Data Structure and Lab (CSE123)

Review - Python Programming

Muhammad Tariq Mahmood

tariq@koreatech.ac.kr
School of Computer Science and Engineering

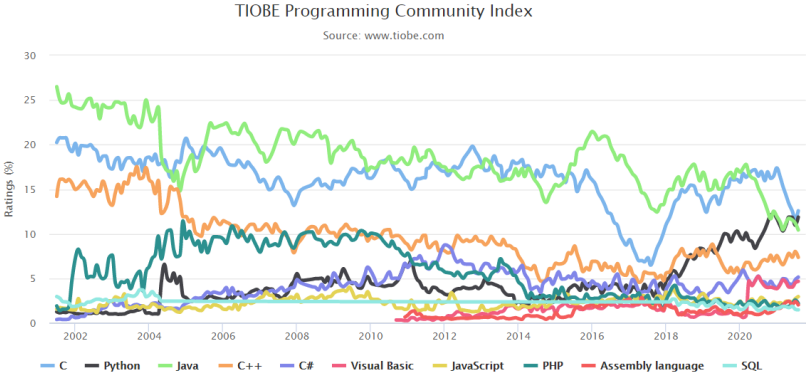
Note: These notes are prepared from the following resources.

- Data Structures and Algorithms Using Python by Rance D. Necaise
- Data Structures Using Python by Y.K. Choi and I.G. Chan (Korean)
- https://www.geeksforgeeks.org/data-structures/
- Starting Out with Python, Pearson by Tony Gaddis (2021)
- Introduction to Programming Using Python, Pearson by Y. Daniel Liang, .
- https://docs.oracle.com/javase/tutorial/ (tutorials, and references).
- https://www3.ntu.edu.sg/home/ehchua/programming/index.html#Java
- https://docs.python.org/3/tutorial/

Contents

Python Overview

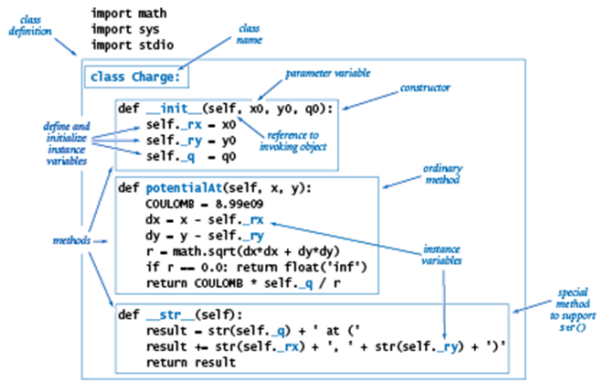
- The Python programming language was originally developed by Guido van Rossum in the early 1990s, and has since become a prominently used language in industry and education.



https://www.tiobe.com/tiobe-index/

Python Overview (cont...)

- Top Programming Languages 2021: IEEE Spectrum:
- Rankings are created by weighting and combining 11 metrics from eight sources: CareerBuilder, GitHub, Google, Hacker News, the IEEE, Reddit, Stack Overflow, and Twitter.



https://spectrum.ieee.org/top-programming-languages/

Main features of Python

Main **features** of Python are listed here:

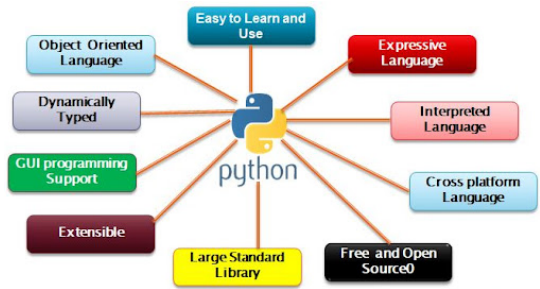
1. Easy to Learn and Use: Python is an easy and intuitive language. Python scripts are easy to read and understand.

2. Expressive Language: Python (like Perl) is expressive. A single line of Python code can do many lines of code in traditional general-purpose languages (such as C/C++/Java).

3. Free and Open Source: Python is free and open-source. It is cross-platform and runs on Windows, Linux/UNIX, and Mac OS X.

4. Extensible Python is easily extensible with C/C++/Java code, and easily embeddable in applications.

5. Dynamically Typed: Python is a dynamically typed language. Means the type for a value is decided at runtime, not in advance. It is not necessary to specify the type of data when declaring it.



6. cross-platform language: A Python program written on a Macintosh computer can run on a Linux system or on a Windows computer.

Main features of Python (cont...)

7. object-oriented: Python is an object-oriented language since its beginning. Object-oriented programming is a programming paradigm that provides a means of structuring programs so that properties(data members) and behaviors (methods) are bundled into individual objects.

8. Interpreted Language: Python is an interpreted language, which means the source code of a Python program is converted into bytecode that is then executed by the Python virtual machine.

9. Large Standard Library: Python has a large and extensive library and offers a wide set of modules and functions for rapid application development.

10. GUI Programming Support: We can develop GUI (Graphical user interfaces) applications using Python.

11. Python provides automatic memory management. You do not need to allocate and free memory in your programs. Python provides high-level data types. Python is well suited for rapid application development (RAD).

Python Installation

- The latest version of the language is freely available at *www.python.org*, along with **documentation** and **tutorials**.
- You could install either:
  - It is suggested, install "Anaconda distribution" of Python 3, which includes a Command Prompt, IDEs (Jupyter Notebook and **Spyder**, **VS code**), and bundled with commonly-used packages (such as NumPy, Matplotlib and Pandas that are used for data analytics).
  - Goto Anaconda mother site (@ https://www.anaconda.com/) Choose "Anaconda Distribution" Download Choose "Python 3.x" Follow the instructions to install.
  - Plain Python from Python Software Foundation @ https://www.python.org/download/, download the 32-bit or 64-bit MSI installer, and run the downloaded installer.

Python Basic Syntax's

Comments

- Comments are ignored by the Python Interpreter, but they are critical in providing explanation and documentation for others to read your program.
- Comments can be written in three ways - entirely on its own line, next to a statement of code, and as a multi-line comment block.

1. A Python comment begins with a hash sign and last till the end of the current line.

```
#defining the post code
postCode = 75000
```

2. You can also write a comment next to a code statement.

```
product = {
"productId": 0,         # product id, default:  0
"description": "",      # item description, default:  empty
"categoryId": 0,        # item category, default:  0
"price": 0.00           # price, default:  0.00
}
```

3. Python doesn't have explicit support for multi-line comments, however it can be done using """ text """

```
"""
If I really hate pressing 'enter' and
typing all those hash marks, I could
just do this instead
"""
product = {
"productId": 0,         # product id, default:  0
"description": "",      # item description, default:  empty
"categoryId": 0,        # item category, default:  0
"price": 0.00           # price, default:  0.00
}
```

Python Basic Syntax's (cont...)

Statements

- Each statement in Python has its own specific purpose and its own specific syntax (the rules that define its structure)
- A Python statement is delimited by a newline. A statement cannot cross line boundaries, except:

1. An expression in parentheses (), square bracket [], and curly braces {} can span multiple lines.
2. A backslash (\) at the end of the line denotes continuation to the next line.
3. Unlike C/C++/Java, you don't place a semicolon (;) at the end of a Python statement. But you can place multiple statements on a single line, separated by semicolon (;).

Statement	Role	Example
Assignment	Creating references	`a, b = 'good', 'bad'`
Calls and other expressions	Running functions	`log.write("spam, ham")`
print calls	Printing objects	`print('The Killer', joke)`
if/elif/else	Selecting actions	`if "python" in text:` `  print(text)`
for/else	Iteration	`for x in mylist:` `  print(x)`
while/else	General loops	`while X > Y:` `  print('hello')`
pass	Empty placeholder	`while True:` `  pass`
break	Loop exit	`while True:` `  if exittest(): break`
continue	Loop continue	`while True:` `  if skiptest(): continue`
def	Functions and methods	`def f(a, b, c=1, *d):` `  print(a+b+c+d[0])`
return	Functions results	`def f(a, b, c=1, *d):` `  return a+b+c+d[0]`
yield	Generator functions	`def gen(n):` `  for i in n: yield i*2`

Python Basic Syntax's (cont...)

Block and Indentation

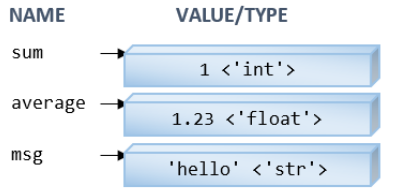
- A block is a group of statements executing as a unit.
- Unlike C/C++/Java, which use braces to group statements in a body block, Python uses indentation for body block.
- Indentation is syntactically significant in Python - the body block must be properly indented.
- You can place the body-block in the same line, separating the statement by semi-colon (;)
- Python does not specify how much indentation to use, but all statements of the SAME body block must start at the SAME distance from the right margin.
- It is recommended to use 4 spaces for each indentation level.

```
header_1:        # Headers are terminated by a colon
    statement_1_1 # Body blocks are indented
    statement_1_2
    ......
header_2:
    statement_2_1
    statement_2_2
    ......

# This is NOT recommended.
header_1: statement_1_1
header_2: statement_2_1; statement_2_2; ......
```

```
# if-else
x = 0
if x == 0:
    print('x is zero')
else:
    print('x is not zero')

#in the same line
sum = 0
number = 1
while number <= 100: sum += number; number += 1
```

Variables, Identifiers and Constants

- Like all programming languages, a variable is a named storage location. A variable has a name (or identifier) and holds a value.
- Python is dynamically typed. You do NOT need to declare a variable before using it. A variables is created via the initial assignment.
- Python associates types with the objects, not the variables, i.e., a variable can hold object of any types,



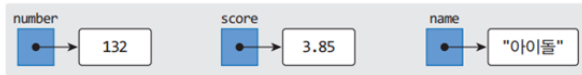
- **Examples**

```
sum = 1 # Create a variable called sum by assigning an integer into it
print(type(sum)) # <class 'int'>
average = 1.23 # Create a variable called average by assigning a floating-point number into it
print(type(average)) # <class 'float'>
average = 78    # Re-assign an integer value
print(type(average)) # <class 'int'>
msg = 'Hello'   # Create a variable msg by assigning a string into it
print(type(msg) ) #<class 'str'>
```

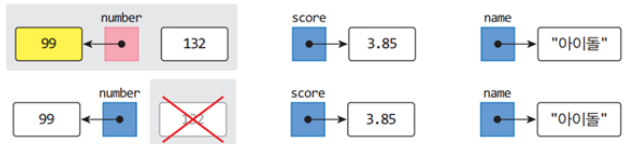
- Behavior of variables

Python Basic Syntax's (cont...)

```
number = 132
score = 3.85
name = "아이돌"
```

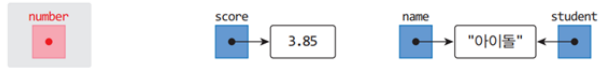



```
number = 99
```




```
student = name
```


```
number = None
```



- Python does not support constants, where its contents cannot be modified. (C supports constants via keyword const, Java via final.)

Python Basic Syntax's (cont...)

- Rules of Identifier (Names) An identifier starts with a letter (A-Z, a-z) or an underscore (_), followed by zero or more letters, underscores and digits (0-9). Python does not allow special characters such as $ and@.

1. Use long descriptive names:

```
# Not recommended
# The au variable is the number of active users
au = 105
# Recommended
total_active_users = 105
```

2. Use descriptive intention revealing names

```
# Not recommended
c = [UK, USA, UAE]
for x in c:
print(x)
# Recommended
cities = [UK, USA, UAE]
for city in cities:
print(city)
```

3. Avoid using ambiguous shorthand:

```
# Not recommended
fn = 'John'
Ln = Doe
cre_tmstp = 1621535852
# Recommended
first_name = JOhn
Las_name = Doe
creation_timestamp = 1621535852
```

4. Always use the same vocabulary:

```
# Not recommended
client_first_name = John
customer_last_name = Doe;
# Recommended
client_first_name = John
client_last_name = Doe
```

Python Basic Syntax's (cont...)

- Keywords: Python has a set of keywords that are reserved words that cannot be used as variable names, function names, or any other identifiers:

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

- True and False are truth values in Python. They are the results of comparison operations or logical (Boolean) operations in Python.
- None is a special constant in Python that represents the absence of a value or a null value. and, or, not are the logical operators in Python

Data Types

- Python has a large number of built-in data types, such as Numbers (Integer, Float, Boolean, Complex Number), String, List, Tuple, Set, Dictionary and File. More high-level data types, such as Decimal and Fraction, are supported by external modules.
- Python associates types with objects, instead of variables. That is, a variable does not have a fixed type and can be assigned an object of any type. A variable simply provides a reference to an object.
- You can use the built-in function type(varName) to check the type of a variable or literal.

Class	Description	Immutable?
bool	Boolean value	✓
int	integer (arbitrary magnitude)	✓
float	floating-point number	✓
list	mutable sequence of objects	
tuple	immutable sequence of objects	✓
str	character string	✓
set	unordered set of distinct objects	
frozenset	immutable form of set class	✓
dict	associative mapping (aka dictionary)	

- Pythons classes may also define one or more **methods** (also known as **member functions**), which are invoked on a specific instance of a class using the **dot (.)** operator.
- A class is **immutable** if each object of that class has a fixed value upon instantiation that cannot subsequently be changed.
- The bool Class: The bool class is used to manipulate logical (Boolean) values, and the only two instances of that class are expressed as the literals True and False.

Data Types (cont...)

- The int Class: The int class is designed to represent integer values with arbitrary magnitude.
- The float Class: The float class is the sole floating-point type in Python, using a fixed-precision representation.
- The list Class: A list instance stores a sequence of objects. Lists are array-based sequences and are zero-indexed, thus a list of length n has elements indexed from 0 to $n1$ inclusive.
- The tuple Class: The tuple class provides an **immutable version** of a sequence. While Python uses the [ ] characters to delimit a list, parentheses delimit a tuple, with () being an empty tuple.
- The str class: Pythons str class is specifically designed to efficiently represent an immutable sequence of characters, based upon the Unicode international character set.
- The set class: Pythons set class represents the mathematical notion of a set, namely a **collection of elements**, without duplicates, and without an inherent order to those elements.
- The dict class : Pythons dict class represents a **dictionary, or mapping**, from a set of distinct **keys** to associated **values**.

Data Types (cont...)

- **Create variable through Assignment Operator (=):** In Python, you do not need to declare variables before using the variables. The initial assignment creates a variable and links the assigned value to the variable.

Example	Data Type
x = "Hello World"	str
x = 20	int
x = 20.5	float
x = 1j	complex
x = ["apple", "banana", "cherry"]	list
x = ("apple", "banana", "cherry")	tuple
x = range(6)	range
x = "name" : "John", "age" : 36	dict
x = "apple", "banana", "cherry"	set
x = True	bool
x = b"Hello"	bytes

- **Setting the Specific Data Type: (Type Casting:)** You can perform type conversion (or type casting) via built-in functions int(x), float(x), str(x), bool(x)

Example	Data Type
x = str("Hello World")	str
x = int(20)	int
x = float(20.5)	float
x = complex(1j)	complex
x = list(("apple", "banana", "cherry"))	list
x = tuple(("apple", "banana", "cherry"))	tuple
x = range(6)	range
x = dict(name="John", age=36)	dict
x = set(("apple", "banana", "cherry"))	set
x = bool(5)	bool
x = bytes(5)	bytes

Basic Input and Output

- Python provides numerous built-in functions that are readily available to us at the Python prompt.
- Some of the functions like **input()** and **print()** are widely used for standard input and output operations respectively. Let us see the output section first.
- **Function *input()***
- Syntax

```
inp = input('STATEMENT')
```

- Examples

```
value= input ("Enter a string value: ")
num= int(input ("Enter an integer value: "))
float_num= float(input ("Enter an float value: "))
complex_num= complex(input ("Enter a complex number: "))
value = input('Enter an integer: ')
value = int(value)
anotherValue = int(input('Enter another integer: '))
print(value + anotherValue)
x, y = input("Enter a two value: ").split()
x, y, z = input("Enter a three value: ").split()
x = list(map(int, input("Enter a multiple value: ").split()))
```

Basic Input and Output (cont...)

Function *print()*

- Python print() function prints the message to the screen or any other standard output device.
- Syntax:

```
print(value(s), sep= ' ', end = '\n', file=file, flush=flush)
```

  - bluevalue(s) : Any value, and as many as you like. Will be converted to string before printed
  - bluesep='separator : (Optional) Specify how to separate the objects, if there is more than one.Default :' '
  - blueend=end: (Optional) Specify what to print at the end.Default : new line $\backslash n$.
  - bluefile : (Optional) An object with a write method. Default :sys.stdout
  - blueflush : (Optional) A Boolean, specifying if the output is flushed (True) or buffered (False). Default: False

- Example

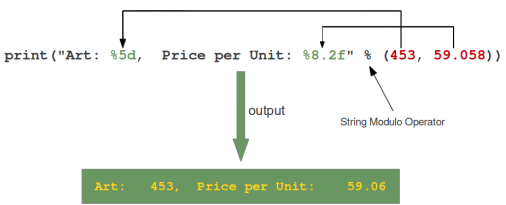
```
#The syntax of the print() function
#print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

print('Welcome to Python!')
print("Welcome to Python!")
print('Welcome', 'to', 'Python!')
print('Welcome\nto\n\nPython!')
print('this is a longer string, so we \
split it over two lines')
print('Sum is', 7 + 3)
print(1, 2, 3, 4)
print(1, 2, 3, 4, sep='*')
print(1, 2, 3, 4, sep='#', end='&')
```

Basic Input and Output (cont...)

Formatted Output

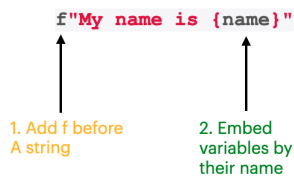
- print() function with modulo operator %: The format string contains placeholders.

```
print('The value of x is %3.2f' %x)
print('%d %s cost $%.2f' % (6, 'bananas', 1.74))
print("%10.3e"% (356.08977)) #3.561e+02
print("%10.3E"% (356.08977)) # 3.561E+02
print("%10o"% (25)) #31
print("%10.3o"% (25)) #031
print("%10.5o"% (25)) #00031
print("%5x"% (47)) #2f
print("%5.4x"% (47)) #002f
print("%5.4X"% (47)) # 002F
```



- f-Strings in Python (3.6+): Formatted string or F-string in Python 3.6+ makes it possible to insert variables into strings.

```
name = "Nick"
print(f"My name is {name}")
#
first_name = "Nick"
last_name = "Jones"
profession = "Software Engineer"
platform = "Codingem.com"
print(f"Hi! I am {first_name} {last_name}, a {profession}. I'm writing a new
      article on {platform}.")
```



Basic Input and Output (cont...)

String Formatting in Python Using Format Specifiers

- Like other programming languages, we can also use format specifiers to format strings in Python.
- Format specifiers are used with the % operator to perform string formatting.
- The % operator, when invoked on a string, takes a variable or tuple of variables as input and places the variables in the specified format specifiers.

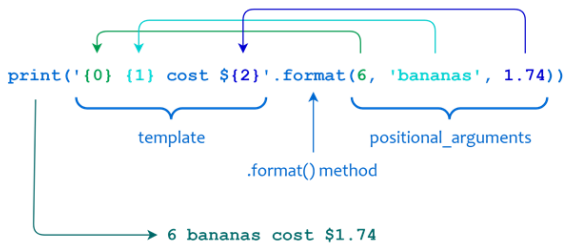
Escape	Description	Example
%d	Decimal integers (not floating point)	`"%d" % 45 == '45'`
%i	Same as %d	`"%i" % 45 == '45'`
%o	Octal number	`"%o" % 1000 == '1750'`
%u	Unsigned decimal	`"%u" % -1000 == '-1000'`
%x	Hexadecimal lowercase	`"%x" % 1000 == '3e8'`
%X	Hexadecimal uppercase	`"%X" % 1000 == '3E8'`
%e	Exponential notation, lowercase “e”	`"%e" % 1000 == '1.000000e+03'`
%E	Exponential notation, uppercase “E”	`"%E" % 1000 == '1.000000E+03'`
%f	Floating point real number	`"%f" % 10.34 == '10.340000'`
%F	Same as %f	`"%F" % 10.34 == '10.340000'`
%g	Either %f or %e, whichever is shorter	`"%g" % 10.34 == '10.34'`
%G	Same as %g but uppercase	`"%G" % 10.34 == '10.34'`
%c	Character format	`"%c" % 34 == '"'`

Basic Input and Output (cont...)

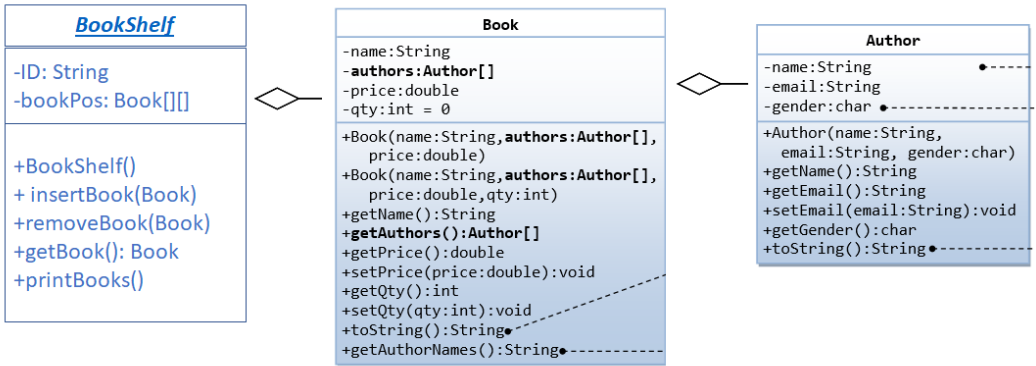
str.format() function :

- Python 3 introduced a new way to do string formatting that was later back-ported to Python 2.7. This makes the syntax for string formatting more regular.

```
print(print('{q} {i} cost ${p}'.format(q=6, i='bananas',p=1.74)))
print('{}{}{}'.format('foo','bar','baz'))
print('{}{}'.format('foo', 'bar', 'baz'))
print('{}{}{}{}'.format('foo','bar','baz')) # error
print('{2}.{1}.{0}/{0}.{1}{1}.{2}'.format('foo', 'bar', 'baz'))
print('{0}/{1}/{2}'.format('foo', 'bar', 'baz'))
print('{0}/{1}/{2}'.format('foo', 'bar', 'baz'))
print('{0}/{1}/{2}'.format('bar', 'baz', 'foo'))
print('{x}/{y}/{z}'.format(x='foo', y='bar', z='baz'))
print('{x}/{y}/{z}'.format(y='bar', z='baz', x='foo'))
```



- Formatting Digits



Garbage Collection

Python Import

- A module is a file containing Python definitions and statements. Python modules have a filename and end with the extension .py.
- Definitions inside a module can be imported to another module or the interactive interpreter in Python. We use the import keyword to do this.

Garbage Collection

- Python creates objects in memory and removes them from memory as necessary.
- When an object is no longer bound to a variable, Python can automatically removes the object from memory.
- This process is called **garbage collection**, it helps to ensure that memory is available for new objects.

Python Operators

Python Operators

- Python operator is a symbol that performs an operation on one or more operands. An operand is a variable or a value on which we perform the operation.
- The Python operators are classified into seven different categories:
  - Arithmetic operators
  - Assignment operators
  - Comparison operators
  - Logical operators
  - Identity operators
  - Membership operators
  - Bitwise operators

Arithmetic operators

- Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, etc.

Python operation	Arithmetic operator	Python expression
Addition	`+`	`f + 7`
Subtraction	`-`	`p - c`
Multiplication	`*`	`b * m`
Exponentiation	`**`	`x ** y`
True division	`/`	`x / y`
Floor division	`//`	`x // y`
Remainder (modulo)	`%`	`r % s`

- Example : Arithmetic operators in Python

```
x = 15
y = 4
print('x + y =',x+y)  # Output:  x + y = 19
print('x - y =',x-y)  # Output:  x - y = 11
print('x * y =',x*y)  # Output:  x * y = 60
print('x / y =',x/y)  # Output:  x / y = 3.75
print('x // y =',x//y) # Output:  x // y = 3
print('x ** y =',x**y) # Output:  x ** y = 50625
```

Python Operators (cont...)

Comparison Operators

- Comparison operators are used to compare values. It returns either True or False according to the condition.

Algebraic operator	Python operator	Sample condition	Meaning
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y

- Example : Comparison operators in Python

```
x = 10
y = 12
print('x > y is',x>y)  # Output:  x > y is False
print('x < y is',x<y)  # Output:  x < y is True
print('x == y is',x==y) # Output:  x == y is False
print('x != y is',x!=y) # Output:  x != y is True
print('x >= y is',x>=y) # Output:  x >= y is False
print('x <= y is',x<=y) # Output:  x <= y is True
```

Python Operators (cont...)

Bitwise Operators

- Bitwise operators are used to perform bit-level operations on (binary) numbers. Bitwise operators act on operands as if they were strings of binary digits.
- Let x = 10 (0000 1010 in binary) and y = 4 (0000 0100 in binary)

Operator	Meaning	Example
&	Bitwise AND	x & y = 0 (0000 0000)
|	Bitwise OR	x | y = 14 (0000 1110)
~	Bitwise NOT	~x = -11 (1111 0101)
^	Bitwise XOR	x ^ y = 14 (0000 1110)
>>	Bitwise right shift	x >> 2 = 2 (0000 0010)
<<	Bitwise left shift	x << 2 = 40 (0010 1000)

- Example : Bitwise operators in Python

```
a = 60            # 60 = 0011 1100
b = 13            # 13 = 0000 1101
c = 0

c = a \& b;       # 12 = 0000 1100
print ("Line 1 - Value of c is ", c)
c = a | b;        # 61 = 0011 1101
print ("Line 2 - Value of c is ", c)
c = a ^ b;        # 49 = 0011 0001
print ("Line 3 - Value of c is ", c)
c = ~a;           # -61 = 1100 0011
print ("Line 4 - Value of c is ", c)
c = a << 2;       # 240 = 1111 0000
print ("Line 5 - Value of c is ", c)
c = a >> 2;       # 15 = 0000 1111
print ("Line 6 - Value of c is ", c)
```

Python Operators (cont...)

Assignment Operators

- Assignment operators are used to assign new values to variables. Assignment statement forms include Tuple- and list-unpacking assignments, Sequence assignments, Extended sequence unpacking, Multiple-target assignments, Augmented assignments

Operation	Interpretation
`spam = 'Spam'`	Basic form
`spam, ham = 'yum', 'YUM'`	Tuple assignment (positional)
`[spam, ham] = ['yum', 'YUM']`	List assignment (positional)
`a, b, c, d = 'spam'`	Sequence assignment, generalized
`a, *b = 'spam'`	Extended sequence unpacking (Python 3.X)
`spam = ham = 'lunch'`	Multiple-target assignment
`spams += 42`	Augmented assignment (equivalent to `spams = spams +`

- There are various compound operators in Python like a $+=$ 5 that adds to the variable and later assigns the same. It is equivalent to a $=$ a $+$ 5.

Python Operators (cont...)

Operator	Meaning	Example	Equivatent to
=	Assignment	x = 3	x = 3
+=	Addition assignment	x += 3	x = x + 3
-=	Subtraction assignment	x -= 3	x = x – 3
*=	Multiplication assignment	x *= 3	x = x * 3
/=	Division assignment	x /= 3	x = x / 3
%=	Modulus assignment	x %= 3	x = x % 3
//=	Floor division assignment	x //= 3	x = x // 3
**=	Exponentiation assignment	x **= 3	x = x ** 3
&=	Bitwise AND assignment	x &= 3	x = x & 3
|=	Bitwise OR assignment	x |= 3	x = x | 3
^=	Bitwise XOR assignment	x ^= 3	x = x ^ 3
>>=	Bitwise right shift assignment	x >>= 3	x = x >> 3
<<=	Bitwise left shift assignment	x <<= 3	x = x << 3

Python Operators (cont...)

Special types of operators

- Python language offers some special types of operators like the identity operator or the membership operator.
- Identity operators : is and is not are the identity operators in Python. They are used to check if two values (or variables) are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

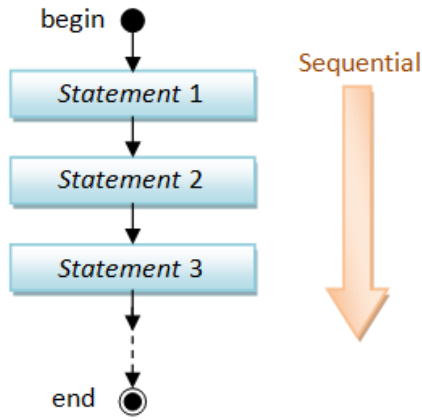
```
x1 = 5
y1 = 5
x2 = 'Hello'
y2 = 'Hello'
x3 = [1,2,3]
y3 = [1,2,3]

print(x1 is not y1) # Output:  False
print(x2 is y2) # Output:  True
print(x3 is y3) # Output:  False
```

- Membership operators:in and not in are the membership operators in Python. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).

```
x = 'Hello world'
y = {1:'a',2:'b'}

print('H' in x) # Output:  True
print('hello' not in x) # Output:  True
print(1 in y) # Output:  True
print('a' in y) # Output:  False
```

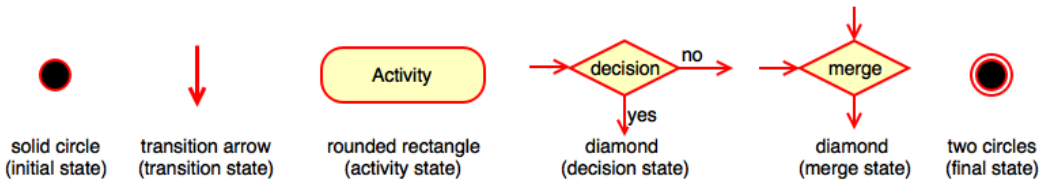
Flow Controls in Python

- **Sequential execution:** Statements in a program execute one after the other in the order in which they are written.
- **Transfer of control:** The control flow of a Python program is regulated by conditional statements, loops, and function calls.
- It includes the *if statement*, *for* and *while* loops
- Raising and handling exceptions also affects control flow



UML activity diagram

- Models the workflow (also called the activity) of a portion of a software system or an algorithm



- An activity diagram is composed of symbols
  1. rounded rectangles (action-state symbol)
  2. arrow (state transition symbol)
  3. diamonds (decision and merge symbols)
  4. small circles (initial and final states symbols)

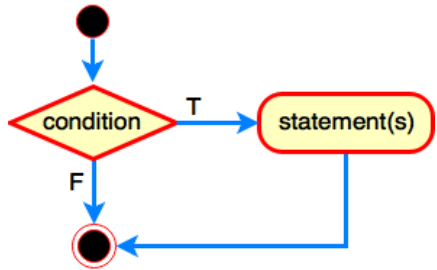
Selection structure

- Three types of selection statements, *if-then*, *if-then-else*, *nested-if*.

if-then statement:

- It performs an action, if a condition is true; skips it, if false.
- It is also known as **single-selection statement**
- **Syntax**

```
if condition:
    # Statements to execute if
    # condition is true
```



- **Example**

```
i = 10
if (i > 15):
    print ("10 is less than 15")
print ("I am Not in if")
```

- **Activity diagram**

Selection structure (cont...)

if-then-else statement:

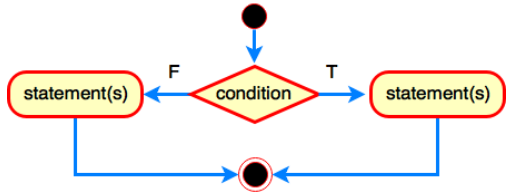
- It performs an action if a condition is true and performs a different action if the condition is false.
- It is also known as **double-selection statement**
- **Syntax**

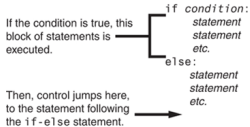
```
if (condition):
   # Executes this block if
   # condition is true
else:
   # Executes this block if
   # condition is false
```

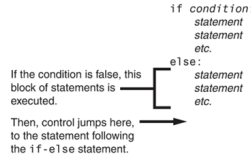
- **Example**

```
i = 20;
if (i < 15):
   print ("i is smaller than 15")
   print ("i'm in if Block")
else:
  print ("i is greater than 15")
  print ("i'm in else Block")
  print ("i'm not in if and not in else Block")
```

- **Activity diagram**





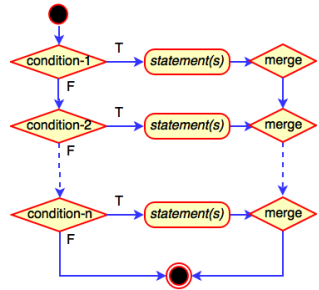


Selection structure (cont...)

if-elif-else ladder

- They performs one of several actions, based on the value of an expression.
- These two statements are known as **multiple-selection statements**
- A multiple-selection statement selects among many different actions (or groups of actions).
- **Syntax**

```
if (condition):
  statement
elif (condition):
  statement
  .
  .
else:
  statement
```



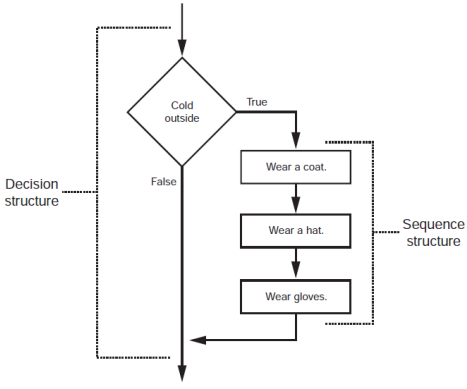
- **Example**

```
i = 20
if (i == 10):
  print ("i is 10")
elif (i == 15):
  print ("i is 15")
elif (i == 20):
  print ("i is 20")
else:
  print ("i is not present")
```

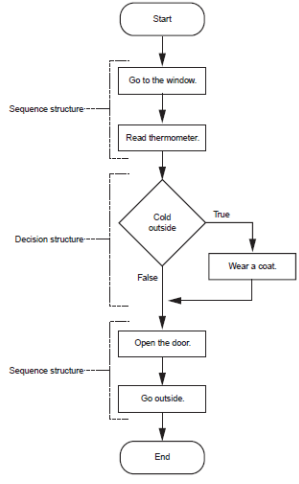
- **Activity diagram**

Selection structure (cont...)

- Programs are usually designed as combinations of different control structures.



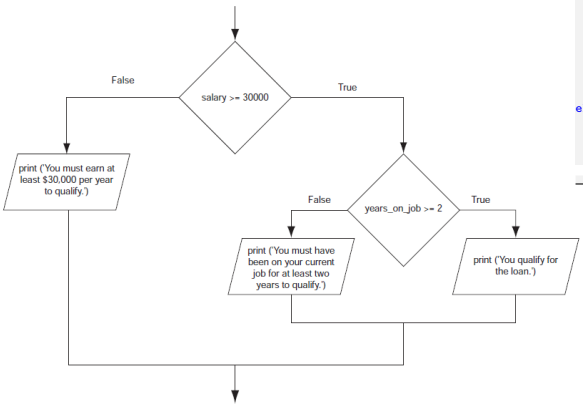
- Figure shows a decision structure with a sequence structure nested inside it.



- Figure shows a flowchart that combines a decision structure with two sequence structures.

Selection structure (cont...)

- consider a program that determines whether a bank customer qualifies for a loan. To qualify, two conditions must exist: (1) the customer must earn at least 30,000 USD per year, and (2) the customer must have been employed for at least two years




```
def loadQualifier():
""" This program determines whether a bank customer qualifies for a loan. """
min_salary = 30000.0 # The minimum annual salary
min_years = 2 # The minimum years on the job
# Get the customer's annual salary.
salary = float(input('Enter your annual salary:'))
years_on_job = int(input('Enter the number of' + 'years employed:'))
# Determine whether the customer qualifies.
if salary >= min_salary:
   if years_on_job >= min_years:
      print('You qualify for the loan.')
   else:
      print('You must have been employed', \
      'for at least', min_years, \
      'years to qualify.')
else:
   print('You must earn at least $', \
   format(min_salary, ',.2f'), \
   ' per year to qualify.', sep='') # Get the number of years on the current
            job.
```

Program Output (with input shown in bold)

```
Enter your annual salary: 35000 Enter
Enter the number of years employed: 1 Enter
You must have been employed for at least 2 years to qualify.
```

Program Output (with input shown in bold)

```
Enter your annual salary: 25000 Enter
Enter the number of years employed: 5 Enter
You must earn at least $30,000.00 per year to qualify.
```

Program Output (with input shown in bold)

```
Enter your annual salary: 35000 Enter
Enter the number of years employed: 5 Enter
You qualify for the loan.
```

Ternary Conditional Operator

- It offers one-line code to evaluate the first expression if the condition is true, and otherwise it evaluates the second expression.

```
<expression1> if <condition> else <expression2>
```

- Examples

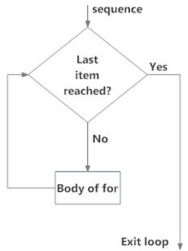
```
age = 15
# this if statement:
if age < 18:
   print('kid')
else:
   print('adult')
# output:  kid
# is equivalent to this ternary operator:
print('kid' if age < 18 else 'adult')
```

```
age = 15
print('kid' if age < 13 else 'teen' if age < 18 else 'adult')
# is equivalent to this if statement:
if age < 18:
   if age < 13:
      print('kid')
   else:
      print('teen')
else:
   print('adult')
```

A Count-Controlled Loop: for-loop

- The for loop in Python is used to iterate over a sequence (*list*, *tuple*, *string*) or other iterable objects.
- **Syntax**

```
for var in iterable:
    # statements
```



- The range() function : we can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers).

- **Examples**

```
print('I will display the numbers 1 through 5.')
for num in [1, 2, 3, 4, 5]:
    print(num)

print('I will display the odd numbers 1 through 9.')
for num in [1, 3, 5, 7, 9]:
    print(num)

for name in ['Winken', 'Blinken', 'Nod']:
    print(name)

# Print a message five times.
for x in range(5):
    print('Hello world')
```

```
for num in range(1, 5):
    print(num)

        for num in [0, 1, 2, 3, 4]:
            print(num)

for num in range(1, 10, 2):
    print(num)
```

```
# Print the numbers 1 through 10
# and their squares.
for number in range(1, 11):
    square = number**2
    print(number, '\t', square)
```

Repetition Structures (cont...)

- Example-1: speed converter
- A program that displays a table of speeds in KPH with their values converted to miles per hour (MPH).
- The formula for converting KPH to MPH is: MPH = KPH * 0.6214
- In the formula, MPH is the speed in miles per hour and KPH is the speed in kilometers per hour.

Program Output

```
KPH         MPH
----------------
60          37.3
70          43.5
80          49.7
90          55.9
100         62.1
110         68.4
120         74.6
130         80.8
```

```
def speedC():
""" This program converts the speeds 60 kph through 130 kph (in 10 kph
        increments) to mph."""
    start_speed = 60 # Starting speed
    end_speed = 131 # Ending speed
    increment = 10 # Speed increment
    conversion_factor = 0.6214 # Conversion factor
    # Print the table headings.
    print('KPH\tMPH')
    print('--------------')
    # Print the speeds.
    for kph in range(start_speed, end_speed, increment):
        mph = kph * conversion_factor
        print(kph, '\t', format(mph, '.1f'))
```

- Generating an Iterable Sequence that Ranges from Highest to Lowest

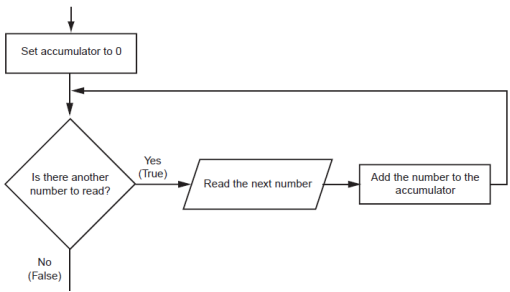
Here is an example of a `for` loop that prints the numbers 5 down to 1

```
for num in range(5, 0, -1):
    print(num)
```

Repetition Structures (cont...)

- Example-2: Calculating a Running Total

- A running total is a sum of numbers that accumulates with each iteration of a loop. The variable used to keep the running total is called an accumulator.




```
1  # This program calculates the sum of a series
2  # of numbers entered by the user.
3
4  max = 5 # The maximum number
5
6  # Initialize an accumulator variable.
7  total = 0.0
8
9  # Explain what we are doing.
10 print('This program calculates the sum of')
11 print(max, 'numbers you will enter.')
12
13 # Get the numbers and accumulate them.
14 for counter in range(max):
15     number = int(input('Enter a number: '))
16     total = total + number
17
18 # Display the total of the numbers.
19 print('The total is', total)
```

Program Output (with input shown in bold)

```
This program calculates the sum of
5 numbers you will enter.
Enter a number: 1 Enter
Enter a number: 2 Enter
Enter a number: 3 Enter
Enter a number: 4 Enter
Enter a number: 5 Enter
The total is 15.0
```

Repetition Structures (cont...)

A Condition-Controlled Loop: while-do

- The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.
- **Syntax**

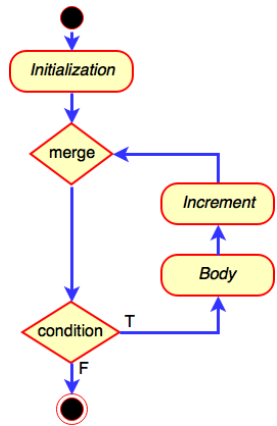
```
while test_expression:
  Body of while
```

- **Example**

```
count = 0
while (count < 3):
  count = count + 1
  print("Hello Geek")
```

- Infinite Loops: An infinite loop continues to repeat until the program is interrupted. Infinite loops usually occur when the programmer forgets to write code inside the loop that makes the test condition false. In most circumstances you should avoid writing infinite loops.

- **Activity Diagram**



Repetition Structures (cont...)

- Sentinels : sentinel is a special value that marks the end of a sequence of values.
- Example: The county tax office calculates the annual taxes on property using the following formula: property tax = property value 3 0.0065

Program Output (with input shown in bold)

```
Enter the property lot number
or enter 0 to end.
Lot number: 100 Enter
Enter the property value: 100000.00 Enter
Property tax: $650.00.
Enter the next lot number or
enter 0 to end.
Lot number: 200 Enter
Enter the property value: 5000.00 Enter
Property tax: $32.50.
Enter the next lot number or
enter 0 to end.
Lot number: 0 Enter
```

```
1  # This program displays property taxes.
2
3  tax_factor = 0.0065 # Represents the tax factor.
4
5  # Get the first lot number.
6  print('Enter the property lot number')
7  print('or enter 0 to end.')
8  lot = int(input('Lot number: '))
9
10 # Continue processing as long as the user
11 # does not enter lot number 0.
12 while lot ! = 0:
13     # Get the property value.
14     value = float(input('Enter the property value: '))
15
16     # Calculate the property's tax.
17     tax = value * tax_factor
18
19     # Display the tax.
20     print('Property tax: $', format(tax, ',.2f'), sep='')
21
22     # Get the next lot number.
23     print('Enter the next lot number or')
24     print('enter 0 to end.')
25     lot = int(input('Lot number: '))
```

Repetition Structures (cont...)

- Nested Loops: A nested loop is a loop that is inside another loop. A clock is a good example of something that works like a nested loop. The second hand, minute hand, and hour hand all spin around the face of the clock.

```
for hours in range(24):
    for minutes in range(60):
        for seconds in range(60):
            print(hours, ':', minutes, ':', seconds)
```

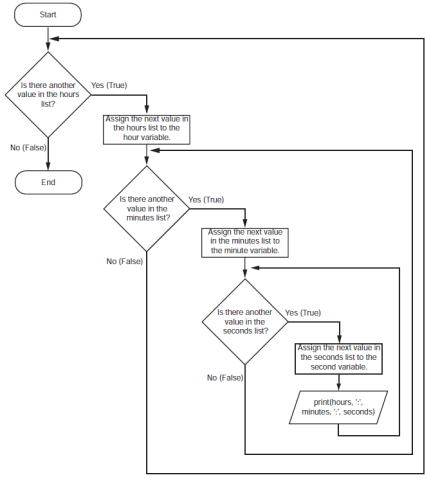
This code's output would be:

```
0:0:0
0:0:1
0:0:2
```

(The program will count through each second of 24 hours.)

```
23:59:59
```

- Hours hand moves once for every twelve movements of the minutes hand: for each iteration of the hours, do twelve iterations of minutes
- Seconds hand moves 60 times for each movement of the minutes hand: for each iteration of minutes, do 60 iterations of seconds



- Figure shows a flowchart for the complete clock simulation program

Repetition Structures (cont...)

- Another Example: averages test scores It is a program that a teacher might use to get the average of each students test scores. The statement in line 5 asks the user for the number of students, and the statement in line 8 asks the user for the number of test scores per student. The for loop that begins in line 11 iterates once for each student. The nested inner loop, in lines 17 through 21, iterates once for each test score.

```
Student number 1
-----------------
Test number 1: 100 Enter
Test number 2: 95 Enter
Test number 3: 90 Enter
The average for student number 1 is: 95.0

Student number 2
-----------------
Test number 1: 80 Enter
Test number 2: 81 Enter
Test number 3: 82 Enter
The average for student number 2 is: 81.0

Student number 3
-----------------
Test number 1: 75 Enter
Test number 2: 85 Enter
Test number 3: 80 Enter
The average for student number 3 is: 80.0
```

```
1  # This program averages test scores. It asks the user for the
2  # number of students and the number of test scores per student.
3
4  # Get the number of students.
5  num_students = int(input('How many students do you have? '))
6
7  # Get the number of test scores per student.
8  num_test_scores = int(input('How many test scores per student? '))
9
10 # Determine each student's average test score.
11 for student in range(num_students):
12     # Initialize an accumulator for test scores.
13     total = 0.0
14     # Get a student's test scores.
15     print('Student number', student + 1)
16     print('-----------------')
17     for test_num in range(num_test_scores):
18         print('Test number', test_num + 1, end='')
19         score = float(input(': '))
20         # Add the score to the accumulator.
21         total += score
22
23     # Calculate the average test score for this student.
24     average = total / num_test_scores
25
26     # Display the average.
27     print('The average for student number', student + 1, \
28           'is:', average)
29     print()
```

Repetition Structures (cont...)

Interrupting Loop Flow - "break" and "continue"

- The break statement breaks out and exits the current (innermost) loop.

```
for val in "string":
  if val == "i":
    break
  print(val)
print("The end")
```

- The continue statement aborts the current iteration and continue to the next iteration of the current (innermost) loop.

```
for val in "string":
   if val == "i":
      continue
   print(val)
   print("The end")
```

- break and continue are poor structures as they are hard to read and hard to follow. Use them only if absolutely necessary.