal to 3.5
```

```
4
```

```
math.floor(3.5)   # the largest integer smaller than or equal to 3.5
```

```
3
```

```
math.sqrt(144)   # square root of 144
```

```
12.0
```

```
math.log(2)     # find log of 2 to the base e
```

```
0.6931471805599453
```

```
math.log(2, 5)   # find log of 2 to the base 5
```

```
0.43067655807339306
```

```
math.cos(0)
```

```
1.0
```

```
math.cos(0)
```

```
1.0
```

```
math.sin(math.pi/2)
```

```
1.0
```

```
math.degrees(math.pi/2)
```

```
90.0
```

```
math.hypot(3,5)   # The Eculidean norm sqrt(3**2 + 5**2)
```

```
5.830951894845301
```

```
math.factorial(5)   # 5! = 1 * 2 * 3 * 4 * 5
```

```
120
```

Exercise (5)

- Write a program that computes the area of a triangle given its sides
- Get from the user the lengths of the three triangle sides: *a*, *b*, *c*
- Use Heron's formula to compute the triangle's area:

$$A = \sqrt{s(s-a)(s-b)(s-c)}, \quad \text{where } s = \frac{1}{2}(a+b+c)$$

- For example, if *a* = 4.503, *b* = 2.377, *c* = 3.902, the area is 4.63511

Exercise (6)

- Explain the (surprising?) behavior of the following short code:

```
d = 8
e = 2
from math import *
sqrt(d ** e)
```

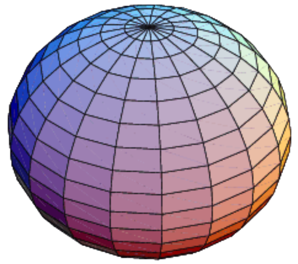
```
16.88210319127114
```

Exercise (7)

- The Earth's surface area can be approximated to the form of an oblate spheroid with semi-major and semi-minor axes *a* = 6378137.0 m and *c* = 6356752.314245 m
  - Oblate shperiod is a sphere that is wider at its horizontal axis than it is at its vertical axis
- Use the formula for the surface area of an oblate spheroid to calculate the surface area of Earth:

$$S_{obl} = 2\pi a^2 \left(1 + \frac{1-e^2}{e}\operatorname{atanh}(e)\right), \quad \text{where } e^2 = 1 - \frac{c^2}{a^2}$$

- Compare it with the surface area of Earth if it were assumed to be a perfect sphere with radius 6371 km $S_{sphere} = 4\pi r^2$



Exercise (8)

- Get a positive integer *x* from the user
- Print the number of digits that the number *x* has
  - For example, if x = 5731 you should print 4
- Use only numerical functions and operators
  - i.e., you're not allowed to convert the number into a string

The bool Type

- The **bool** data type represents two states: true or false
- The reserved keywords **True** and **False** define the values true and false
- A variable of type bool can contain only one of these two values

```
var1 = True
```

```
type(var1)
```

```
bool
```

```
var1
```

```
True
```

- Internally, Python uses 1 and 0 to represent True and False respectively:

```
int(True)
```

```
1
```

```
int(False)
```

```
0
```

Truthy and Falsy Values

- **Truthy values**: Values which are equivalent to the bool value True
- **Falsy values**: Values which are equivalent to the bool value False
- In Python, the following values are considered falsy:
  - False
  - Zero: 0, 0.0
  - None
  - Empty sequence, e.g., '', [], ()
  - Empty dictionary {}
- Everything else is considered as truthy
- We can also test whether a value is truthy or falsy by using the **bool()** function
  - If value a truthy then bool() function returns True, otherwise it returns False

Truthy and Falsy Values

```
bool("")    # an empty string is a falsy value
```

```
False
```

```
bool("bool")    # the string "bool" is a truthy value
```

```
True
```

```
bool(12)    # int 12 is a truthy value
```

```
True
```

```
bool(0)    # int 0 is a falsy value
```

```
False
```

```
bool([])    # an empty list is a falsy value
```

```
False
```

```
bool(None)  # None is a falsy value
```

```
False
```

Relational Operators

- Relational operators (aka comparison operators) allow us to compare values
- If the result of the comparison is true, then a bool value True is returned, otherwise the bool value False is returned
- The relational operators available in Python:

Operator	Description	Example
<	Smaller than	3 < 4 (True)
>	Greater than	90 > 450 (False)
<=	Smaller than or equal to	10 <= 11 (True)
>=	Greater than or equal to	31 >= 40 (False)
!=	Not equal to	100 != 101 (True)
==	Equal to	50 == 50 (True)

- Note the difference between == and =
  The single equals sign = is an assignment operator, used to assign a value to a variable, while the double equals sign == is an equality operator, used to test whether two values are equal or not.

Relational Operators

- Python can, as far as possible without ambiguity, compare objects of different types:

```
4 >= 3.14
```

```
True
```

```
0 == False
```

```
True
```

- Care should be taken in comparing floating point numbers for equality
  - Because they are not stored exactly and calculations involving them frequently leads to a loss of precision, this can give unexpected results to the unwary:

```
a = 0.01
b = 0.1 ** 2
a == b
```

```
False
```

```
print("a =", a, "b =", b)
```

```
a = 0.01 b = 0.010000000000000002
```

Logical Operators

- Logical operators are used to combine two or more boolean expressions and tests whether they are true or false
- Expressions containing logical operators are known as **boolean expressions** (or logical expressions)
- The logical operators available in Python:

Operator	**Description**
and	AND operator
or	OR operator
not	NOT operator

Logical Operators

- The **and** operator returns True if both operands are true, otherwise it returns False

P	Q	P and Q
False	False	False
False	True	False
True	False	False
True	True	True

- The **or** operator returns True if one of the operands is true, otherwise it returns False

P	Q	P or Q
False	False	False
False	True	True
True	False	True
True	True	True

- The **not** operator negates the value of the expression

P	not P
False	True
True	False

Logical Operators

- In compound expressions, the comparison operators are evaluated first, and then the logic operators in order of precedence: not, and, or
- This precedence can be overridden with parentheses, as for arithmetic

```
7.0 > 4 and -1 >= 0    # equivalent to True and False
```

```
False
```

```
5 < 4 or 1 != 2    # equivalent to False or True
```

```
True
```

```
not 7.5 < 0.9 or 4 == 4
```

```
True
```

```
not (7.5 < 0.9 or 4 == 4)
```

```
False
```

Logical Operators

- In a logic test expression, it is not always necessary to make an explicit comparison to obtain a boolean value
- Python will try to convert an object to a bool type if needed

```
a = 0
a or 4 < 3     # same as: False or 4 < 3
```

```
False
```

```
not a + 1     # same as: not True
```

```
False
```

  - In the last example, addition has higher precedence than the operator not, so a+1 is evaluated first to give 1, which corresponds to boolean True, so the whole expression equals to not True
- To explicitly convert an object to a boolean object, use the bool() function:

```
bool(-1)
```

```
True
```

Short Circuiting

- The **and** and **or** operators always return one of their operands and not just its bool equivalent:

```
a = 0
a+2 or 4 == 4
```

```
2
```

- Logic expressions are evaluated left to right, and those involving **and** or **or** are *short-circuited:* the second expression is only evaluated if necessary to decide the truth value of the whole expression

```
4 > 3 and a-2
```

```
-2
```

  - 4 > 3 is True, so the second expression must be evaluated to establish the truth of the and condition. a−2 is equal to −2, which is also equivalent to True, so the and condition is fulfilled and −2 (as the most recently evaluated expression) is returned.

Short Circuiting

- A common Python idiom is to assign a variable using the return value of a logic expression:

```
a = 0
b = a or 5
b
```

```
5
```

Exercise (9)

- Predict the results of the following expressions and check them in Jupyter Notebook:

```
1 < 2 or 4 < 2
not 1 < 2 or 4 < 2
not (1 < 2 or 4 < 2)
4 > 2 or 10/0 == 0
not 0 < 2
(not 0) < 2
1 and 2
0 and 1
1 or 0
```

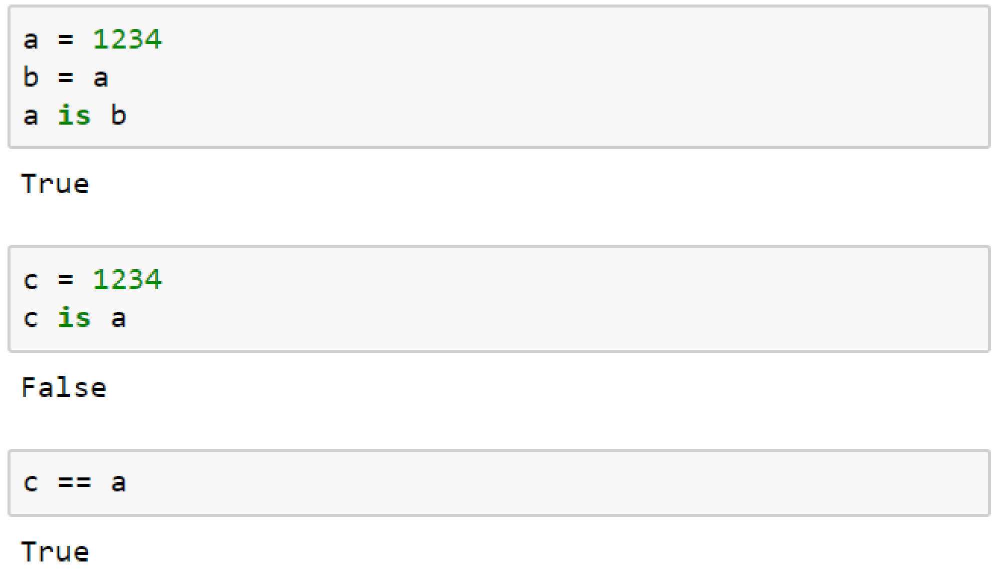
Exercise (10)

- There is no XOR (exclusive-or) operator provided “out of the box” by Python, but one can be constructed from the existing operators
- Devise two different ways of doing this
- The truth table for the xor operator is:

P	Q	P xor Q
False	False	False
False	True	True
True	False	True
True	True	False

Identity Operator

- The **is** operator is used to check if two variables refer to the same object:



```
a = 1234
b = a
a is b
```

```
True
```

```
c = 1234
c is a
```

```
False
```

```
c == a
```

```
True
```

- Here, the assignment c = 1234 creates an entirely new integer object, so c **is** a evaluates as False, even though a and c have the same *value*

Identity Operator

- Python also provides the operator **is not**, which returns true if the two variables don’t refer to the same object
- It is more natural to use c is not a, than not c is a

```
a = 8
b = a
b is a
```

```
True
```

```
b /= 2
b is not a
```

```
True
```

Identity Operator

- Given the previous discussion, the following result might come out as a surprise:

```
a = 250
b = 250
a is b
```

```
True
```

- This happens because Python keeps a **cache** of commonly used, small integer objects in order to improve performance
  - typically, the numbers –5-256
- The assignment a = 250 attaches the variable name a to the existing integer object without having to allocate new memory for it
- Because the same thing happens with b, the two variables point to the same object

Identity Operator

- The identity operator in combination with the type() built-in function can be used to check the type of an object:

```
x = 5.2
type(x) is float
```

```
True
```

```
type(x) is int
```

```
False
```

Special Value None

- Python defines a single value, **None**, of the special type, **NoneType**
- It is used to represent the absence of a defined value
  - For example, where no value is possible or relevant
  - This is particularly helpful in avoiding arbitrary default values for bad or missing data
- Comparisons to singletons like None should always be done with is or is not, and not with the equality operators
  - is always compares by object identity, while the result of == depends on the type of the operands

```
my_var = None
my_var is None
```

```
True
```

Exercise (11)

- Predict the results of the following expressions and check them in Jupyter Notebook:

```
a, b = 5, 5
a == b
```

```
a is b
```

```
a * 100 == b * 100
```

```
a * 100 is b * 100
```

```
a + 0.0 is a
```

```
a = None
b = None
a == b
```

```
a is b
```

```
a > b
```

Strings

- A string object (of type str) is an ordered, immutable sequence of characters
- To define a variable containing some constant text (a *string literal*), enclose the text in either single or double quotes:

```
str1 = 'String in single quotes'
str1
```

```
'String in single quotes'
```

```
str2 = "String in double quotes"
str2
```

```
'String in double quotes'
```

  - Inside the Python Shell a string is always displayed using single quotation marks
- However, if you use the print() function only contents of the string is displayed:

```
print(str1)
print(str2)
```

```
String in single quotes
String in double quotes
```

Strings

- Double quotes comes in handy when you have single quotation marks inside a string
- For example:

```
print("I'm learning Python")
```

```
I'm learning Python
```

- If we had used the single quotes, we would get a SyntaxError:

```
print('I'm learning Python')
```

```
  File "<ipython-input-11-45cd6c952520>", line 1
    print('I'm learning Python')
             ^
SyntaxError: invalid syntax
```

- Similarly, If you want to print double quotes inside a string, just wrap the entire string inside single quotes instead of double quotes

Strings

- Some languages like C, C++, Java treats a single character as a special type called char, but in Python a single character is also a string:

```
ch = 'a' # a string containing a single character
type(ch)
```

```
str
```

```
type("a string") # a string containing multiple characters
```

```
str
```

String Concatenation

- Strings can be concatenated using either the + operator or by placing them next to each other on the same line:

```
"abc" + "def"
```

```
'abcdef'
```

```
'one ' 'two ' 'three'
```

```
'one two three'
```

- What would happen if one of the operand is not a string?

```
s = "Python" + 101
```

```
---------------------------------------------------------------------------
----
TypeError                                 Traceback (most recent call l
ast)
<ipython-input-19-cd2000b866bc> in <module>()
----> 1 s = "Python" + 101

TypeError: must be str, not int
```

  - Python is a strongly typed language, thus it doesn't convert data of one type to a different type automatically

String Repetition Operator (*)

- When used with strings the * operator repeats the string *n* number of times

```
5 * 'a'
```

```
'aaaaa'
```

```
"-0-" * 5
```

```
'-0--0--0--0--0-'
```

```
print("We have got some", "spam" * 5)
```

```
We have got some spamspamspamspamspam
```

- Strings concatenated with the '+' operator can repeated with '*', but only if enclosed in parentheses:

```
('a' * 4 + 'B')*3
```

```
'aaaaBaaaaBaaaaB'
```

Long Strings

- To break up a long string over two or more lines of code, use the line continuation character, '\' or (better) enclose the string literal in parentheses:

```python
long_str = 'We hold these truths to be self-evident,'\
           ' that all men are created equal...'
long_str
```

```
'We hold these truths to be self-evident, that all men are
created equal...'
```

```python
long_str = ('We hold these truths to be self-evident,'
            ' that all men are created equal...')
long_str
```

```
'We hold these truths to be self-evident, that all men are
created equal...'
```

Escape Sequences

- Escape sequences are a set of special characters used to print characters which can't be typed directly using the keyboard
- Each escape sequence starts with a backslash (\ ) character
- Common Python escape sequences:

Escape Sequence	Meaning
\n	Line Feed (LF)
\r	Carriage return (CR)
\t	Horizontal tab
\b	Backspace
\'	Single quote
\"	Double quote
\\	The backslash character itself
\u, \U, \N	Unicode character
\x	Hex-encoded byte

Escape Sequences

- For example, \t inside a string prints a tab character (four spaces)

```
s = "Name\tAge\tGrades"
s
```

```
'Name\tAge\tGrades'
```

```
print(s)
```

```
Name	Age	Grades
```

  - Note that just typing a variable's name at the Python shell prompt simply echoes its literal value back to you
- \n character inside the string prints a newline character

```
s = "One\nTwo\nThree"
print(s)
```

```
One
Two
Three
```

Escape Sequences

- You can also use the \' and \" escape sequences to print a single or a double quotation marks in a string

```
print('I\'m learning Python')
```

```
I'm learning Python
```

```
print("John says \"Hello there\"")
```

```
John says "Hello there"
```

- Similarly to print a single backslash character \ you need to double it \\

```
path = "C:\\Users\\Roi"
print(path)
```

```
C:\Users\Roi
```

Raw Strings

- If you want to define a string to include character sequences such as '\n' without them being escaped, define a **raw string** prefixed with **r**:

```
rawstring = r'The escape sequence for a new line is \n'
rawstring
```

```
'The escape sequence for a new line is \\n'
```

```
print(rawstring)
```

```
The escape sequence for a new line is \n
```

Triple-Quoted Strings

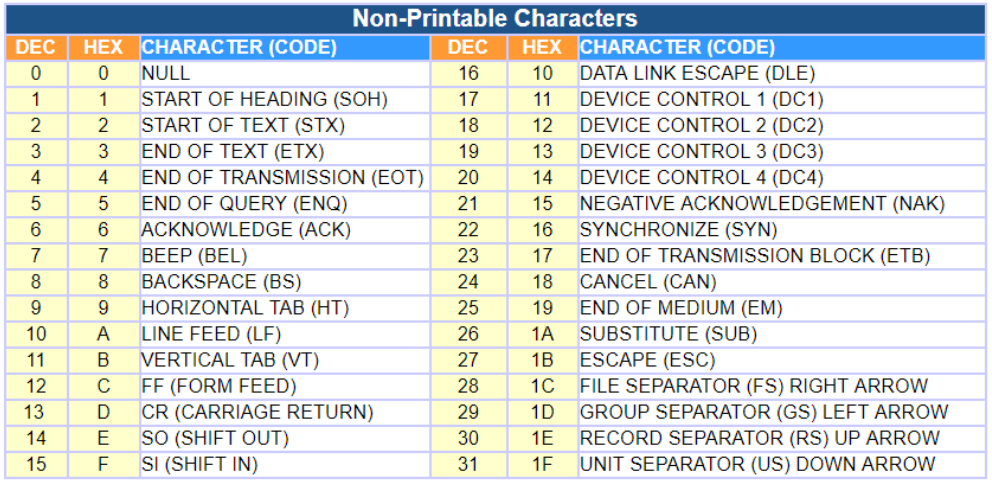
- When defining a block of text including several line endings it is often inconvenient to use \n repeatedly
- This can be avoided by' using **triple-quoted strings**: newlines defined within strings delimited by """ and ''' are preserved in the string

```
a = """one
two
three"""
print(a)
```

```
one
two
three
```

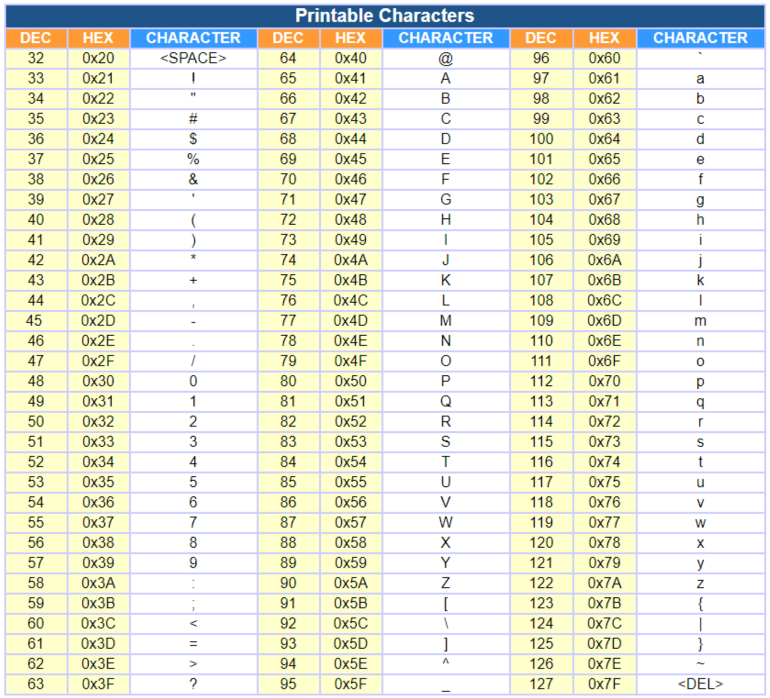
ASCII

- **ASCII** - The **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange is a standard seven-bit code that consists of 128 decimal numbers assigned to letters, numbers, punctuation marks, and the most common special characters
- **ASCII code** is the numerical representation of a character



Non-Printable Characters

DEC	HEX	CHARACTER (CODE)	DEC	HEX	CHARACTER (CODE)
0	0	NULL	16	10	DATA LINK ESCAPE (DLE)
1	1	START OF HEADING (SOH)	17	11	DEVICE CONTROL 1 (DC1)
2	2	START OF TEXT (STX)	18	12	DEVICE CONTROL 2 (DC2)
3	3	END OF TEXT (ETX)	19	13	DEVICE CONTROL 3 (DC3)
4	4	END OF TRANSMISSION (EOT)	20	14	DEVICE CONTROL 4 (DC4)
5	5	END OF QUERY (ENQ)	21	15	NEGATIVE ACKNOWLEDGEMENT (NAK)
6	6	ACKNOWLEDGE (ACK)	22	16	SYNCHRONIZE (SYN)
7	7	BEEP (BEL)	23	17	END OF TRANSMISSION BLOCK (ETB)
8	8	BACKSPACE (BS)	24	18	CANCEL (CAN)
9	9	HORIZONTAL TAB (HT)	25	19	END OF MEDIUM (EM)
10	A	LINE FEED (LF)	26	1A	SUBSTITUTE (SUB)
11	B	VERTICAL TAB (VT)	27	1B	ESCAPE (ESC)
12	C	FF (FORM FEED)	28	1C	FILE SEPARATOR (FS) RIGHT ARROW
13	D	CR (CARRIAGE RETURN)	29	1D	GROUP SEPARATOR (GS) LEFT ARROW
14	E	SO (SHIFT OUT)	30	1E	RECORD SEPARATOR (RS) UP ARROW
15	F	SI (SHIFT IN)	31	1F	UNIT SEPARATOR (US) DOWN ARROW

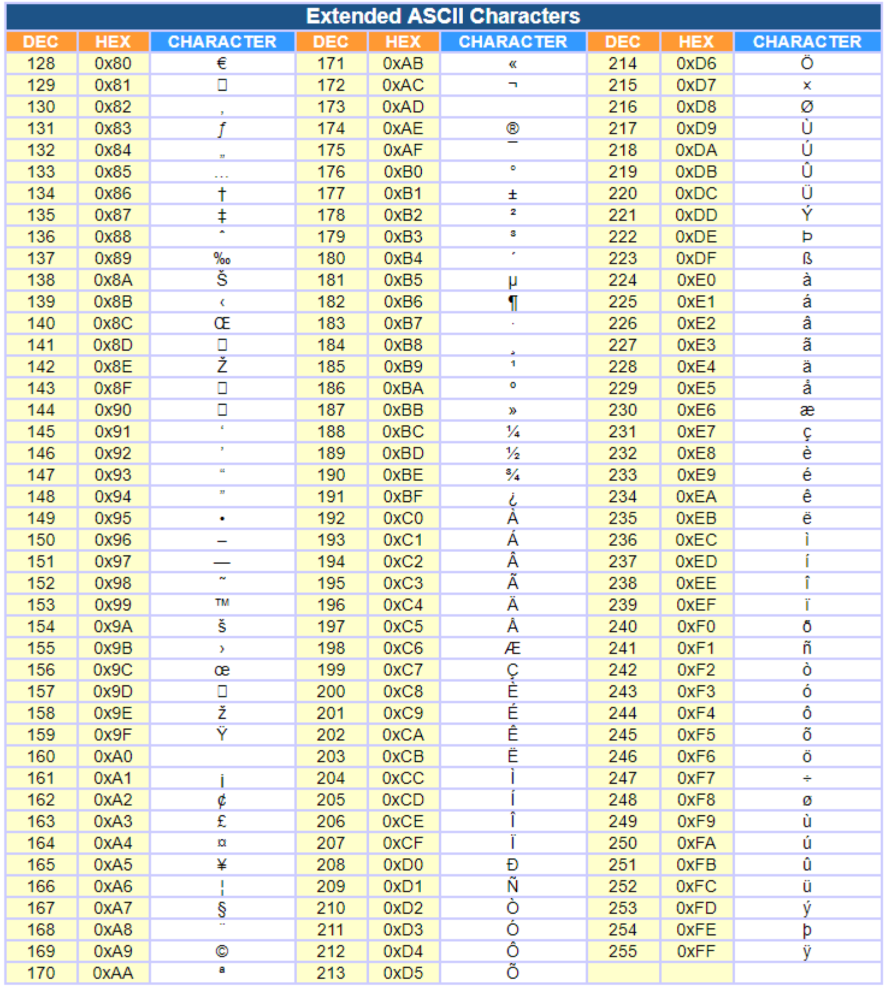


Printable Characters

DEC	HEX	CHARACTER	DEC	HEX	CHARACTER	DEC	HEX	CHARACTER
32	0x20	<SPACE>	64	0x40	@	96	0x60	`
33	0x21	!	65	0x41	A	97	0x61	a
34	0x22	"	66	0x42	B	98	0x62	b
35	0x23	#	67	0x43	C	99	0x63	c
36	0x24	$	68	0x44	D	100	0x64	d
37	0x25	%	69	0x45	E	101	0x65	e
38	0x26	&	70	0x46	F	102	0x66	f
39	0x27	'	71	0x47	G	103	0x67	g
40	0x28	(	72	0x48	H	104	0x68	h
41	0x29	)	73	0x49	I	105	0x69	i
42	0x2A	*	74	0x4A	J	106	0x6A	j
43	0x2B	+	75	0x4B	K	107	0x6B	k
44	0x2C	,	76	0x4C	L	108	0x6C	l
45	0x2D	-	77	0x4D	M	109	0x6D	m
46	0x2E	.	78	0x4E	N	110	0x6E	n
47	0x2F	/	79	0x4F	O	111	0x6F	o
48	0x30	0	80	0x50	P	112	0x70	p
49	0x31	1	81	0x51	Q	113	0x71	q
50	0x32	2	82	0x52	R	114	0x72	r
51	0x33	3	83	0x53	S	115	0x73	s
52	0x34	4	84	0x54	T	116	0x74	t
53	0x35	5	85	0x55	U	117	0x75	u
54	0x36	6	86	0x56	V	118	0x76	v
55	0x37	7	87	0x57	W	119	0x77	w
56	0x38	8	88	0x58	X	120	0x78	x
57	0x39	9	89	0x59	Y	121	0x79	y
58	0x3A	:	90	0x5A	Z	122	0x7A	z
59	0x3B	;	91	0x5B	[	123	0x7B	{
60	0x3C	<	92	0x5C	\	124	0x7C	|
61	0x3D	=	93	0x5D	]	125	0x7D	}
62	0x3E	>	94	0x5E	^	126	0x7E	~
63	0x3F	?	95	0x5F	_	127	0x7F	<DEL>

Extended ASCII

- **Extended ASCII** character encodings are eight-bit encodings that include the standard seven-bit ASCII characters, plus additional characters
- There are many extended ASCII encodings, that support different human languages



Extended ASCII Characters

DEC	HEX	CHARACTER	DEC	HEX	CHARACTER	DEC	HEX	CHARACTER
128	0x80	€	171	0xAB	«	214	0xD6	Ö
129	0x81	□	172	0xAC	¬	215	0xD7	×
130	0x82	‚	173	0xAD		216	0xD8	Ø
131	0x83	ƒ	174	0xAE	®	217	0xD9	Ù
132	0x84	„	175	0xAF	¯	218	0xDA	Ú
133	0x85	…	176	0xB0	°	219	0xDB	Û
134	0x86	†	177	0xB1	±	220	0xDC	Ü
135	0x87	‡	178	0xB2	²	221	0xDD	Ý
136	0x88	ˆ	179	0xB3	³	222	0xDE	Þ
137	0x89	‰	180	0xB4	´	223	0xDF	ß
138	0x8A	Š	181	0xB5	µ	224	0xE0	à
139	0x8B	‹	182	0xB6	¶	225	0xE1	á
140	0x8C	Œ	183	0xB7	·	226	0xE2	â
141	0x8D	□	184	0xB8	¸	227	0xE3	ã
142	0x8E	Ž	185	0xB9	¹	228	0xE4	ä
143	0x8F	□	186	0xBA	º	229	0xE5	å
144	0x90	□	187	0xBB	»	230	0xE6	æ
145	0x91	‘	188	0xBC	¼	231	0xE7	ç
146	0x92	’	189	0xBD	½	232	0xE8	è
147	0x93	“	190	0xBE	¾	233	0xE9	é
148	0x94	”	191	0xBF	¿	234	0xEA	ê
149	0x95	•	192	0xC0	À	235	0xEB	ë
150	0x96	–	193	0xC1	Á	236	0xEC	ì
151	0x97	—	194	0xC2	Â	237	0xED	í
152	0x98	˜	195	0xC3	Ã	238	0xEE	î
153	0x99	™	196	0xC4	Ä	239	0xEF	ï
154	0x9A	š	197	0xC5	Å	240	0xF0	ð
155	0x9B	›	198	0xC6	Æ	241	0xF1	ñ
156	0x9C	œ	199	0xC7	Ç	242	0xF2	ò
157	0x9D	□	200	0xC8	È	243	0xF3	ó
158	0x9E	ž	201	0xC9	É	244	0xF4	ô
159	0x9F	Ÿ	202	0xCA	Ê	245	0xF5	õ
160	0xA0		203	0xCB	Ë	246	0xF6	ö
161	0xA1	¡	204	0xCC	Ì	247	0xF7	÷
162	0xA2	¢	205	0xCD	Í	248	0xF8	ø
163	0xA3	£	206	0xCE	Î	249	0xF9	ù
164	0xA4	¤	207	0xCF	Ï	250	0xFA	ú
165	0xA5	¥	208	0xD0	Ð	251	0xFB	û
166	0xA6	¦	209	0xD1	Ñ	252	0xFC	ü
167	0xA7	§	210	0xD2	Ò	253	0xFD	ý
168	0xA8	¨	211	0xD3	Ó	254	0xFE	þ
169	0xA9	©	212	0xD4	Ô	255	0xFF	ÿ
170	0xAA	ª	213	0xD5	Õ			

ASCII in Python

- The **\x** escape denotes a character encoded by the single-byte hex value given by the subsequent two characters
- For example, the capital letter ‘N’ has the value 78, which is 4E in hex, thus:

```
print('\x4e')
```

```
N
```

- The **ord()** function returns the ASCII value of a character and the **chr()** function returns the character represented by the ASCII value:

```
ord("a")   # the ASCII value of character a
```

```
97
```

```
chr(65)   # the character represented by ASCII value 65
```

```
'A'
```

Exercise (12)

- Ask the user to enter a small English letter (a-z)
- Print its corresponding capital letter, e.g. K for k
- Ask the user to enter another character
- Print if the second character is a digit or not (i.e., one of the characters 0-9)

Unicode

- Python 3 strings are composed of *Unicode* characters
- Unicode is a standard describing the representation of more than 140,000 characters in almost every human language as well as many other special characters
- Unicode assigns a number (**code point**) to every character
- These code points can be implemented as byte sequences by different character encodings (e.g., UTF-8, UTF-16, UTF-32)
- By default, Python 3 uses the **UTF-8** encoding, which is the most widely used today
- UTF-8 uses one byte for the first 128 code points, and up to 4 bytes for the others
    - The first 128 UTF-8 code points are the ASCII characters, which means that any ASCII text is also a UTF-8 text
- For a list of code points, see the official Unicode website's code charts at http://www.unicode.org/charts/

Unicode Characters

- If your editor doesn't allow you to enter a Unicode character, you can use its 16- or 32-bit hex value or its Unicode character name:

```
'\u00E9'    # 16-bit hex value
```

```
'é'
```

```
'\U000000E9'   # 32-bit hex value
```

```
'é'
```

```
'\N{LATIN SMALL LETTER E WITH ACUTE}'   # by name
```

```
'é'
```

- Python even supports Unicode variable names:

```
Σ = 4
Σ
```

```
4
```

  - This is mostly a bad idea, because of the difficulty in entering non-ASCII characters from a standard keyboard

Unicode Characters

- **ord**(*c*) returns an integer representing the Unicode code point of the character *c*

```
print(ord('a'))
print(ord('€'))
```

```
97
8364
```

- **chr**(*i*) returns the string representing a character whose Unicode code point is the integer *i* (the inverse of ord)
  - The valid range for the argument is from 0 through 1,114,111 (0x10FFFF in base 16)

```
print(chr(97))
print(chr(8364))
```

```
a
€
```

String Length

- The **len()** built-in function counts the number of characters in the string
  - If the string is in Unicode, len() returns the number of Unicode characters

```
len("hello world")
```

```
11
```

```
s = "הוא אוסף אותי מחר בשלוש וחצי"
len(s)
```

```
28
```

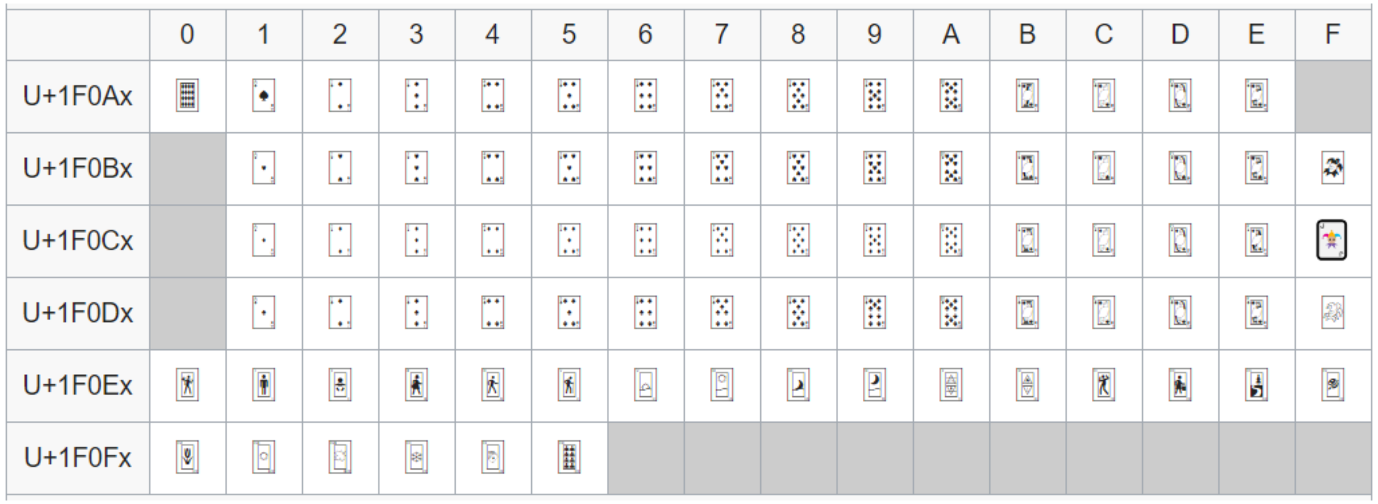
- If you want to know the number of bytes needed to store the string in memory, you first have to convert it into a Python byte string using the **encode()** method

```
len(s.encode('utf-8'))
```

```
51
```

Exercise (13)

- Ask the user to enter a card number between 1 and 14
- Print the symbol of the corresponding card from the Spades suite
- The Unicode characters of the symbol cards is given in the following table:



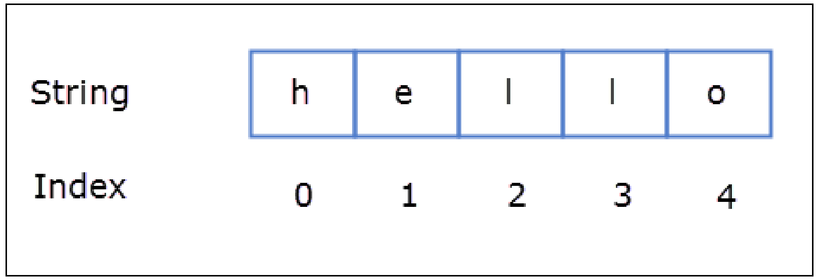
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
U+1F0Ax	🂠	🂡	🂢	🂣	🂤	🂥	🂦	🂧	🂨	🂩	🂪	🂫	🂬	🂭	🂮	
U+1F0Bx		🂱	🂲	🂳	🂴	🂵	🂶	🂷	🂸	🂹	🂺	🂻	🂼	🂽	🂾	🂿
U+1F0Cx		🃁	🃂	🃃	🃄	🃅	🃆	🃇	🃈	🃉	🃊	🃋	🃌	🃍	🃎	🃏
U+1F0Dx		🃑	🃒	🃓	🃔	🃕	🃖	🃗	🃘	🃙	🃚	🃛	🃜	🃝	🃞	🃟
U+1F0Ex	🃠	🃡	🃢	🃣	🃤	🃥	🃦	🃧	🃨	🃩	🃪	🃫	🃬	🃭	🃮	🃯
U+1F0Fx	🃰	🃱	🃲	🃳	🃴	🃵										

- For example:

```
Enter a card number: 5
🂥
```

Indexing Strings

- **Indexing** (or “subscripting”) a string returns a single character at a given location
- An index refers to the position of a character inside a string
- Like all sequences in Python, strings are zero-indexed
  - which means that the first character is at index 0, and the final character in a string consisting of *n* characters is at index *n* – 1



- To access a character at index *i* in a string *str*, write the index number of the character inside square brackets [], like this: str[i]
  - The character is returned in a str object of length 1

Indexing Strings

- Examples:

```
s = "hello"
```

```
s[0] # get the first character
```

```
'h'
```

```
s[1] # get the second character
```

```
'e'
```

```
s[len(s) - 1] # get the last (fifth) character
```

```
'o'
```

- The last valid index for string s is 4
- Trying to access characters beyond the last valid index will raise an IndexError

Negative Indexes

- We can also use negative indexes to access characters from the end of the string
- Negative index start from -1, so the index position of the last character is -1, and the index position of the first character is –*n* (for a string of length *n*)

```python
s = "markdown"
```

```python
s[-1] # get the last character
```

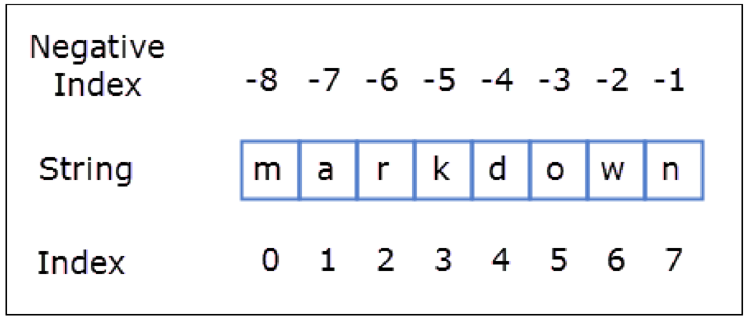
```
'n'
```

```python
s[-2] # get the second last character
```

```
'w'
```

```python
s[-len(s)] # get the first character
```

```
'm'
```



Membership Operators

- **in** or **not in** operators are used to check if a string contains a given substring:

```
str1 = "object oriented"
```

```
"ted" in str1 # Does "ted" exist in str1?
```

```
True
```

```
"eject" in str1 # Does "eject" exist in str1?
```

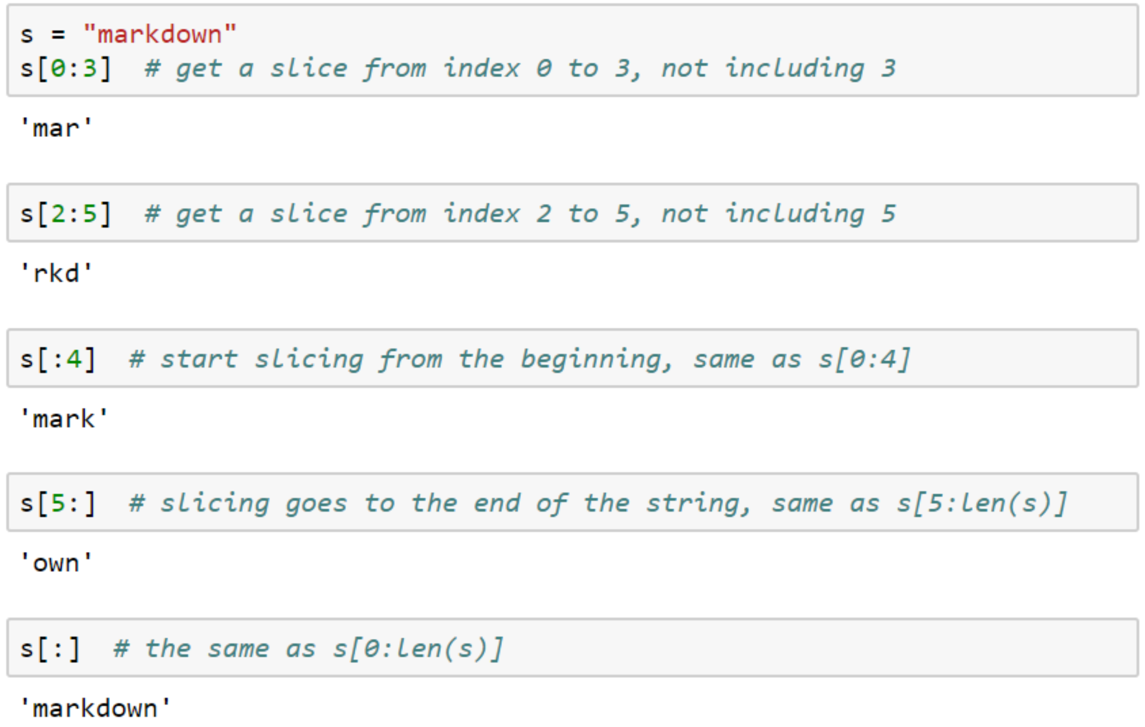
```
False
```

```
"orion" not in str1 # Does "orion" doesn't exist in str1?
```

```
True
```

Slicing Strings

- Slicing a string, **s[i:j]**, produces a substring of a string between the characters at two indexes, including the first (i) but excluding the second (j)
  - If the first index is omitted, 0 is assumed
  - If the second index is omitted, the string is sliced to its end



```
s = "markdown"
s[0:3]  # get a slice from index 0 to 3, not including 3
```

```
'mar'
```

```
s[2:5]  # get a slice from index 2 to 5, not including 5
```

```
'rkd'
```

```
s[:4]  # start slicing from the beginning, same as s[0:4]
```

```
'mark'
```

```
s[5:]  # slicing goes to the end of the string, same as s[5:len(s)]
```

```
'own'
```

```
s[:]  # the same as s[0:len(s)]
```

```
'markdown'
```

Slicing Strings

- Unlike indexing, slicing a string outside its bounds does not raise an error:

```
s[2:len(s)+10]
```

```
'rkdown'
```

```
s[10:]
```

```
''
```

- We can also use negative index in string slicing:

```
s[1:-1]  # slice the string from index 1 to index -1, not including -1
```

```
'arkdow'
```

Slicing Strings

- The optional, third number in a slice specifies the *stride*
  - If omitted the default is 1: return every character in the requested range
  - To return every kth letter, set the stride to k
  - Negative values of k reverse the string

```
s = 'Markdown'
s[2:6:2]
```

```
'rd'
```

```
s[::2]
```

```
'Mrdw'
```

```
s[-1:4:-1]  # take the characters from the last (index-1)
            # down to (but not including) character at index 4
            # with stride -1 (select every character in reverse direction)
```

```
'nwo'
```

```
s[::-1]   # reverse a string
```

```
'nwodkraM'
```

Exercise (14)

- Slice the string s='seehemewe' to produce the following substrings:
  - 'see'
  - 'he'
  - 'me'
  - 'we'
  - 'hem'
  - 'meh'
  - 'wee'

Exercise (15)

- Get a string from the user and prints whether it is a palindrome
- A palindrome is a string that reads the same forward as backward
  - e.g., level, radar, racecar, madam, noon, civic
- Use a single-line expression for determining if the string is a palindrome

String Methods

- String objects come with a large number of methods for manipulating and transforming
- These methods are accessed using the usual dot notation we’ve met already
- They can be grouped into the following categories:
  - Testing strings
  - Searching for a substring inside a string
  - Formatting strings
  - Converting strings

Testing Strings

- Methods for testing strings:

Method	Description
isalpha()	Returns True if all characters in the string are alphabetic; otherwise return False.
isdigit()	Returns True if all characters in the string are digits; otherwise return False.
isalnum()	Returns True if all characters in the string are alphanumeric (digits or alphabets); otherwise return False.
islower()	Returns True if all the characters in the string are in lowercase; otherwise return False.
isupper()	Returns True if all the characters in the string are in uppercase; otherwise return False.
isspace()	Returns True if all the characters in the string are whitespace characters ; otherwise return False.

Testing Strings – Examples

```
"hello".isalpha()
```

```
True
```

```
"abc123".isalpha()
```

```
False
```

```
"2048".isdigit()
```

```
True
```

```
"101.29".isdigit()
```

```
False
```

```
"Abc123".isalnum()
```

```
True
```

```
"$$$".isalnum()
```

```
False
```

```
"abc".islower()
```

```
True
```

```
s = "A bite of python"
s.islower()
```

```
False
```

```
s.isupper()
```

```
False
```

```
"\n\t".isspace()
```

```
True
```

```
"1 2 3".isspace()
```

```
False
```

Searching in Strings

- Methods that allows you to search for a substring inside a string:

Method	Description
endswith(*suffix*)	Returns True if the string ends with the substring *suffix*; otherwise return False.
startswith(*prefix*)	Returns True if the string starts with the substring *prefix*; otherwise return False.
find(*substring*)	Returns the lowest index in the string where *substring* is found. If *substring* is not found return -1.
rfind(*substring*)	Returns the highest index in the string where *substring* is found. If *substring* is not found return -1.
index(*substring*)	Returns the lowest index in the string where *substring* is found. If *substring* doesn't exist in the list, an exception is raised.
count(*substring*)	Returns the number of occurrences of *substring* found in the string. If no occurrence is found return 0.

Searching in Strings – Examples

```
s = "abc"
s.endswith("bc")
```

```
True
```

```
"python".startswith("py")
```

```
True
```

```
"Learning Python".find("n")
```

```
4
```

```
"Learning Python".find("at")
```

```
-1
```

```
"Learning Python".rfind("n")
```

```
14
```

```
"Learning Python".count("n")
```

```
3
```

Manipulating Strings

- The following methods are used to return a modified version of the string:

Method	Description
lower()	Returns a copy of the string with all characters in uppercase.
upper()	Returns a copy of the string with all characters in lowercase.
capitalize()	Returns a copy of the string after capitalizing only the first letter in the string.
title()	Returns a copy of the string with all words starting with capitals and other characters in lowercase.
lstrip([*chars*])	Returns a copy of the string with leading characters specified by [chars] removed. If [*chars*] is omitted, any leading whitespace is removed.
rstrip([*chars*])	Returns a copy of the string with trailing characters specified by [chars] removed. If [*chars*] is omitted, any trailing whitespace is removed.
strip([*chars*])	Returns a copy of the string with leading and trailing characters specified by [chars] removed. If [*chars*] is omitted, any leading and trailing whitespace is removed.
replace(old, new)	Returns a copy of the string with each substring *old* replaced with *new*.

Manipulating Strings – Examples

```python
"abcDEF".lower()
```

```
'abcdef'
```

```python
"abc".lower()
```

```
'abc'
```

```python
"ABCdef".upper()
```

```
'ABCDEF'
```

```python
"a long string".capitalize()
```

```
'A long string'
```

```python
"a long string".title()
```

```
'A Long String'
```

```python
s1 = "\n\tName\tAge"
print(s1)
```

```

        Name    Age
```

```python
s2 = s1.strip()
s2
```

```
'Name\tAge'
```

```python
print(s2)
```

```
Name    Age
```

```python
s = "--Name\tAge--"
s.lstrip("-")
```

```
'Name\tAge--'
```

```python
s1 = "Learning C"
s2 = s1.replace("C", "Python")
s2
```

```
'Learning Python'
```

Formatting Strings

- The following table lists some formatting methods of strings:

Method	Description
center(*width*)	Returns a copy of the string after centering it in a string with total number of characters *width*
ljust(*width*)	Returns a copy of the string after centering it in a string with total number of characters *width*
rjust(*width*)	Returns a copy of the string after centering it in a string with total number of characters *width*

```
"Name".center(10)
```

```
'   Name   '
```

```
"Name".ljust(10)
```

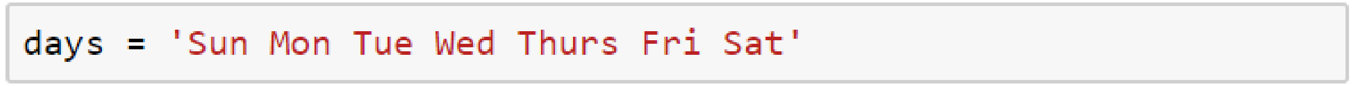
```
'Name      '
```

```
"Name".rjust(10)
```

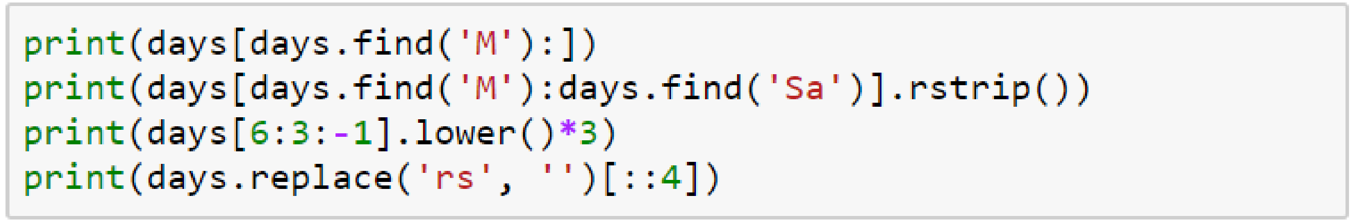
```
'      Name'
```

Exercise (16)

- Predict the results of the following statements and check them in Jupyter Notebook:



```python
days = 'Sun Mon Tue Wed Thurs Fri Sat'
```



```python
print(days[days.find('M'):])
print(days[days.find('M'):days.find('Sa')].rstrip())
print(days[6:3:-1].lower()*3)
print(days.replace('rs', '')[::4])
```

String Comparison

- We can compare strings using the relational operators
- Strings are compared using a **Lexicographical comparison**:
  - The strings are compared using the Unicode value of their corresponding characters
  - The comparison starts off by comparing the first character from both strings:
    - If they differ, the Unicode values of the corresponding characters are compared to determine the outcome of the comparison
    - If they are equal, the next two characters are compared
  - This process continues until either string is exhausted
  - If a short string appears at the start of another long string then the short string is smaller

```
"linker" > "linquish"
```

```
False
```

```
"ab" < "abc"
```

```
True
```

The print() Function

- print() is a built-in function takes a list of objects, and also the optional arguments:
  - **end** – specify which characters should end the string
  - **sep** – specify which characters should be used to separate the printed objects
  - Omitting these additional arguments prints the object fields separated by a *single space* and the line is ended with a *newline* character

```
ans = 6
print("Solve:", 2, "x =", ans, "for x")
```

```
Solve: 2 x = 6 for x
```

```
print("Solve: ", 2, "x = ", ans, " for x", sep="", end="!\n")
```

```
Solve: 2x = 6 for x!
```

- To suppress the newline at the end of a printed string, specify end=""

```
print("A line with no newline character.", end="")
print("Another line")
```

```
A line with no newline character.Another line
```

The print() Function

- print() can be used to create simple text tables:

```python
title = "|  Life Satisfaction Index |"
line = '+' + '-'*11 + '-'*15 + '+'
header = "|  Country   | Satisfaction |"

print(line,
      title,
      line,
      header,
      line,
      "| Australia |     7.3      |",
      "| Israel    |     7.2      |",
      "| Spain     |     6.4      |",
      line,
      sep="\n")
```

```
+--------------------------+
|  Life Satisfaction Index |
+--------------------------+
|  Country   | Satisfaction |
+--------------------------+
| Australia |     7.3      |
| Israel    |     7.2      |
| Spain     |     6.4      |
+--------------------------+
```

String Formatting

- It is possible to use a string's **format()** method to insert objects into it

```
"{} + {} = {}".format(2, 3, 5)
```

```
'2 + 3 = 5'
```

  - The format() method is called on the string literal with the arguments 2, 3 and 5, which are interpolated, in order, into the locations of the **replacement fields**, indicated by braces {}
- Replacement fields can also be numbered (starting at 0) or named
  - Helps with longer strings and allows for the same value to be interpolated more than once

```
"{1} + {0} = {2}".format(2, 3, 5)
```

```
'3 + 2 = 5'
```

```
"{num1} + {num2} = {answer}".format(num1=2, num2=3, answer=5)
```

```
'2 + 3 = 5'
```

```
"{0} + {0} = {1}".format(2, 2 + 2)
```

```
'2 + 2 = 4'
```

String Formatting

- Replacement fields can be given a minimum size within the string by the inclusion of an integer length after a colon as follows:

```
"=== {0:12} ===".format("Python")
```

```
'=== Python       ==='
```

  - If the string is too long for the minimum size, it will take up as many characters as needed
- The alignment of the interpolated string can be controlled with the single characters < (left), > (right) and ^ (center):

```
"=== {0:<12} ===".format("Python")
```

```
'=== Python       ==='
```

```
"=== {0:>12} ===".format("Python")
```

```
'===       Python ==='
```

```
"=== {0:^12} ===".format("Python")
```

```
'===    Python    ==='
```

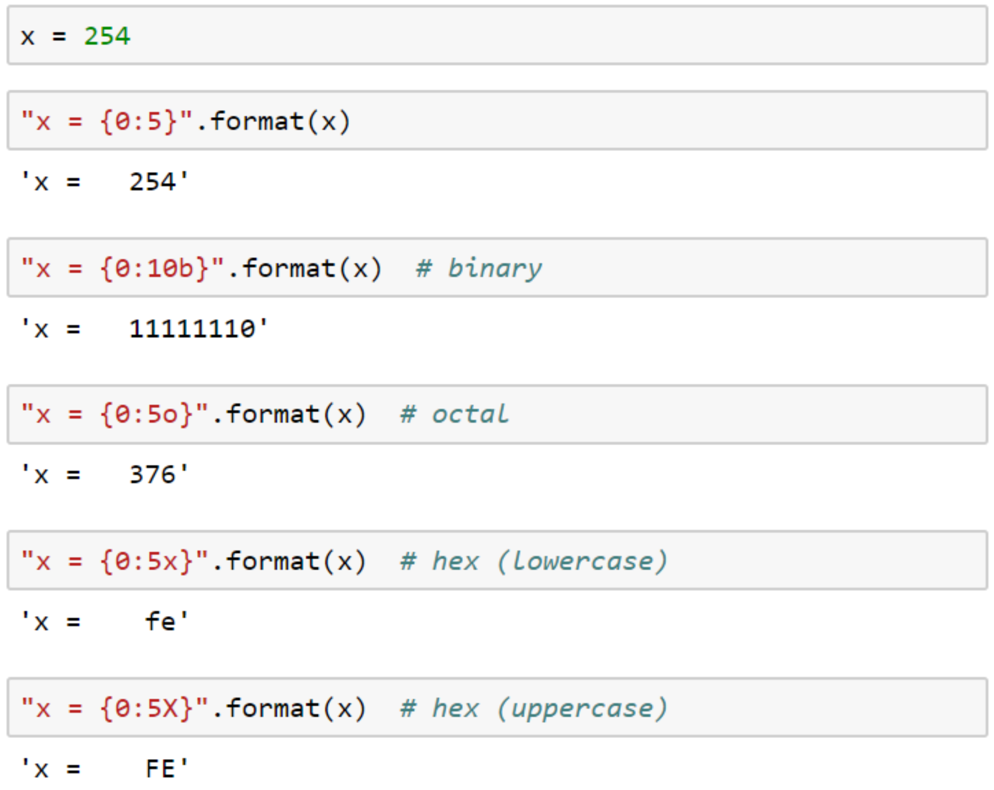
Exercise (17)

- What is the output of the following code? How does it work?

```
suff = "thstndrdthththththth"
n = 1
print("{}{}".format(n, suff[n*2:n*2+2]))
n = 3
print("{}{}".format(n, suff[n*2:n*2+2]))
n = 5
print("{}{}".format(n, suff[n*2:n*2+2]))
```

Formatting Numbers

- The string format() method provides a powerful way to format numbers
- The specifiers ‘d’, ‘b’, ‘o’, ‘x’, ‘X’ indicate a decimal, binary, octal, lowercase hex, and uppercase hex integer, respectively:



```
x = 254
```

```
"x = {0:5}".format(x)
```

```
'x =   254'
```

```
"x = {0:10b}".format(x)   # binary
```

```
'x =   11111110'
```

```
"x = {0:5o}".format(x)   # octal
```

```
'x =   376'
```

```
"x = {0:5x}".format(x)   # hex (lowercase)
```

```
'x =    fe'
```

```
"x = {0:5X}".format(x)   # hex (uppercase)
```

```
'x =    FE'
```

Formatting Numbers

- By default, all types of numbers are right aligned
- We can change the default alignment by using following the characters < (left justify) and > (right justify)

```
"x = {0:<5}".format(x)
```

```
'x = 254  '
```

```
"x = {0:>5}".format(x)
```

```
'x =   254'
```

- Numbers can be padded with zeros to fill out the specified field size by prefixing the minimum width with a 0:

```
"x = {0:05}".format(x)
```

```
'x = 00254'
```

Formatting Numbers

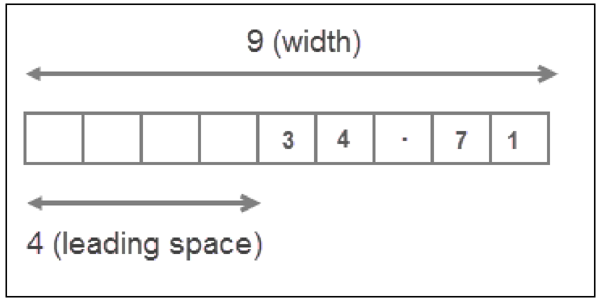
- To format floating point numbers use the following format specifier:

```
width.precisiont
```

  - **width** includes the digits before and after the decimal and the decimal point itself
  - **precision** is the number of decimal places after the decimal point
  - The t character following the precision is the type code or specifier. Most useful specifiers:
    - ‘f’: fixed-point notation
    - ‘e’/‘E’: exponent (i.e., “scientific” notation)
    - ‘g’/‘G’: a general format which uses scientific notation for very large and very small numbers
- Example:

```
a = 34.71248
"{0:9.2f}".format(a)
```

```
'    34.71'
```



Formatting Numbers

- We can also omit the width entirely, in which case it is automatically determined by the length of the value:

```
import math
"{0:.5f}".format(math.pi)
```

```
'3.14159'
```

- To format a number in Scientific Notation, just replace the type code from f to e or E:

```
"{0:10.3e}".format(5482.52291)
```

```
'  5.483e+03'
```

```
"{0:.2e}".format(0.0000212354)
```

```
'2.12e-05'
```

- You can separate thousands by commas by adding a comma , just after the width field or before the precision:

```
"{0:,.2f}".format(98813343817.71)
```

```
'98,813,343,817.71'
```

Exercise (18)

- The table that follows gives the names, symbols, values, uncertainties and units of some physical constants

Name	**Symbol**	**Value**	**Uncertainty**	**Units**
Boltzmann constant	k_B	$1.3806504 \times 10^{-23}$	2.4×10^{-29}	$\mathrm{J\,K^{-1}}$
Speed of light	c	299792458	(def)	$\mathrm{m\,s^{-1}}$
Planck constant	h	$6.62606896 \times 10^{-34}$	3.3×10^{-41}	$\mathrm{J\,s}$
Avogadro constant	N_A	$6.02214179 \times 10^{23}$	3×10^{16}	$\mathrm{mol^{-1}}$
Electron magnetic moment	μ_e	$-9.28476377 \times 10^{-24}$	2.3×10^{-31}	J/T
Gravitational constant	G	6.67428×10^{-11}	6.7×10^{-15}	$\mathrm{N\,m^2\,kg^{-2}}$

- Defining variables of the form:

```
kB = 1.3806504e-23 # J/K
kB_unc = 2.4e-29 # uncertainty
kB_units = "J/K"
```

Exercise (18) Cont.

- Use the string object's format() method to produce the following output:

```
kB = 1.381e-23 J/K

G = 0.0000000000667428 Nm^2/kg^2

c = 2.9979e+08 m/s

=== G = +6.67E-11 [Nm^2/kg^2] ===

=== µe = -9.28E-24 [ J/T] ===
```

  - Hint: the Unicode codepoint for the lowercase Greek letter mu is U+03BC

Exercise (19)

- Create the following table:

```
+------------------------------------------+
|           Cereal Yields (kg/ha)          |
+-------------+-------+-------+-------+-------+
|   Country   | 1980  | 1990  | 2000  | 2010  |
+-------------+-------+-------+-------+-------+
| China       | 2,937 | 4,321 | 4,752 | 5,527 |
| Germany     | 4,225 | 5,411 | 6,453 | 6,718 |
| United States | 3,772 | 4,755 | 5,854 | 6,988 |
+-------------+-------+-------+-------+-------+
```

Iterable Objects and Sequences

- **Iterable** is an object capable of returning its members one at a time
- Examples of iterables include all sequence types (such as list, str, and tuple) and some non-sequence types like dict, set, and file objects
- Objects of any classes you define with an **__iter__()** method, or with a **__getitem__()** method that implements Sequence semantics, are iterables
- Iterables can be used in a for loop and in many other places where a sequence is needed (e.g., list(), map(), zip())

- A **sequence** is an iterable which supports efficient element access using integer indices via the **__getitem__()** special method and defines a **__len__()** method that returns the length of the sequence
- Some built-in sequence types are list, str, tuple, range, and bytes

Lists

- A Python list is an ordered, *mutable* and dynamically sized array of objects
  - In some other languages (like C and Java) this data structure is called an array
  - However, while arrays are of sixed size, lists grow automatically as needed
- A list is constructed by specifying its objects, separated by commas, between square brackets [], as follows:

```
variable = [item1, item2, item3, ..., itemN]
```

- For example:

```
numbers = [5, 88, 17, 35]
```

- Note that just like everything else, a list is an object too, of type list

```
type(numbers)
```

```
list
```

  - The numbers variable only stores the address where the list object is actually stored in memory

Lists

- To print the contents of a list just type the list name, or use the print() function:

```
numbers
```

```
[5, 88, 17, 35]
```

```
print(numbers)
```

```
[5, 88, 17, 35]
```

- A list can contain elements of different types:

```
mixed = ["a string", 3.14, [1, 2, 3], True]
mixed
```

```
['a string', 3.14, [1, 2, 3], True]
```

- To create an empty list simply type square brackets [] without any elements inside it
  - e.g., list1 = []

Multi-Dimensional Lists

- The elements of a list can be lists themselves:

```
matrix = [
    [37, 88, 25],  # first row
    [99, 31, 64]   # second row
]
matrix
```

```
[[37, 88, 25], [99, 31, 64]]
```

  - matrix contains two elements of type list

Indexing

- The elements in a list are zero-indexed (like strings)
- That means that the first element is at index 0, second is at 1, third is at 2 and so on
- The last valid index will be one less than the length of the list
- We use the following syntax to access an element from the list: `my_list[index]`
- Examples:

```
list1 = [5, 24, 3.14, 16.5, 35]
list1[0]  # get the first element
```

```
5
```

```
list1[1]  # get the second element
```

```
24
```

```
list1[len(list1) - 1]  # get the last element
```

```
35
```

Indexing

- Trying to access an element beyond the last valid index will result in an IndexError:

```
list1[10]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-18-1a2e5d318e75> in <module>()
----> 1 list1[10]

IndexError: list index out of range
```

- Just like strings, we can use negative indexes to access elements from the list end
  - Negative indexes start from -1

```
list1[-1]  # get the last element
```

```
35
```

```
list1[-2]  # get the second last element
```

```
16.5
```

```
list1[-len(list1)]  # get the first element
```

```
5
```

Membership Operators

- Just like strings, we can check whether an element exists in the list or not using the **in** and **not in** operators:

```
list1 = [3, "hello", True, 5.3]
5.3 in list1
```

```
True
```

```
"Hello" in list1
```

```
False
```

```
"joker" not in list1
```

```
True
```

Lists and Mutability

- Lists are **mutable**, i.e., we can modify a list without creating a new list in the process
  - Unlike strings, which cannot be altered once defined
- For example, the items of a list can be reassigned:

```
list1 = ["str", "list", "int", "float"]
```

```
id(list1)   # the address where the list is stored
```

```
2902477589384
```

```
list1[0] = "string"   # update element at index 0
```

```
list1   # list1 has changed
```

```
['string', 'list', 'int', 'float']
```

```
id(list1)  # the id remains the same
```

```
2902477589384
```

Lists and Mutability

- Let's examine another example:

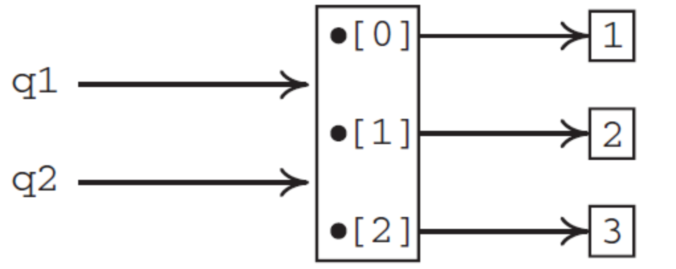
```
q1 = [1, 2, 3]
q2 = q1
```

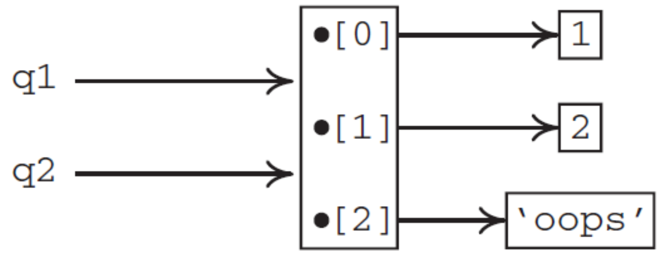
```
q1[2] = "oops"
q1
```

```
[1, 2, 'oops']
```

```
q2
```

```
[1, 2, 'oops']
```





  - The variables q1 and q2 refer to the *same list*, stored in the same memory location
  - Because lists are mutable, the line q1[2] = "oops" actually changes one of the stored values at that location
  - q2 still points to the same location and so it appears to have changed as well

Slicing Lists

- Lists can be sliced in the same way as string sequences:

```
list1 = [0, 0.1, 0.2, 0.3, 0.4, 0.5]
list1[1:4]
```

```
[0.1, 0.2, 0.3]
```

```
list1[::-1]  # return a reversed copy of the list
```

```
[0.5, 0.4, 0.3, 0.2, 0.1, 0]
```

```
list1[1::2]  # striding: returns elements at 1,3,5
```

```
[0.1, 0.3, 0.5]
```

- Taking a slice *copies the data* into a new list:

```
list2 = list1[1:4]
list2[1] = 999  # only affects list2
list2
```

```
[0.1, 999, 0.3]
```

```
list1
```

```
[0, 0.1, 0.2, 0.3, 0.4, 0.5]
```

List Concatenation

- Like strings, lists can be joined using the + operator, which creates a new list containing the elements from both lists:

```
list1 = [1, 2, 3]
list2 = [11, 22, 33]
list3 = list1 + list2
list3
```

```
[1, 2, 3, 11, 22, 33]
```

- Another way to concatenate lists is to use the += operator, which modifies the existing list instead of creating a new one:

```
id(list1)
```

```
2902478184776
```

```
list1 += list2    # append list2 to list1
list1
```

```
[1, 2, 3, 11, 22, 33]
```

```
id(list1)  # the address remains the same
```

```
2902478184776
```

Repetition Operator

- We can use the * operator with lists too
- The operator replicates the list:

```
actions = ["eat", "sleep", "repeat"]
daily_life = actions * 4
daily_life
```

```
['eat',
 'sleep',
 'repeat',
 'eat',
 'sleep',
 'repeat',
 'eat',
 'sleep',
 'repeat',
 'eat',
 'sleep',
 'repeat']
```

- This operator is useful for creating an empty list with a specific size, e.g.:

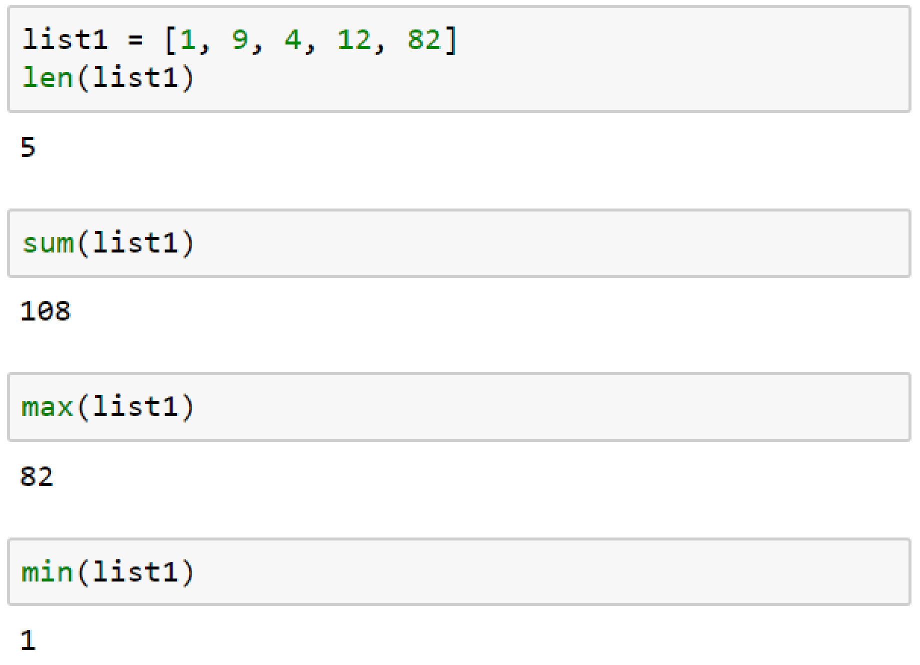
```
list1 = [None] * 10
list1
```

```
[None, None, None, None, None, None, None, None, None, None]
```

List Built-in Functions

- The following table lists some functions, commonly used while working with lists:

Function	**Description**
len(*list*)	Returns the number of elements in *list*
sum(*list*)	Returns the sum of elements in the *list*
max(*list*)	Returns the element with the greatest value in *list*
min(*list*)	Returns the element with the smallest value in *list*



```
list1 = [1, 9, 4, 12, 82]
len(list1)
```

```
5
```

```
sum(list1)
```

```
108
```

```
max(list1)
```

```
82
```

```
min(list1)
```

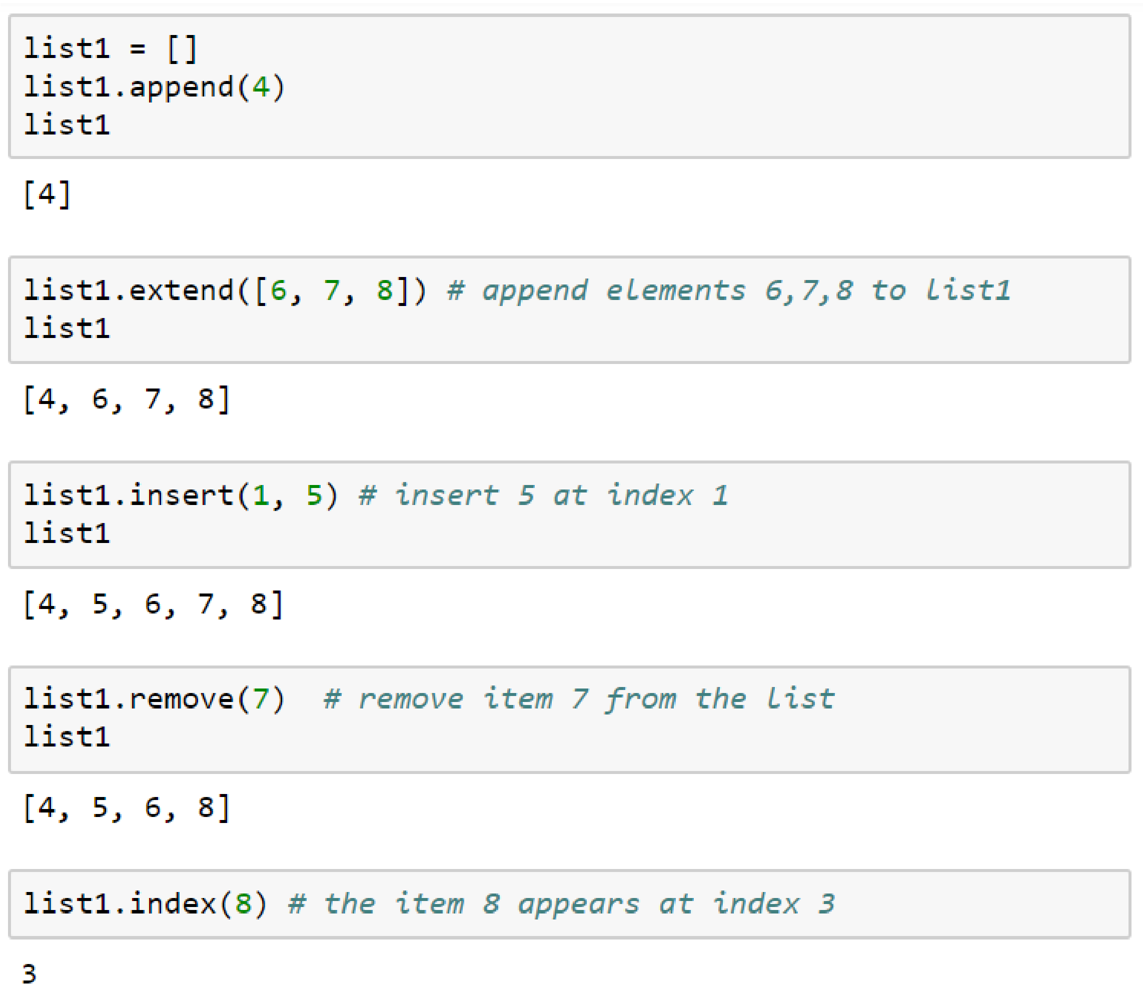
```
1
```

List Methods

- Just as for strings, Python lists come with a large number of useful methods:

Method	**Description**
append(*element*)	Appends *element* to the end of the list
extend(*sequence*)	Appends the elements of the *sequence* to the end of the list
index(*element*)	Returns the index of the first occurrence of *element* in the list. If *element* doesn't exist in the list, an exception is raised.
insert(*index*, *element*)	Inserts *element* at the specified *index*
pop([*index*])	Removes the element at the specified *index* and returns the element. If *index* is not specified, it removes and returns the last element from the list.
reverse()	Reverses the list in place
remove(*element*)	Removes the first occurrence of *element* from the list
sort()	Sorts the list in place (in ascending order)
copy()	Returns a copy of the list
count(*element*)	Returns the number of elements equal to the *element* in the list
clear()	Removes all elements from the list

List Methods - Examples



```
list1 = []
list1.append(4)
list1
```

```
[4]
```

```
list1.extend([6, 7, 8]) # append elements 6,7,8 to list1
list1
```

```
[4, 6, 7, 8]
```

```
list1.insert(1, 5) # insert 5 at index 1
list1
```

```
[4, 5, 6, 7, 8]
```

```
list1.remove(7)  # remove item 7 from the list
list1
```

```
[4, 5, 6, 8]
```

```
list1.index(8) # the item 8 appears at index 3
```

```
3
```

Sorting Lists

- The **sort()** method sorts the list *in place*, i.e., it changes the list object but doesn't return a value

```
list1 = [2, 0, 4, 3, 1]
list1.sort()
list1
```

```
[0, 1, 2, 3, 4]
```

- To get a sorted *copy of the list*, leaving it unchanged, use the **sorted()** built-in function:

```
list1 = ['a', 'e', 'A', 'c', 'b']
list2 = sorted(list1)  # returns a new list
list2
```

```
['A', 'a', 'b', 'c', 'e']
```

```
list1   # the old list is unchanged
```

```
['a', 'e', 'A', 'c', 'b']
```

Sorting Lists

- By default, sort() and sorted() order the items in an array in *ascending order*
- Set the optional argument **reverse=True** to sort the items in descending order:

```
a = [10, 5, 5, 2, 6, 1, 67]
sorted(a, reverse=True)
```

```
[67, 10, 6, 5, 5, 2, 1]
```

- Python 3, unlike Python 2, doesn't allow direct comparisons between different types, so it is an error to attempt to sort a list containing a mixture of types:

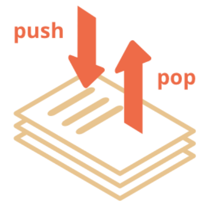
```
b = [5, '4', 2, 8]
b.sort()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-82-d6e2bccdd6f3> in <module>()
      1 b = [5, '4', 2, 8]
----> 2 b.sort()

TypeError: '<' not supported between instances of 'str' and 'int'
```

List as a Stack

- A **stack** is an ordered collection of elements which supports two operations:
  - push adds an element to the end
  - pop takes an element from the end
- The list methods **append()** and **pop()** make it very easy to use a list as a stack:
  - The end of the list is the top of the stack from which items may be added or removed



```
stack = []
stack.append(1)
stack.append(2)
stack.append(3)
print(stack)
```

```
[1, 2, 3]
```

```
stack.pop()
```

```
3
```

```
print(stack)
```

```
[1, 2]
```

Split and join

- The string method **split()** generates a list of substrings from a given string, split on a specified separator:

```
months = "Jan Feb Mar Apr May Jun"
months.split()     # By default, splits on whitespace
```

```
['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
```

```
fruits = "Apple,Banana,Melon,Orange"
fruits.split(",")
```

```
['Apple', 'Banana', 'Melon', 'Orange']
```

- The string method **join(sequence)** uses the string as a separator in joining a sequence of strings:

```
date = ["24", "6", "2018"]
"/".join(date)
```

```
'24/6/2018'
```

Exercise (20)

- Read a sentence from the user and print it in reverse (i.e., reverse the words in the sentence)
- For example:

```
Please enter a sentence:
An apple a day keeps the doctor away
The sentence in reverse:
away doctor the keeps day a apple An
```

Tuples

- A tuple is an *immutable* sequence of objects
  - i.e., you can't add, remove or modify its elements once it is created
  - As a result tuples are slightly faster for many uses than lists
- Tuples are constructed by placing the items inside parentheses:

```
t = (1, 2, 3, 4, 5)
t
```

```
(1, 2, 3, 4, 5)
```

```
t = ("alpha", "beta", "gamma")
t
```

```
('alpha', 'beta', 'gamma')
```

```
t0 = ()    # an empty tuple
t0
```

```
()
```

```
t1 = ("one", )   # a singleton tuple
t1
```

```
('one',)
```

To create a tuple with just one item (a singleton), you must type a trailing comma after the item

Operations on Tuples

- A tuple is essentially an immutable list
- As a result, most of the operations that can be performed on the list are also valid for tuples, such as:
  - Indexing or slicing using the [] operator
  - Built-in functions like len(), max(), min(), sum()
  - Membership operator in and not in
  - + and * operators
  - Comparison operators
- However, tuples don't support operations that modify the tuple itself, such as:
  - Item assignment
  - Methods such as append(), insert(), remove(), reverse(), and sort()

Operations on Tuples – Examples

```
t1 = ("alpha", "beta", "gamma")
len(t1)  # length of tuple
```

```
3
```

```
t1[1]  # indexing
```

```
'beta'
```

```
t1[1:]  # slicing
```

```
('beta', 'gamma')
```

```
"kappa" in t1  # membership
```

```
False
```

```
t1 * 2  # multiplication
```

```
('alpha', 'beta', 'gamma', 'alpha', 'beta', 'gamma')
```

```
t1 + ("delta",)   # addition
```

```
('alpha', 'beta', 'gamma', 'delta')
```

Packing and Unpacking

- Packing is a simple syntax which lets you create tuples "on the fly" without using parenthesis around the tuple's items:

```
t = 1, 2, 3
t
```

```
(1, 2, 3)
```

- You can also go the other way, and unpack the tuple into separate variables:

```
a, b, c = t
print(a, b, c)
```

```
1 2 3
```

- Tuple unpacking is a common way of assigning multiple variables in one line:

```
a, b, c = 10, 20, 30
b
```

```
20
```

  - The values 10, 20, 30 on the right-hand side are first packed into a tuple, which is then unpacked into the variables assigned on the left-hand side

Swapping Variables

- Swapping values between two variables is a common programming operation
- In languages like C, to perform swapping you have to create an additional variable to store the data temporarily
- In Python, we can use tuples to swap the values of two variables:

```
# Swapping values C-style
x, y = 10, 20
print(x, y)

tmp = x  # now tmp contains 10
x = y  # now x contains 20
y = tmp  # now y contains 10
print(x, y)
```

```
10 20
20 10
```

```
# Swapping values Python-style
x, y = 10, 20
print(x, y)

y, x = x, y
print(x, y)
```

```
10 20
20 10
```

Iterable Objects

- Strings, lists and tuples are all examples for *iterable objects*
  - i.e., collections of objects which can be taken one at a time
- One way of seeing this is to use the alternative method of initializing a list (or tuple) using the built-in constructor methods **list()** and **tuple()**
  - These take any iterable object and generate a list or a tuple from its sequence of items
  - The data elements are copied in the construction of the new object

```
list("hello")
```

```
['h', 'e', 'l', 'l', 'o']
```

```
tuple([1, 2, 3, 4])
```

```
(1, 2, 3, 4)
```

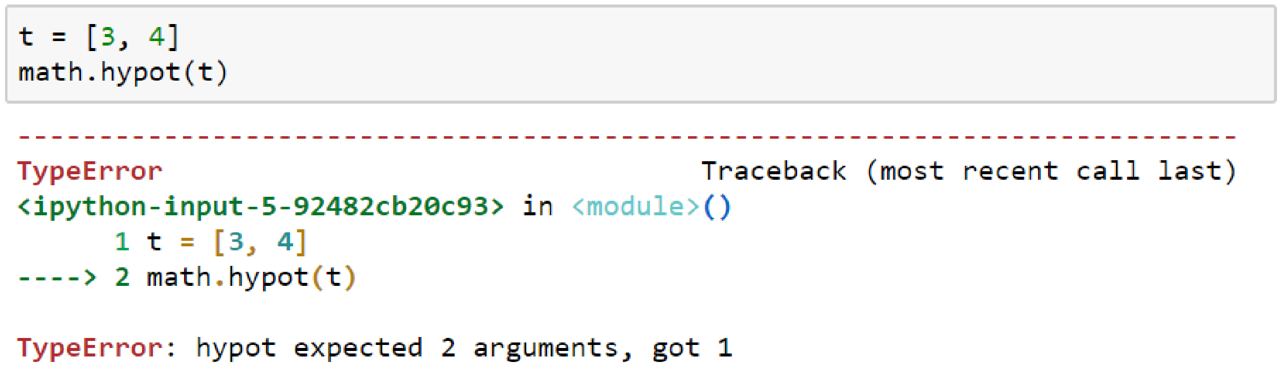
  - Because slices also return a copy of the elements in the sequence, the idiom b = a[:] is often used in preference to b = list(a)

Iterable Unpacking Operator *

- It is sometimes necessary to call a function with arguments taken from a list or other sequence
- The * syntax, used in a function call unpacks such a sequence into positional arguments to the function
- For example, the math.hypot function takes two arguments, a and b, and returns the quantity √*a2 + b2. If the arguments yo u*
- wish to use are in a list or tuple, the following will fail

Iterable Unpacking Operator *

- It is sometimes necessary to call a function with arguments taken from a list or other sequence
- The * syntax, used in a function call, unpacks such a sequence into positional arguments to the function
- For example, the math.hypot function takes two arguments, a and b
- If the arguments you wish to use are in a list or tuple, the following will fail:



```
t = [3, 4]
math.hypot(t)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-92482cb20c93> in <module>()
      1 t = [3, 4]
----> 2 math.hypot(t)

TypeError: hypot expected 2 arguments, got 1
```

Iterable Unpacking Operator *

- We could index the list explicitly to retrieve the two values we need:

```
t = [3, 4]
math.hypot(t[0], t[1])
```

```
5.0
```

- but a more elegant method is to unpack the list into arguments to the function:

```
math.hypot(*t)
```

```
5.0
```

- Python 3.5 extends the allowed uses of the * operator in more cases, e.g., in expressions involving iterators/lists/tuples:

```
list1 = [1, 2, 3, *[4, 5, 6]]
list1
```

```
[1, 2, 3, 4, 5, 6]
```

Exercise (21)

- Predict and explain the outcome of the following statements:

```
s = "hello"
a = [4, 10, 2]

print(s, sep="-")
print(*s, sep="-")
print(a)
print(*a, sep="\t")
list((*a,))
```

Control Statements

- Control statements allow us to execute a set of statements only when certain conditions are met
- Python has the following control statements:
  - if statement
  - if-else statement
  - if-elif-else statement
  - while loop
  - for loop
  - break statement
  - continue statement

if Statement

- The syntax of if statement is as follows:

```
if <boolean expression>:
    <indented statement 1>
    <indented statement 2>
    …
<non-indented statements>
```

- If the boolean expression evaluates to true, then all the statements inside the if block are executed
  - Each statement in the if block must be indented by the same number of spaces
  - It is strongly recommended to use **four spaces** to indent code
- If the expression evaluates to false, then all the statements in the if block are skipped
  - And execution continues with the statements following the if block (which are not indented)

if Statement

- The following program checks if the input number is greater than 10:

```python
num = int(input("Enter a number: "))
if num > 10:
    print("The number is greater than 10")
```

```
Enter a number: 100
The number is greater than 10
```

- The **if** block can have any number of statements:

```python
num = int(input("Enter a number: "))
if num > 10:
    print("Statement 1")
    print("Statement 2")
    print("Statement 3")
print("Executes every time you run the program")
print("Program ends here")
```

```
Enter a number: 45
Statement 1
Statement 2
Statement 3
Executes every time you run the program
Program ends here
```

if-else Statement

- An if-else statement executes one set of statements when the condition is true and a different set of statements when the condition is false
- In this way, an if-else statement allows us to follow two courses of action
- The syntax of if-else statement is as follows:

```
if <boolean expression>:
    <statements 1>
else:
    <statements 2>
```

- When if-else statement executes, the boolean expression is tested:
  - if it evaluates to True then statements inside the if block are executed
  - if it evaluates to False then the statements in the else block are executed

if-else Statement Examples

- A program to calculate the area and circumference of the circle:

```python
radius = int(input("Enter radius: "))
if radius > 0:
    print("Circumference = ", 2 * 3.14 * radius)
    print("Area = ", 3.14 * radius ** 2)
else:
    print("Please enter a positive number")
```

```
Enter radius: 4
Circumference =  25.12
Area =  50.24
```

- A program to check the password entered by the user:

```python
password = input("Enter a password: ")
if password == "secret":
    print("Welcome to the secret world")
else:
    print("Go home")
```

```
Enter a password: secret
Welcome to the secret world
```

if-else Statement

- The test expressions doesn’t have to evaluate explicitly to the boolean values True and False
- It is enough if they evaluate to a truthy or a falsy value
- For example:

```
num = int(input("Enter a number: "))
if num % 2:
    print(num, "is odd!")
else:
    print(num, "is even!")
```

```
Enter a number: 5
5 is odd!
```

  - This works because num % 2 = 1 for odd integers, which is equivalent to True and num % 2 = 0 for even integers, which is equivalent to False

Nested if Statements

- We can also write if or if-else statement inside another if or if-else statement
- For example, a program to find the largest of two numbers:

```python
num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))

if num1 > num2:
    print("The first number is greater")
else:
    if num1 < num2:
        print("The second number is greater")
    else:
        print("The numbers are equal")
```

```
Enter first number: 5
Enter second number: 7
The second number is greater
```

if-elif-else Statement

- The if-elif-else statement is another variation of the if-else statement, which allows us to test multiple conditions easily instead of writing nested if-else statements
- The syntax of the if-elif-else statement is:

```
if <boolean expression 1>:
    <statements 1>
elif <boolean expression 2>:
    <statements 2>
elif <boolean expression 3>:
    <statements 3>
…
else:
    <statements>
```

  - if <boolean expression 1> evaluates to True, <statements 1> are executed
  - otherwise, if <boolean expression 2> evaluates to True, <statements 2> are executed, and so on
  - if none of the preceding conditions evaluate to True, the statements in the block of code following else: are executed

if-elif-else Example

- A program to determine a student grade based upon its score in the test:

```
score = int(input("Enter your test score: "))

if score >= 90:
    print("Excellent! Your grade is A")
elif score >= 80:
    print("Great! Your grade is B")
elif score >= 70:
    print("Good. Your grade is C")
elif score >= 60:
    print("Your grade is D. You should work harder.")
else:
    print("You failed in the exam")
```

```
Enter your test score: 87
Great! Your grade is B
```

Conditional Expressions

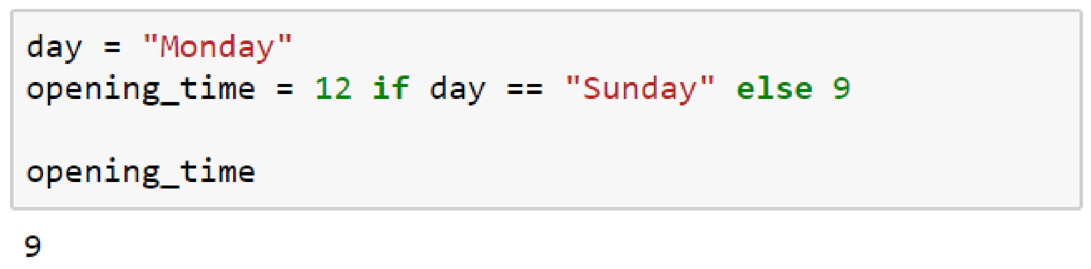
- A conditional expression lets you write a single assignment statement that assigns a value to a variable which depends on the truthness of some condition

```
x = a if condition else b
```

  - a is assigned to x if condition evaluates to true, and b is assigned to x otherwise
- This is equivalent to writing:

```
if condition:
    x = a
else:
    x = b
```

- Example:



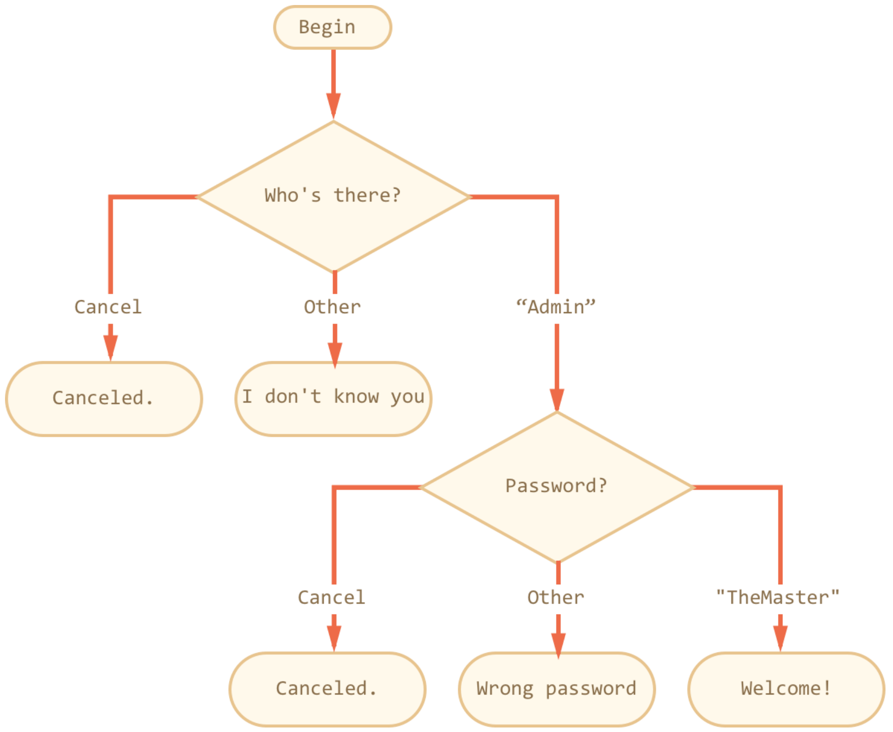
```
day = "Monday"
opening_time = 12 if day == "Sunday" else 9

opening_time
```

```
9
```

Exercise (22)

- Write the code which asks for a login name
  - If the visitor enters “Admin“, then ask for a password
  - If the input is an empty line – print “Canceled”
  - If it’s another string – then print “I don’t know you”
- The password is checked as follows:
  - If it equals “TheMaster”, then show “Welcome!”
  - Another string – show “Wrong password”
  - For an empty string, show “Canceled”



Exercise (23)

- In the Gregorian calendar a year is a *leap year* if it is divisible by 4, with the exception that years divisible by 100 are *not* leap years, unless they are also divisible by 400
  - For example, the years 1700 and 1800 were not leap years, but the years 1804 and 2000 were
- Write a program that lets the user check if a given year is a leap year or not

Loops in Python

- A loop allows us to execute some set of statements multiple times
- Python provides two types loops:
  - while loop
  - for loop

The while loop

- A while loop is a conditionally controlled loop, which executes a block of statements as long as the condition is true. Its syntax is:

```
while <boolean expression>:
    <indented statement 1>
    <indented statement 2>
    …
<non-indented statements>
```

- The indented group of statements is known as the **while block** or **loop body**
- The statements in the while block will keep executing as long as the condition is true
  - Each execution of the loop body is called **iteration**
- When the condition becomes false, the loop terminates and program control continues with the execution of the statement following the while block

The while loop

- The following while loop calculates the sum of numbers between 1 and 10:

```
sum = 0

i = 1
while i < 11:
    sum += i
    i += 1

print("sum is", sum)
```

```
sum is 55
```

  - This while loop executes until i < 11
  - The variable sum is used to accumulate the sum of numbers from 1 to 10
  - In each iteration, the value of i is added to the variable sum and i is incremented by 1
  - When i becomes 11, the loop terminates and the program control comes out of the while loop to execute the print() function at the last line

The while loop

- The following program converts temperatures from Fahrenheit to Celsius, until the use decides to exit by answering ‘n’:

```python
# A program that converts tempratures from Fahrenheit to Celsius
keep_calculating = True

while keep_calculating:
    fah = int(input("Enter temprature in Fahrenheit: "))
    cel = (fah - 32) * 5/9
    print(format(fah, "0.2f") + "°F is same as", format(cel, "0.2f") + "°C\n")

    keep_calculating = input("Want to calculate more? Press y for Yes, n for No: ") == 'y'
```

```
Enter temprature in Fahrenheit: 60
60.00°F is same as 15.56°C

Want to calculate more? Press y for Yes, n for No: y
Enter temprature in Fahrenheit: 75
75.00°F is same as 23.89°C

Want to calculate more? Press y for Yes, n for No: n
```

Exercise (24)

- Write a program that asks the user to enter a number and prints the sum of its digits
- For example, if the input is the number 8402 your program should print 14

Exercise (25)

- The *Fibonacci sequence* is the sequence of numbers generated by applying the rules:

  $a_1 = a_2 = 1, a_i = a_{i-1} + a_{i-2}$
- That is, the ith Fibonacci number is the sum of the previous two: 1, 1, 2, 3, 5, 8, 13, …
- Write a program that gets a number *n* from the user, and prints all the Fibonacci numbers which are less than or equal to *n*
- For example:

```
Enter the maximum for Fibonacci numbers: 500
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377
```

The for loop

- It is often necessary to take the items in an iterable object one by one and do something with each in turn
- Other languages, such as C, use for loops to refer to each item in turn by its integer index
- In Python the more natural and convenient way is with the idiom:

```
for item in iterable object:
    # loop body
    <indented statements>
```

- The loop executes a block of statements (the loop body) for each item of the iterable object

The for loop – Examples

```python
# iterating over a list
fruit_list = ['apple', 'melon', 'banana', 'orange']
for fruit in fruit_list:
    print(fruit, end=" ")
```

```
apple melon banana orange
```

```python
# iterating over a tuple
for i in (35, 22, 78, 5, 123):
    print(i, end=" ")
```

```
35 22 78 5 123
```

```python
# iterating over a string
for c in "hello":
    print(c, end=",")
```

```
h,e,l,l,o,
```

Nested Loops

- Loops can be nested – the inner loop block needs to be indented by the same amount of whitespace as the outer loop (i.e. eight spaces):

```
fruit_list = ['apple', 'melon', 'banana', 'orange']
for fruit in fruit_list:
    for letter in fruit:
        print(letter, end=".")
    print()
```

```
a.p.p.l.e.
m.e.l.o.n.
b.a.n.a.n.a.
o.r.a.n.g.e.
```

  - In this example, we iterate over the string items in fruit_list one by one, and for each string (fruit name), iterate over its letters

The range Type

- The **range** type represents a sequence of numbers, which is generally used to iterate over with for loops
  - It would have been possible to create a list to hold the numbers, but this is memory inefficient
- A range object can be constructed with up to three arguments defining the first integer, the integer to stop at and the stride (the space between values)

```
range([a0=0], m, [stride=1]
```

  - If the initial value a0 is not given it is taken to be 0
  - stride is also optional and if it is not given it is taken to be 1
  - stride can be negative
- The range represents the arithmetic progression $a_n = a_0 + nd$ for n = 0, 1, 2,...
  - *d* is the stride

The range Type

- Examples:

```
a = range(5)         # 0,1,2,3,4
b = range(1, 6)      # 1,2,3,4,5
c = range(0, 6, 2)   # 0,2,4
d = range(10, 0, -2) # 10,8,6,4,2
```

- In Python 3, the object created by range is not a list, but rather it is an iterable object that can produce integers on demand
- range objects can be indexed, cast into lists and tuples, and iterated over:

```
d[0]
```

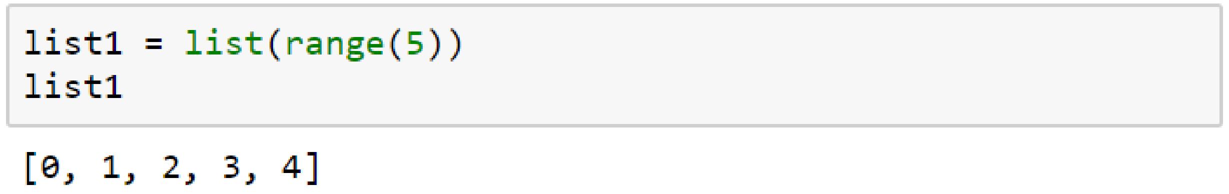
```
10
```

```
d[1]
```

```
8
```

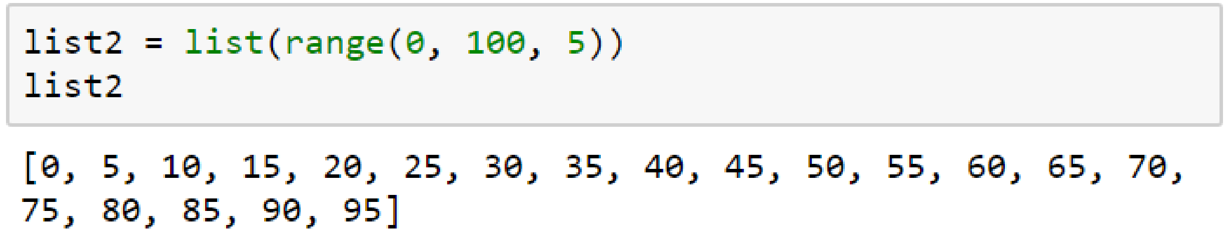
Creating Lists using range() Function

- The range() function can be used to create long lists, by passing the range object to the list() constructor function
  - The list() function uses the numbers from the range sequence to create a list



```
list1 = list(range(5))
list1
```

```
[0, 1, 2, 3, 4]
```



```
list2 = list(range(0, 100, 5))
list2
```

```
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70,
75, 80, 85, 90, 95]
```

For loop with range()

- Examples:

```
for i in range(5):
    print(i, end=" ")
```

```
0 1 2 3 4
```

```
for i in range(90, 99):
    print(i, end=" ")
```

```
90 91 92 93 94 95 96 97 98
```

```
for i in range(10, 20, 3):
    print(i, end=" ")
```

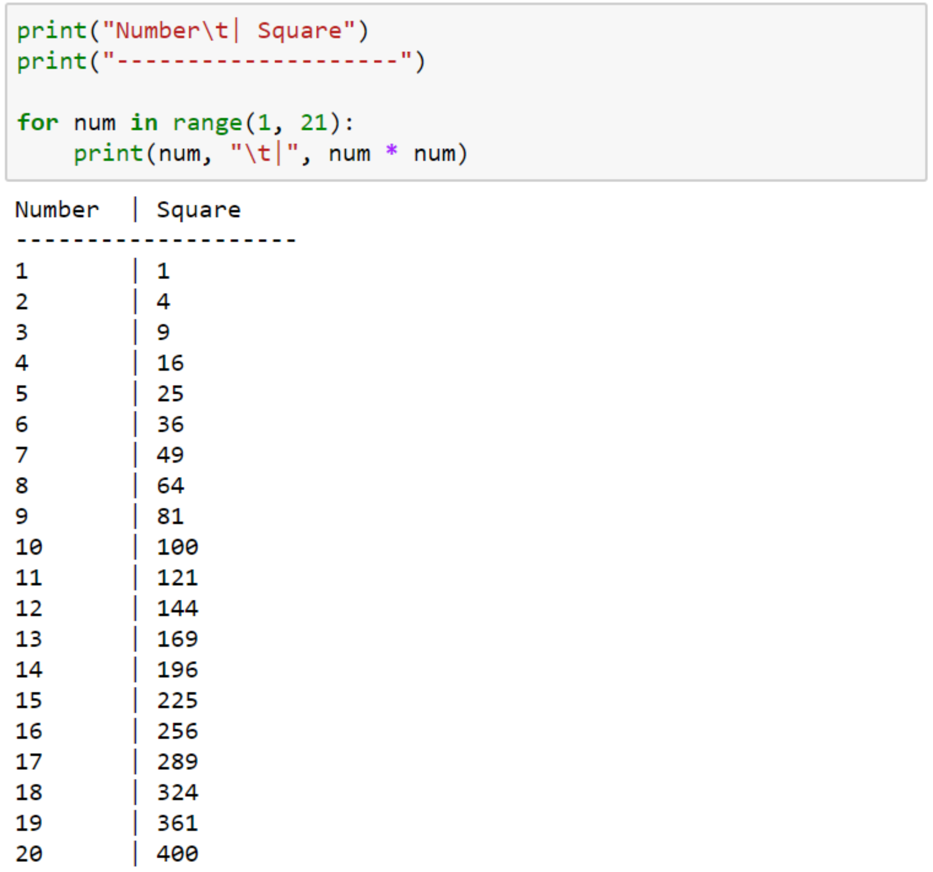
```
10 13 16 19
```

```
for i in range (20, 10, -2):
    print(i, end=" ")
```

```
20 18 16 14 12
```

For loop with range()

- The following program uses a for loop to generate the squares of numbers from 1 to 20:



```python
print("Number\t| Square")
print("--------------------")

for num in range(1, 21):
    print(num, "\t|", num * num)
```

```
Number  | Square
--------------------
1       | 1
2       | 4
3       | 9
4       | 16
5       | 25
6       | 36
7       | 49
8       | 64
9       | 81
10      | 100
11      | 121
12      | 144
13      | 169
14      | 196
15      | 225
16      | 256
17      | 289
18      | 324
19      | 361
20      | 400
```

Break Statement

- The **break** statement is used to terminate the loop prematurely when a certain condition is met
- When a break statement is encountered inside the body of the loop, the current iteration stops and program control jumps to the statements following the loop:

```
for i in range(1, 10):
    if i == 5: # when i 5, exit the loop
        break
    print("i =", i)
print("End of program")
```

```
i = 1
i = 2
i = 3
i = 4
End of program
```

Break Statement

- Similarly, to find the index of the first occurrence of a negative number in a list:

```
list1 = [0, 4, 5, -2, 5, 10]
for idx, num in enumerate(list1):
    if num < 0:
        break
print(num, "occurs at index", idx)
```

```
-2 occurs at index 3
```

  - Note that after escaping from the loop, the variables i and a have the values that they had within the loop at the break statement

Break Statement

- In a nested loop the break statement only terminates the loop in which it appears

```python
for i in range(1, 5):
    print("Outer loop: i =", i)
    for j in range (10, 15):
        print("\tInner loop: j =", j)
        if j == 12:
            print("\tBreaking out of inner loop")
            break
```

```
Outer loop: i = 1
        Inner loop: j = 10
        Inner loop: j = 11
        Inner loop: j = 12
        Breaking out of inner loop
Outer loop: i = 2
        Inner loop: j = 10
        Inner loop: j = 11
        Inner loop: j = 12
        Breaking out of inner loop
Outer loop: i = 3
        Inner loop: j = 10
        Inner loop: j = 11
        Inner loop: j = 12
        Breaking out of inner loop
Outer loop: i = 4
        Inner loop: j = 10
        Inner loop: j = 11
        Inner loop: j = 12
        Breaking out of inner loop
```

Continue Statement

- The **continue** statement is used to move ahead to the next iteration without executing the remaining statement in the body of the loop

```
for i in range(1, 10):
    if i % 2 == 1:
        continue
    print("i =", i)
```

```
i = 2
i = 4
i = 6
i = 8
```

  - if i is not divisible by 2 (and hence i % 2 == 1), that loop iteration is canceled and the loop resumed with the next value of i (the print statement is skipped)

Continue Statement

- We can also use break and continue statement together in the same loop:

```
while True:
    value = input("Enter a number: ")
    if value == 'q': # if input is 'q' exit from the loop
        break
    if not value.isdigit(): # if input is not a digit move to the next iteration
        print("Enter digits only\n")
        continue

    value = int(value)
    print("Cube of", value, "is", value**3, "\n")
```

```
Enter a number: 5
Cube of 5 is 125

Enter a number: @
Enter digits only

Enter a number: 11
Cube of 11 is 1331

Enter a number: q
```

Pass Statement

- The pass command does nothing (a “null” statement)
- It is useful as a “stub” for code that has not yet been written but where a statement is syntactically required by Python’s whitespace convention

```python
for i in range(1, 11):
    if i == 6:
        pass        # do something special if i is 6
    if i % 3 == 0:
        print(i, "is divisible by 3")
```

```
3 is divisible by 3
6 is divisible by 3
9 is divisible by 3
```

else block

- A for or while loop may be followed by an **else** block of statements, which will be executed only if the loop finished "normally" (without the intervention of a break)
  - For for loops, this means these statements will be executed after the loop has reached the end of the sequence it is iterating over
  - For while loops, they are executed when the while condition becomes False
- For example, consider again our program to find the first occurrence of a negative number in a list. It behaves oddly if there aren't any negative numbers in the list:

```python
list1 = [0, 4, 5, 2, 5, 10]
for idx, num in enumerate(list1):
    if num < 0:
        break
print(num, "occurs at index", idx)
```

```
10 occurs at index 5
```

  - It outputs the index and number of the last item in the list (whether it is negative or not)

else block

- A way to improve this is to notice when the for loop runs through every item without encountering a negative number (and hence the break) and output a message:

```python
list1 = [0, 4, 5, 2, 5, 10]
for idx, num in enumerate(list1):
    if num < 0:
        print(num, "occurs at index", idx)
        break
else:
    print("no negative numbers in the list")
```

```
no negative numbers in the list
```

- As another example, the following routine checks if a given number is prime or not:

```python
import math
num = int(input("Enter a number: "))

for i in range(2, int(math.sqrt(num)) + 1):
    if num % i == 0:
        print(num, "is not prime")
        break
else:
    print(num, "is prime!")
```

```
Enter a number: 97
97 is prime!
```

Exercise (26)

- Get from the user two numbers: low and high
- Output all the even numbers between low and high (note that low and high themselves might be odd numbers)
- For example, if the user enters low = 5 and high = 14, you should print the numbers 6,8,10,12,14

Exercise (27)

- Use a for loop to calculate π from the first 20 terms of the *Madhava series:*

$$\pi = \sqrt{12}\left(1 - \frac{1}{3\cdot 3} + \frac{1}{5\cdot 3^2} - \frac{1}{7\cdot 3^3} + \ldots\right)$$

Exercise (28)

- Get from the user a number
- Print to the console a square of stars whose length is the number specified by the user
- For example, if the user entered the number 8, your should print:

```
********
********
********
********
********
********
********
********
```

Exercise (29)

- The *Luhn algorithm* is a simple checksum formula used to validate credit card and bank account numbers
- The algorithm may be written as the following steps:
  - Reverse the number
  - Treating the number as an array of digits, take the even-indexed digits (where the indexes *start at 1*) and double their values
  - If a doubled digit results in a number greater than 10, add the two digits (e.g., the digit 6 becomes 12 and hence 1 + 2 = 3)
  - Sum this modified array
  - If the sum of the array modulo 10 is 0 the credit card number is valid
- Write a Python program to take a credit card number as a string of digits (possibly in groups, separated by spaces) and establish if it is valid or not
- For example, the string ’4799 2739 8713 6272’ is a valid credit card number, but any number with a single digit in this string changed is not

enumerate()

- We cannot modify items in the list while iterating over it like this:

```
grades = [75, 83, 92, 86, 97]

for g in grades:
    g *= 1.05   # g holds copies of the values in the list
grades
```

```
[75, 83, 92, 86, 97]
```

  - Changing the value of g in the loop body doesn't affect the elements in the list
- It is tempting to use the range() function to provide the indexes of the list like this:

```
for i in range(len(grades)):
    grades[i] *= 1.05
print(grades)
```

```
[78.75, 87.15, 96.60000000000001, 90.3, 99.75]
```

enumerate()

- This works, of course, but it is more natural to avoid the explicit construction of a range object (and the call to the len built-in) by using **enumerate()**
- This method takes an iterable object and produces, for each item in turn, a tuple (index, item), consisting of a counting index and the item itself:

```python
grades = [75, 83, 92, 86, 95]
for i, grade in enumerate(grades):
    grades[i] = grade * 1.05
print(grades)
```

```
[78.75, 87.15, 96.60000000000001, 90.3, 99.75]
```

  - Each (index, item) tuple is unpacked in the for loop into the variables i and grade
- It is also possible to set the starting value of index to something other than 0:

```python
mammals = ['kangaroo', 'wombat', 'platypus']
list(enumerate(mammals, 4))
```

```
[(4, 'kangaroo'), (5, 'wombat'), (6, 'platypus')]
```

zip()

- What if you want to iterate over two (or more) sequences at the same time?
- The **zip()** built-in function creates an iterator object, in which each item is a tuple of items taken in turn from the sequences passed to it:

```
a = [1, 2, 3, 4]
b = ['a', 'b', 'c', 'd']
zip(a, b)
```

```
<zip at 0x23bc19420c8>
```

```
for pair in zip(a, b):
    print(pair)
```

```
(1, 'a')
(2, 'b')
(3, 'c')
(4, 'd')
```

```
list(zip(a, b))   # convert to list
```

```
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
```

zip()

- A nice feature of zip is that it can be used to *unzip* sequences of tuples as well:

```
z = zip(a, b)      # z generates (1,'a'),(2,'b'),(3,'c'),(4,'d')
A, B = zip(*z)     # zip((1,'a'),(2,'b'),(3,'c'),(4,'d'))
print(A, B)
```

```
(1, 2, 3, 4) ('a', 'b', 'c', 'd')
```

```
list(A) == a, list(B) == b
```

```
(True, True)
```

- zip does not copy the items into a new object, so it is memory-efficient and fast
  - but this means that you only get to iterate over the zipped items once and you can't index it

```
z = zip(a, b)
print(list(z))

for pair in z:
    print(pair) # nothing: we've already exhausted the iterator z
```

```
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
```

Exercise (30)

- A list could be used as a simple representation of a polynomial, *P(x), with* the items as the coefficients of the successive powers of *x,* and their indexes as the powers themselves
- Thus, the polynomial $P(x) = 4+5x+2x^3$ would be represented by the list [4, 5, 0, 2]
- Why does the following attempt to differentiate a polynomial fail to produce the correct answer?

```python
P = [4, 5, 0, 2]
dPdx = []
for i, c in enumerate(P[1:]):
    dPdx.append(i * c)
dPdx
```

```
[0, 0, 4]
```

- How can this code be fixed?

Exercise (31)

- Sorting a list of tuples arranges them in order of the first element in each tuple first
- If two or more tuples have the same first element, they are ordered by the second element, and so on:

```
sorted([(3, 1), (1, 4), (3, 0), (2, 2), (1, -1)])
```

```
[(1, -1), (1, 4), (2, 2), (3, 0), (3, 1)]
```

- This suggests a way of using zip to sort one list using the elements of another
- Implement this method on the data below to produce an ordered list of the average amount of sunshine in hours in London by month
- Output the sunniest month first

Jan	Feb	Mar	Apr	May	Jun
44.7	65.4	101.7	148.3	170.9	171.4
Jul	Aug	Sep	Oct	Nov	Dec
176.7	186.1	133.9	105.4	59.6	45.8

List Comprehension

- A list comprehension is a construct for creating a list based on another iterable object in a single line of code
- The syntax of list comprehension is:

```
[ expression for item in iterable ]
```

  - In each iteration of the for loop an item is assigned a value from the iterable object, and the result of the expression is then used to produce values for the list
- For example, given a list of numbers, list1, a list of the squares of those numbers may be generated as follows:

```
list1 = [1, 2, 3, 4, 5, 6]
```

```
list2 = [x**2 for x in list1]
list2
```

```
[1, 4, 9, 16, 25, 36]
```

```
# the same as:
list2 = []
for x in list1:
    list2.append(x**2)
list2
```

```
[1, 4, 9, 16, 25, 36]
```

  - This is a faster and syntactically nicer way of creating the same list with the for loop on the right

List Comprehension

- List comprehensions can also contain conditional statements, as follows:

```
[ expression for item in iterable if condition ]
```

  - The only difference is that the expression before the for keyword is evaluated only when the the condition is True

- For example, the following list comprehension generates a list of the squares of the even numbers in list1:

```
list2 = [x**2 for x in list1 if x % 2 == 0]
list2
```

```
[4, 16, 36]
```

- You can also use an if .. else expression, which must appear before the for loop:

```
[x**2 if x % 2 == 0 else x**3 for x in list1]
```

```
[1, 4, 27, 16, 125, 36]
```

  - This comprehension squares the even integers and cubes the odd integers in list1

List Comprehension

- The sequence used to construct the list can be any iterable object:

```
[x**3 for x in range(1, 10)]
```

```
[1, 8, 27, 64, 125, 216, 343, 512, 729]
```

```
[w.upper() for w in "hello"]
```

```
['H', 'E', 'L', 'L', 'O']
```

- Finally, list comprehensions can be nested
- For example, the following code flattens a list of lists:

```
vlist = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
[e for row in vlist for e in row]
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Exercise (32)

- Consider the lists:

```
a = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
b = [4, 2, 6, 1, 5, 0, 3]
```

- Predict and explain the output of the following statements:

```
[a[x] for x in b]
[a[x] for x in sorted(b)]
[a[b[x]] for x in b]
[x for (y,x) in sorted(zip(b,a))]
```

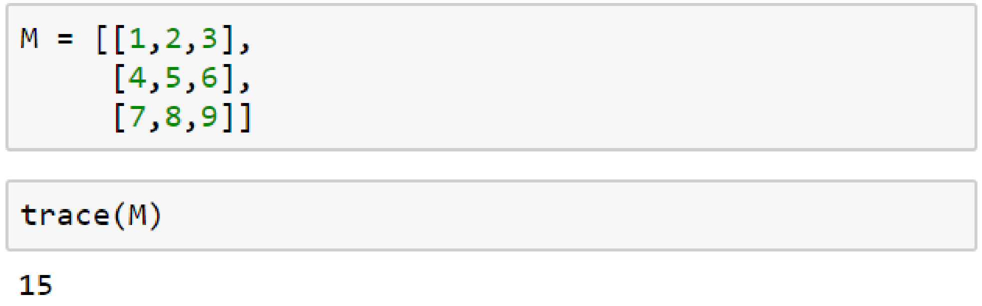
Exercise (33)

- What does the following code do and how does it work?

```
nmax = 5
x = [1]
for n in range(1, nmax + 2):
    print(x)
    x = [([0] + x)[i] + (x + [0])[i] for i in range(n + 1)]
```

Exercise (34)

- Write a function **trace(*M*)** that uses list comprehension to calculate the *trace* of the matrix *M* (that is, the sum of its diagonal elements)
- Hint: the **sum()** built-in function takes an iterable object and sums its values
- Example:



```
M = [[1,2,3],
     [4,5,6],
     [7,8,9]]
```

```
trace(M)
```

```
15
```

Exercise (35)

- The ROT13 substitution cipher encodes a string by replacing each letter with the letter 13 letters after it in the alphabet (cycling around if necessary)
  - For example, a→n and p→c
- Write a function **rot13_word(word)** that gets a word expressed as a string of lowercase characters, and uses list comprehension to construct the ROT13-encoded version of that string
  - Hint: use the Python functions ord() and chr()
- Write a function **rot13_sentence(sentence)** that encodes sentences of words (in lowercase) separated by spaces into a ROT13 sentence (in which the encoded words are also separated by spaces)
  - Use list comprehension and the previous function for this task

```
rot13_sentence("hello world")
```

```
'uryyb jbeyq'
```