::Python Part 1 - Overview, Data, Operations, and Sequence Control

|  |  |
| --- | --- |
| **Overview**   * Python is an **interpreted** language. * Python interactive window evaluates statements you type. * **Comments** use **#.** Note that **//** is a numeric operator. * Python source files should use a **.py** file extension. * Python is one of the fastest accepted languages * Python uses **colons** to indicate the end of control statements (e.g., def, if, else, elif, for, while, try, except). For programmers who started with other languages, Python has caused remarks such as it being a "pain in the colon". * **There are many useful libraries**   **Data**   * **No** explicit **declaration** of variables. Variables receive their data type, structure, size, and value from assignment. * **Numeric** values have **unlimited range** * Strings are immutable * Associative arrays are called **dictionaries** * **Slices** of arrays are provided   **Bindings**   * Excluding language definition and implementation, bindings are at runtime.   **Storage Mgt**   * A stack is used for managing   + Return location   + Dictionary of local variables (or a C array in some implementations) * Globals are kept in a dictionary * Values are maintained in dynamic memory * Virtual machine is responsible for managing memory including reference counting and garbage collection   **Scope**   * Uses **static scoping** * Functions can be **nested** inside other functions * By default, **assignments** make variables **local** * Nonlocal references are to either nonlocal variables in surrounding functions or to global variables * **Global** statement allows assignment to a global variable in a function * **Nonlocal** statement allows assignment to a nonlocal variable in a function   **Parameter Passing**   * Uses **by value object reference;** it copies a reference to the object   **Operations**   * Functions can **return multiple values** * **Assignments** support **chaining**, **multi-target**, copying an **array reference**, and **array slices**   **Sequence Control**   * Compound statements are indicated by **indentation** instead of {} as in C or DO END as in PL/I. * Python(as of Python 3) doe **not** provide a case statement. | Browser-based interpreter:  <https://repl.it/languages/python3>   * In the left window, you can paste code. If you press the **run>** button, it will execute it in the right window. * Alternatively, in the right window, you can press **CTRL-V** to paste code into it.   Linux **fox** machines:  python3 *filename*.py   * Do not use the **python** command since it invokes an old version of python. * Note that your programs will be graded based on **python3** on a fox machine.   VDI:  You can launch it from the Start button.  >>> # function numAs counts the number of grades >= 90  def numAs (list):  count = 0  for num in list:  if num >= 90:  count+=1  return count  >>> grades = [85, 90, 65, 99] # list of grades  >>> numAs(grades)  2 |
| **Primitives**  Primitives(integer, float, boolean) in Python are actually objects. Numeric values have unlimited range.  Note that Python doesn't support negative numeric constants. A value, -5, is considered a unary negation operator followed by the numeric constant 5.  Why are numerics represented as objects?  So they aren’t limited by size  Numeric operators:  + addition  - subtraction (or negation if only one operand)  \* multiplication  / division which doesn't truncate  // division which truncates the result  \*\* exponentiation  % modulo operator yields the remainder  ~ bitwise not  << bitwise left shift  << bitwise right shift  & bitwise and  | bitwise or  ^ bitwise XOR  < less than  > greater than  == equal to  >= greater than or equal to  <= less than or equal to  != not equal to  Boolean operators:  or logical or (short circuits)  and logical and (short circuits)  not logical not | |  |  |  | | --- | --- | --- | | **Data Type** | **Example Constants** | **Comments** | | bool | True, False | Constants True and False have values 1 and 0 respectively, but print as **True** and **False**. | | integer | 5, 10, 2,  0x23A (hex constant) | There is an unlimited range of values in Python. | | float | 3.14, 5., 0.005,  1.5e20 |  | | complex | 5j, 1.5e20j | Imaginary numbers |   Examples:  >>> 3 / 2  1.5  >>> 3 // 2  1  >>> 5 % 3  2  >>> 3.0 // 2.0  1.0  >>> # more complex conditional  >>> x = 20  >>> x > 0 and x < 50 # True if x > 0 and x < 50.  True  >>> 0 < x < 50 # More natural comparison of x being between 0 and 50  True |
| Precedence (lowest to highest):  or  and  not  in, not in, is , is not, <, >, <=, >=, ==, !=  |  ^  &  <<, >>  +, -  \*, /, //, %  ~, unary -, unary +  \*\*  When precedences are the same, it evaluates from left-to-right except for exponentiation.  General rules for the data type of a result of numeric operations:  1. If either value is complex, the result is complex.  2. If either value is float, the result is float. If the other is integer, automatically convert it to float.  3. Both values are integer, so no conversion except the result of a / is a float. | >>> 2 + 5 \* 4 - 1 / 2 # First does the multiplication (5 \* 4)  21.5 # yielding 20,  # then the division yielding 0.5,  # then the addition of 2 + 20 yielding 22,  # then the subtraction 22 - 0.5  # giving the result 21.5  >>> 2 \*\* 2 \*\* 3 # First does 2 \*\* 3 yielding 8,  256 # then it does 2 \*\* 8 yielding 256  >>> 5 % 2 \* 9 // 4 # 1 \* 9 // 4  2 # 9 // 4 |
|  |  |
| **Character Strings**   * literals can be surrounded in " or '; can use \ as an escape * strings are immutable * characters within a string can be referenced by subscripts which begin with 0 and are referenced via [] * negatives subscripts start at -1 for the last item in the list, -2 refers to the next to last item * supports slices * len() function is used for length * strings can be concatenated (producing new strings) * integers and floats can be converted to strings using the **str(*value*)** function * strings can be compared using >, <, <=, >=, !=, == * to convert from a string to an int, use the **int(*strValue*)** function #typecasting | >>> print('I heard Clark say, "Python\'s easy to learn."')  I heard Clark say, "Python's easy to learn."  >>> name = "Bob Wire"  >>> name[0]  'B'  >>> name[7]  'e'  >>> name[-1] # last character  'e'  >>> name[-2] # next to last character  'r'  >>> name[0:3]  'Bob'  >>> name[4:] #position 4 to the end  'Wire'  >>> name[:3] #position 0 to 3  'Bob'  >>> len(name)  8  >>> name[-10] #stops execution here  IndexError: string index out of range  >>> # Comparison  one = "apple a day"  two = "applet"  >>> one > two  False |
| **Character String Operators**  **in** checks to see if a string is in another string  **not in** checks whether a string is not in another string  **+** concatenation  **\*** replication  **%** string formatting  We will discuss pattern matching in part 3. | >>> # in operator  one = "apple a day"  two = "apple"  >>> two in one  True  >>> "APPLE" in one #case sensitive  False  >>> # concatenation uses + to concatenate two strings  bob2 = name[:3] + " L Head"  print (bob2)  **Bob L Head**  >>> # convert a number to a string  iq = 40  iqStr = bob2 + " has an IQ of " + str(iq)  >>> iqStr  **Bob L Head has an IQ of 40**  >>> "." \* 3  '...'  >>> name = "Bob"  >>> name\*3 #returns a string, doesn’t change name  'BobBobBob'  >>> x = name\*3  >>> x #will print the same as above  >>> name = "Bob"  wage = 10.5  yearHire = 2016  >>> "Name is %s, hourly wage is %.1f, year hired is %i" % (name, wage, yearHire)  'Name is Bob, hourly wage is 10.5, year hired is 2016'  # .1f float val  # las % is the actual string formatting |
| **Character String Methods**  **isalpha()** True if string only contains alphas  **isdecimal()** True if string only contains numbers  **upper()** Returns uppercase values for the string  **lower()** Returns lowercase values for the string  **strip()** Returns a string with all leading and trailing white space (blanks, tab, newline) removed | >>> phone = "hello"  >>> phone.isalpha()  True  >>> phone.isdecimal()  False  >>> phone = "2102223333"  >>> phone.isalpha()  False  >>> phone.isdecimal()  True  >>> phone = "210.2223333"  >>> phone.isdecimal()  False  >>> name = "Harry Fingers"  >>> name.upper()  'HARRY FINGERS'  >>> name.lower()  'harry fingers'  >>> name = " Harry Fingers\n"  >>> name.strip()  'Harry Fingers' |
| **Character String Methods - split()**  **split()** Creates a list of the tokens from the string using one or more delimiters  split(delim,n) Creates a list of n tokens with the remaining text placed in the last element of the list. The resulting list usually has n+1 entries unless there isn't any remaining text.  split() can be passed a different delimiter.  the regex module provides a more powerful split:  **re.split(***regExpr, string***)**  *regExpr* is a regular expression (see Python Part 3). *string* is the string to be split. | >>> # By default, split uses white space characters (blank, tab, \n)) as its delimiters.  text1 = "My phone number is 210-555-4444."  tokenM = text1.split()  >>> tokenM  ['My', 'phone', 'number', 'is', '210-555-4444.']  >>> # Using a slash as a delimiter  text2 = "/home/clark/cs3723"  folderM = text2.split('/')  >>> folderM  **['', 'home', 'clark', 'cs3723']**  >>> # use split() to get the area code and place the rest in the second element  text3 = "210-555-444"  tokenM = text3.**split**('-', 1)  >>> tokenM  **['210', '555-444']**  tokenM = text3.split (‘1’, 2)  >>> tokenM  [’210’ , ‘555’, ‘444’]  >>> # use re.split(regex, string) from the re module  # Split on semicolon, comma, period or space. Also ignore 0 to many  # spaces after the delimiter.  import re  text4 = "He loved playing basketball; however, he hated watching it on TV."  wordM = re.split("[;,\s\.]\s\*", text3) # \* = zero or more  >>> wordM  ['He', 'loved', 'playing', 'basketball', 'however', 'he', 'hated', 'watching', 'it', 'on', 'TV', ''] |
| **Character String Methods - centering and justifying**  **rjust(***size, padChar***)**  right justifies the string to *size* bytes. padChar is an optional padding character.  **ljust(***size, padChar***)**  left justifies the string to *size* bytes. padChar is an optional padding character.  **center(***size, padChar***)**  centers the string to *size* bytes. padChar is an optional padding character. | # Right justify to 20 characters using the default space as the padding character  >>> name = 'May King'  >>> name.rjust(20)  ' May King'  # Right justify to 20 characters using the a period as the padding character  >>> name.rjust(20, '.')  '............May King'  # left justify to 20 characters using the a '\*' as the padding character  >>> name.ljust(20,'\*')  'May King\*\*\*\*\*\*\*\*\*\*\*\*'  # center within 20 characters using the a '\*' as the padding character  >>> name.center(20,'\*')  '\*\*\*\*\*\*May King\*\*\*\*\*\*' |
| **Data Structures - lists**   * heterogeneous arrays * unbounded; grows * creation surrounded by []; separate entries with commas * subscripts begin with 0 and are referenced via [] * negatives subscripts start at -1 for the last item in the list, -2 refers to the next to last item * supports slices * len() function is used for length * existing entries can be modified by assigning to an individual element * new entries can be added by either constructing a new list via **concatenation** or using the **append** function #concatenation more expensive than append. Concatenation changes one variable * prints the entire contents when referenced without subscripts * provides **for in** to iterate over contents * operators: in, not in | >>> # create a list of grades  >>> grades = [100, 90, 95, 75]  >>> grades[1]  90  >>> grades[3]  75  >>> grades[-1]  75  >>> grades[-2]  95  >>> grades[0:2] # is a slice which begins at [0] and goes up to (but not including) [2]  [100, 90]  >>> grades[1:] # a slice beginning at [1] to the last element  **[90, 95, 75]**  >>> grades[:2] # a slice [0:2]  [100, 90]  >>> len(grades)  4  >>> # **replace** the 95 with a 100  grades[2] = 100  # add another grade onto the list using + to  # **concatenate** two lists creating a **new list**  **# myGrades = grades🡪 both point to the same thing**  **# If continued, myGrades would not change. Makes brand new list**  grades = grades + [90] # grades is now [100, 90, 100, 75, 90]  print(grades)  [100, 90, 100, 75, 90]  >>> # **append** another grade using the append method  >>> grades.append(55)  >>> grades  [100, 90, 100, 75, 90, 55]  >>> # individually print each value  for grade in grades:  print (grade)  100  90  100  75  90  55  >>> 100 in grades  True  >>> 99 in grades  False |
| **Data structures - tuples**   * similar to lists except immutable * creation surrounded by () | >>> # create a tuple that cannot be changed  >>> student = ("Anita Break", 0.01, "CS")  # if we try to change the gpa to 0.02, we would get an assignment error  student[1] = 0.02  TypeError: 'tuple' object does not support item assignment  >>> # we can recreate student using concatenation. Notice that we are concatenating tuples  # instead of changing an element of student.  student = student[0:1] + (0.02,) + student[2:] #the , makes it a tuple  print (student)  ('Anita Break', 0.02, 'CS') |
| **Data structures - dictionary**   * associative array * surround the dictionary in {}; separate key from value using a colon; separate different dictionary entries using commas * use [] when referencing elements * provides **for in** to iterate over contents referenced with **items()**, **keys()** or **values()** methods * **get(*key*,*notExistsValue*)** method is provided to access an entry which might not exist * len(*dictionary*) provides the number of elements * operators: in, not in keys(), values() * entries can be **added** by simply assigning to the dictionary:   dictionary[*keyValue*] = value | >>> # Create a contact dictionary  contactM = {"Board, Bill": "830-222-2222"  , "Board, Peg": "830-222-3333"  , "Barr, Ted E": "210-555-1111"}  contactM["Board, Emory"] = contactM["Board, Peg"]; #copy Peg's phone to Emory  # access the key and value, calling them name and phone  for name, phone in contactM.items():  print (name, phone)  Output:  Board, Emory 830-222-3333  Board, Bill 830-222-2222  Barr, Ted E 210-555-1111  Board, Peg 830-222-3333  >>> # try getting some entries  >>> contactM["Board, Bill"]  830-222-2222  >>> contactM["Barr, Candy"]  KeyError: 'Barr, Candy'  >>> # try getting some entries  >>> contactM.get("Board, Peg", "NF")  '830-222-3333'    >>> contactM.get("Barr, Candy", "NF") #returns whatever after , if not found  'NF'  >>> # convenient function to pretty print dictionary entries (sorted by key)  import pprint #SORTS  pprint.pprint(contactM)  {'Barr, Ted E': '210-555-1111',  'Board, Bill': '830-222-2222',  'Board, Emory': '830-222-3333',  'Board, Peg': '830-222-3333'}  >>> "Board, Bill" in contactM  True  >>> "830-222-2222" in contactM.values()  True |
| **Assignments**  **Compound Assignments**  **+=, -=, \*=, /=, //=**  **|=, &=**  **Notice that Python doesn’t provide ++ nor --. (they would change val of var in c)**  **Chain Assignments**  **=** can be used to chain assignments  **Multi-target**  separate the targets with commas | >>> x = 5  x += 10  print(x)  15  >>> z = 5  w = x = y = z # Assign z to y, x, and w  print (w, x, y, z)  5 5 5 5  >>> # interchange values using multi-target  x = 5  y = 10  x,y = y, x  print ("x=", x, "y=", y)  x=10 y= 5  >>> # Suppose fruit is a list  fruit = ["orange", "grape", "apple"]  o, g, a = fruit # unpacks the list and  # assigns "orange" to o,  # "grape" to g, and  # "apple" to a  print("1.", o, g, a)  o, \*rest = fruit # The \* says to give anything else  # to the variable rest. Assigns  # "orange" to o and it assigns  # ["grape", "apple"] to rest  print ("2.", o, rest)  fruit = ["orange", "grape", "apple", "clark"]  one, \*two, three = fruit  # Assigns "orange" to one,  # ["grape", "apple"] to two,  # "clark" to three  print ("3.", one, two, three)  **Output:**  1. orange grape apple  2. orange ['grape', 'apple']  3. orange ['grape', 'apple'] clark |
| **Assignments - Array**   * **Copy reference** via array assignment * **Slice assignment** | >>> # Copying a Reference to an Array:  fruit = ["orange", "grape", "apple", "clark"]  profs = fruit  profs[0] = "maynard"  >>> print(fruit)  ["maynard", "grape", "apple", "clark"]  >>> print(profs)  ["maynard", "grape", "apple", "clark"]  #both change(reference to same list)  >>> # Assignment to a slice of an array:  L = [100, 101, 102, 103, 104, 105, 106]  L[2:5] = [22,33,44] # replace items 2, 3, and 4  >>> print (L)  [100, 101, 22, 33, 44, 105, 106]  >>> L[2:5] = [] # remove items 2, 3, and 4  >>> print (L)  [100, 101, 105, 106] |
| **Conditional Statements**  Python has three types of conditional statements.  if-then construct:  if *condExpr* :  *indentedBodyStatements*  if-then-else construct:  if *condExpr* :  *indentedBodyStatements*  else:  *indentedBodyStatements*  if-then-else-if construct:  if *condExpr* :  *indentedBodyStatements*  elif *condExpr* :  *indentedBodyStatements* | >>> # if-then  if grade >= 60:  print("you passed")  >>> # if-then-else with nested if-then-else  if command == "WITHDRAWAL":  if dBalance > 0:  dBalance -= dAmount  print("New balance:", dBalance)  else:  print("Insufficient funds:", dAmount)  elif command == "DEPOSIT":  dBalance += dAmount |
| **Conditional Statements - Case**  Python doesn't provide a case construct. | does that  http://www.wackystock.com/details/1084861-clipart-businessman-wearing-a-clothespin-on-his-nose-due-to-smell---royalty-free-vector-illustration-by-dennis-cox-at-wackystock.jpg ? |
| **Iteration - while**  while-loop construct:  while *condExpr* :  *indentedBodyStatements*  Python also has break and continue statements. The break isn't labeled.  **while**  also supports an **else** statement**.** If the while exits due to a false *condExpr*, the else is executed. If a **break** statement causes the termination, the **else** statement does not execute. | >>> # Example input loop  while True:  print("Please type your name.");  name = input()  if name != "your name":  break  print ("Good morning,", name)  >>> # enter numbers until 0 is entered. Show the sum unless a  # bad number is entered.  num = '1'  sum = 0  while num != '0':  print("Enter a positive number (zero to terminate).")  num = input()  if num.isdecimal() != True:  break #wouldn’t execute the else  sum += int(num)  else:  print ("Sum is", sum) |
| **Iteration With Control Variables**  for *controlVar* in range(*begVal,endVal,stepVal*):  *indentedBodyStatements*  The *stepVal* can be positive or negative.  **for** loops also have an optional **else** statement. | >>> # counting from 0 to 4  for i in range (0, 5):  print(i)  print('outside', i)  **Output:**  0  1  2  3:  4  outside 4  >>> # counting from 0 to 4 by 2  for i in range (0, 5, 2):  print(i)  print('outside', i)  **Output:**  0  2  4  outside 4  # counting from 5 to 1 by -1  for i in range (5, 0, -1):  print(i)  print('outside', i)  5  4  3  2  1  outside 1 |
| **Is the Python for statement impacted by changes to the control variables?** | >>> #1 using additional control variable  e = 5  incr = 2  for i in range (0, e, incr): #temporary i got incremented  print (i)  print('outside', i)  **0**  **2**  **4**  **outside 4**  >>> #2 changing i  e = 5  incr = 2  for i in range (0, e, incr): #incrementing by 2  print (i)  i += 1  print('outside', i)  **0**  **2**  **4**  **outside 5**  **#assign temporary i o i then increments it.**  >>> #3 changing e  e = 5  incr = 2  for i in range (0, e, incr):  print (i)  e = 9  print('outside', i)  **0**  **2**  **4**  **outside 4**  **#what you do inside for loop doesn’t affect amount of iterations** |
| **Iteration with Control Variables - Based on other Objects**  for *controlVar* in *object*:  *indentedBodyStatements* | # example 1  fruitM = ["apple", "orange", "banana"]  for fruit in fruitM:  print (fruit)  Output:  apple  orange  banana  # example 2  i = 0  fruitM = ["apple", "orange", "banana"]  fruitM.append("tangerine")  for fruit in fruitM:  print (fruit)  i += 1  if i < 3:  fruitM.append("grape")    print (fruitM)  Output:  apple  orange  banana  tangerine  grape  grape  ['apple', 'orange', 'banana', 'tangerine', 'grape', 'grape'] |
| for *controlVar1, controlVar2* in *listOfTuple*:  *indentedBodyStatements*  If each element is a tuple, you can specify multiple control variables, representing each of the values in the tuple. | contactM = {"Board, Bill": "830-222-2222"  , "Board, Peg": "830-222-3333"  , "Barr, Ted E": "210-555-1111"}  contactM["Board, Emory"] = contactM["Board, Peg"]; #copy Peg's phone to Emory  # access the key and value, calling them name and phone  for name, phone in contactM.items(): #returns a list of tuples  print (name, phone)  Output:  Board, Emory 830-222-3333  Board, Bill 830-222-2222  Barr, Ted E 210-555-1111  Board, Peg 830-222-3333 |
| **EOF Exception prog 5**  When reading data from a file, Python throws an EOFError when end of file is encountered.  You will probably need this input loop in your Python programming assignments. | >>> # reading data from a file will eventually raise an EOFError  while True:  try:  inputLine = input() # reads from stdin  except(EOFError):  break  do something with it |
| **print() Function**  With Python 3, print changed syntax from Python 2. Previously in Python 2, print was a statement with this syntax: print *listOfItems*  New syntax: print (listOfItems)  With Python 3, it is a function with a variable number of parameters surrounded by parentheses.  By default, the print() function prints a line to stdout. It has some keyword parameters:  end=*endStr* specify what is printed at the end of this print line. By default, it is a \n. To suppress a new line, use end=""  sep=*sepStr* specify what is printed between parameters on this print line. By default, this is a single space.  file=*fileObj* specify which file to use. By default it is **stdout**. | >>> # print numbers across a line with two characters per number  for i in range (1, 20):  print ("%2i"%(i), end="") #sets endstr to emptystring  **1 2 3 4 5 6 7 8 910111213141516171819**  >>> # print numbers and their square across a line.  # Print a space at the end of each print  for i in range (1, 10):  print (i, i\*i, end=" ") #prints two values #without end — prints new line  **1 1 2 4 3 9 4 16 5 25 6 36 7 49 8 64 9 81**  >>> # print numbers and their square across a line.  # Print a space at the end of each print. Print ".." as separators.  for i in range (1, 10):  print (i, i\*i, sep="..", end=" ")  **1..1 2..4 3..9 4..16 5..25 6..36 7..49 8..64 9..81**  # |
| **Continuation of statements**  Some Python statements can be continued to the next line implicitly. This is possible when continuing something that was begun with an open delimiter (e.g. "(", "[", "{"). Otherwise, you can continue to the next line by ending a line with a backslash. | >>> grades = [100, 90, 95, 75, 80, \  90, 35]  >>> state = "TX"  vehicleType = "TRUCK"  if state == "TX" \  and vehicleType == "TRUCK":  print("it is")  else:  print("nope")  x =100000000000000101000100 \ #needs this to continue statement below  + 3434344444 |
|  |  |