Python

* Bits per digit for Python integers
* floats follow IEEE standard of floating-point literals (16/17 digits of precision, 10e+-304)

>>>6775789976587587573425568708.7876547488888888888777777777777622122456789078989

6.775789976587587e+27

* Inf vs overflow

>>> y = 1e200

>>> y \*= y

>>> y

inf

>>> y = 1e200\*1e200

>>> y

inf

>>> x = 1e200

>>> y = x \* x

>>> y

inf

>>> y = x\*\*2

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

OverflowError: (34, 'Result too large')

y /= y

>>> y

nan # not a number

* floating point operations are inaccurate
* There exists real value and representative value for floats
* 1/3 has a representative value of 0.3333333333333333148296162562473909929394721984863281250
* floats always have 24 bytes
* float(‘inf’), float(‘nan’) is not case sensitive
* math.nan, math.inf in math module introduced in Python3

String literals

* Whitespaces stored
* Minimum 49 bytes for string literals
* No str\_info
* No boundaries for string size
* C uses 1 byte for a character
* Python uses 49 bytes for ''
* ASCII/UTF value of strings
* Practice python
* Interned memory for -5 to 256 in interpreter (shell)
* Programmes (script) intern more values during compiler optimisation
* Peep hole optimisation?

28.08.19

* Why is size of int 28 bytes??
* Why is 0 24 bytes?
* Reference count, object, linked list?
* GOOGLE IT!
* Also find out why 1/3 is 0.333333….413928
* CamelCase and other conventions
* Assignment of name and value (packing and unpacking)
* Multiple assignment
* n, m = 10, 20
* Packed 4 values into points

points = (10, 20, 30, 40)

>>> print(points, type(points))

((10, 20, 30, 40), <type 'tuple'>)

>>> p1, p2, p3, p4 = points

>>> print(p1, p2, p3, p4)

(10, 20, 30, 40)

>>> print(type(p1), type(p2), type(p3), type(p4))

(<type 'int'>, <type 'int'>, <type 'int'>, <type 'int'>)

* Needs exact number of containers to unpack

a, b, c = points

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ValueError: too many values to unpack

* Printing values of names that haven’t been bound to is an error (NameError)
* Multiple assignment with data structures
* 63 keywords in Python

help> keywords

Here is a list of the Python keywords. Enter any keyword to get more help.

and elif if print

as else import raise

assert except in return

break exec is try

class finally lambda while

continue for not with

def from or yield

del global pass

* Cannot use keywords as identifiers
* DO NOT DO THAT
* If redefined, del <keyword> deletes the local redefined name <keyword>
* Releases the new binding using del

29.08.19

# Operators and Expressions

* % operator doesn’t work on complex numbers, even if both are complex
* True maps to 1, False maps to 2
* Can be used for arithmetic operators
* None cannot be used

print(int(None))

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: int() argument must be a string or a number, not 'NoneType'

* bool values of non-zero ints and floats are True and of 0 is False
* bool values of empty string, empty list, empty tuple, empty dictionary, complex 0, None are False
* Short circuit evaluation – MUST KNOW WHAT IS FALSE

## Division

1. Classic division

* Both operands of same type: floor division
* Python 2 supports

1. True division
2. Floor division

Python 2

>>> 1/2

0

>>> 1.0/2

0.5

>>> 1/2.0

0.5

>>> 1.0/2.0

0.5

>>>

To force true division in Python 2

from \_\_future\_\_ import division

>>> 1/2

0.5

Python 3

Separates floor and true division

>>> 1/2

0.5

>>> 1.0/2

0.5

>>> 1/2.0

0.5

>>> 1.0/2.0

0.5

>>> 1//2

0

>>> 1.0//2

0.0

>>> 1//2.0

0.0

>>> 1.0//2.0

0.0

>>>

## Constants from modules

* Can create module file and import names and constants

## Chained Assignment

* Top element duplicated (temporary PyObject created, the reference count is increased, and then the PyObject is released) in heap/stack
* test() stored in stack
* Compiler puts main code in a C compiler function call, stack exists
* Use disassembly module to check

>>> def test():

... n1 = n2 = n3 = n4 = 13

...

>>> dis.dis(test)

2 0 LOAD\_CONST 1 (13)

2 DUP\_TOP

4 STORE\_FAST 0 (n1)

6 DUP\_TOP

8 STORE\_FAST 1 (n2)

10 DUP\_TOP

12 STORE\_FAST 2 (n3)

14 STORE\_FAST 3 (n4)

16 LOAD\_CONST 0 (None)

18 RETURN\_VALUE

* Duplicate and then store
* Associativity and precedence of operators
* Precedence rules decide order of operators

2 \* 3 – 4

* Same set of operators of same precedence: resolved by associativity

2 - 3 - 4

* Assignment is right associativity
* Expression are left associativity
* Use parentheses to override precedence and associativity

## Augmented Assignment

* Increment/decrement operators in Python
* No prefix (++n) and postfix (n++) operators in Python
* ++10 just returns 10, --10 just returns 10
* - \* - gives +
* Python built to be simple
* Initially, in C, all four increments operators were different in compilers (n = n + 1; n += 1; n++; ++n)
* Newer compilers with code optimisation generated the same intermediate code
* Impact on Python was the loop; loop was redesigned
* Python uses +=, -=, /=, \*=, \*\*=, %=

## Comparison/Relational

* <, >, ==, !=, <> (only Python 2), >=, <=

>>> print('S' > 'A')

True

* Uses Unicode values of characters, lexographically

print(ord('A'))

65

>>> print(ord('S'))

83

* Use function ord() to check Unicode values

ord(c, /)

Return the Unicode code point for a one-character string.

* Use function chr() to get character for Unicode values

chr(i, /)

Return a Unicode string of one character with ordinal i; 0 <= i <= 0x10ffff.

* Can have chained comparison also; not good programming style

>>> 2 > 1 > 0

True

* Opcodes use rotate

2 0 LOAD\_CONST 1 (2)

2 STORE\_FAST 0 (n1)

3 4 LOAD\_CONST 2 (3)

6 STORE\_FAST 1 (n2)

4 8 LOAD\_CONST 3 (4)

10 STORE\_FAST 2 (n3)

5 12 LOAD\_GLOBAL 0 (print)

14 LOAD\_FAST 0 (n1)

16 LOAD\_FAST 1 (n2)

18 DUP\_TOP

20 ROT\_THREE

22 COMPARE\_OP 0 (<)

24 JUMP\_IF\_FALSE\_OR\_POP 32

26 LOAD\_FAST 2 (n3)

28 COMPARE\_OP 0 (<)

30 JUMP\_FORWARD 4 (to 36)

>> 32 ROT\_TWO

34 POP\_TOP

>> 36 CALL\_FUNCTION 1

38 POP\_TOP

40 LOAD\_CONST 0 (None)

42 RETURN\_VALUE

* This is better

>>> 2 > 1 and 1 > 0

True

* Opcodes are better

2 0 LOAD\_CONST 1 (2)

2 STORE\_FAST 0 (n1)

3 4 LOAD\_CONST 2 (3)

6 STORE\_FAST 1 (n2)

4 8 LOAD\_CONST 3 (4)

10 STORE\_FAST 2 (n3)

5 12 LOAD\_GLOBAL 0 (print)

14 LOAD\_FAST 0 (n1)

16 LOAD\_FAST 1 (n2)

18 COMPARE\_OP 0 (<)

20 JUMP\_IF\_FALSE\_OR\_POP 28

22 LOAD\_FAST 1 (n2)

24 LOAD\_FAST 2 (n3)

26 COMPARE\_OP 0 (<)

>> 28 CALL\_FUNCTION 1

30 POP\_TOP

32 LOAD\_CONST 0 (None)

34 RETURN\_VALUE

## Logical Operators - Introduction

* and, or, not
* Boolean algebra for operators

>>> True and False

False

>>> True or False

True

>>> 1 == 2 or 1 == 3

False

* Use operators for numbers as well

>>> 10 and 20 and 30

30

>>> 10 and 20

20

>>> 10 and 20 or 30

20

>>> 10 and 'S' or 30 and 40

'S'

>>> 'S' or 30 and 10

'S'

## Short circuit evaluation

>>> 5 or 6 or 7

5

* First 5 is evaluated as True, rest of expression is True because or operator
* The one that was evaluated first is returned

>>> 6 or 2 or 1

6

* In and operators, the values until the end are evaluated as all must equal True

>>> 10 and 3

3

Q: Give the logical opposites of these conditions: (Do and, or change?)

1. a > b

a <= b

1. a >= b

a < b

1. a >= 18 and day == 3

a < 18 or day != 3 (?)

1. a >= 18 and day != 3

a < 18 or day == 3

30.08.19

## Logical Operators

Logical **and** operator

* Returns first False value
* Otherwise returns last True value

>>> 5 and 0 and 6

0

>>> 30 and 's' and 77

77

* Inference: left-associativity of **and** operator
* If the left hand side is False, the rest of the expression is not evaluated
* Only if it is evaluated fully, it returns True

>>> print(False and 0)

False

>>> print(0 and False)

0

Logical **or** operator

* Returns first True value
* Otherwise returns last False value
* Inference: left-associativity of **or** operator

>>> print(True or 1)

True

>>> print(1 or True)

1

Priority of **and** and **or** operators

* a **and** b **or** c
* **and** has higher priority over **or**
* **not** has the highest priority
* Note: exponentiation (\*\*) is right associative

## Short Circuit Evaluation

* The next expression is evaluated based on the result of the first one
* x >= 2 and (x/y) > 2
* If x is less than 2, the whole expression evaluates to False and the second term is disregarded
* Python detects that there is nothing to be gained by evaluating the rest of the expression
* Short circuit evaluation
* Guardian pattern

>>> x = 6

>>> y = 2

>>> x >= 2 and (x/y) > 2

True

>>> x = 1

>>> y = 0

>>> x >= 2 and (x/y) > 2

False

>>> x = 6

>>> y = 0

>>> x >= 2 and (x/y) > 2

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ZeroDivisionError: division by zero

* Need a condition to guard division by 0
* Just like base condition for recursion to terminate, we need a guard condition (guard evaluation)
* Introduce y != 0

>>> x = 6

>>> y = 0

>>> x >= 2 and y != 0 and (x/y) > 2

False

>>> x = 6

>>> y = 0

>>> x >= 2 and (x/y) > 2 and y != 0

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ZeroDivisionError: division by zero

Q: Evaluate the following expressions (USE PRECEDENCE)

1. 0 or False and True

False

1. 10 and 5 or 6

5

1. 6 or 5 and 10

6

1. False and True or True

True

## Specialty of % Operator

* Evaluation of % expressions: returns integers with the sign of the divisor

>>> 7 % 3

1

>>> -7 % 3

2

>>> 7 % -3

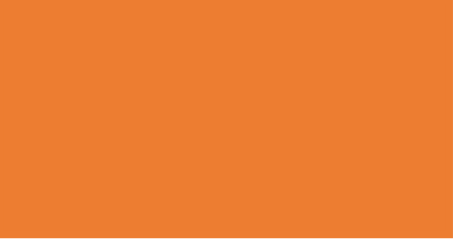
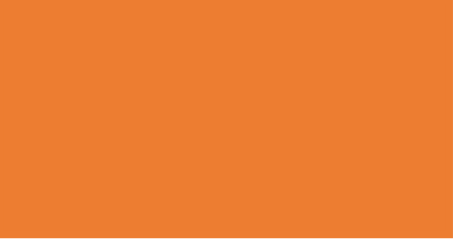
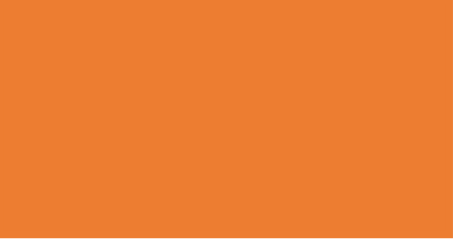
-2

>>> -7 % -3

-1

* 7 % 3 returns 1; 6 is the largest multiple of 3 that is less than or equal to 7
* -7 % 3 returns 2; -9 is the largest multiple of 3 that is less than or equal to -7; baskets of 3 are added to -7 until a positive number is attained

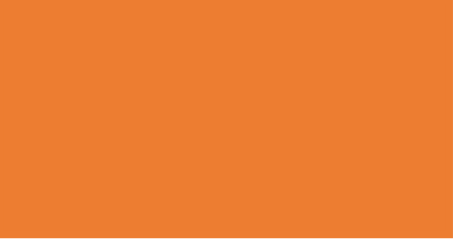
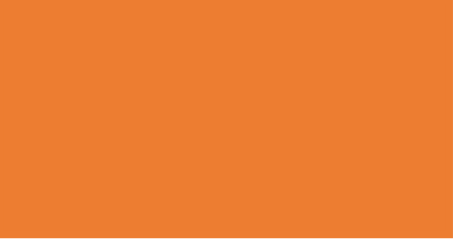






* 7 % -3 returns -2; 9 is the smallest multiple of -3 that is greater than or equal to 7; baskets of -3 are added to 7 until the first negative number is attained
* -7 % -3





Q: Find an expression for the relationship between % and //

* 10 % 3 = 1
* 10 // 3 = 3
* a % b = c
* a // b = d
* (d \* b) + c = a
* ((a // b) \* b) + (a % b) = a

03.09.19

Q: Write a programme that allows the user to enter a four digit binary number and displays the value in base 10. Each binary digit should be entered one per line, starting with the leftmost digit, as shown below.

Enter leftmost digit: 1

Enter next digit: 0

Enter next digit: 0

Enter next digit: 1

The number is 9.

Note: try not to use the traditional approach

* Note: Java, C# and Python accept inputs only in strings
* main arguments are strings (public static void main)
* Unlike C++, where command line arguments are ints
* Why does Python read input as strings?

>>> n = '10.0'

>>> int(n)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ValueError: invalid literal for int() with base 10: '10.0'

* Must first convert to float and then int

>>> int(float(n))

10

* int() doesn’t accept a string of a float, but can truncate a float

>>> n = ' 20 '

>>> int(n)

20

* Any number of whitespaces allowed
* Why bases from 2-36 only?
* Why 2-32 in Python2?

Q: Convert a character to its Unicode encoding

# Chapter 3 – Control Structures

* Modelling real-world scenarios based on decisions
* Modification of control flow of programme
* Comparing strings

>>> print("a" > "zebra")

False

>>> print("zebra" > "elephant")

True

>>> print("dot" > "dog")

True

>>> print("cat" > "cattle")

False

* Operators

6/3//3-(1+2)\*\*2 == 10%3-1-3\*\*2/1

2//3-9 == 1-1-9

-9 == -9

True

* Leader suite???
* if/else/elif
* elif ladder
* Nested else
* No action else – pass keyword

>>> n = 10

>>> if n:

... print(n)

... else:

... pass

...

10

* pass used for dangling ifs
* pass is a fall through statement

>>> x = 10

>>> if x:

... pass

... print("sup")

... else:

... print("else")

...

sup

04.09.19

* Check if number is even or odd
* Use modulus operator

Q: Assume you visit a restaurant, what meal do you choose? Write a programme to model

Q: Visit a place in summer, make a decision based on road/air/water

Q: University scenario – event – what snacks?

05.09.19

Nested vs Flat Structures

* Nesting of data structures and control structures is possible
* Problems with nesting: cyclomatic complexity, code readability reduced
* LOC: lines of code
* FP: function points
* CC: cyclometric complexity
* Arrow anti pattern
* Cleanup happens vertically far from the original cause
* Deadlock situation: no release of resources; programme hangs
* CC increases when decisions/branches are added to code
* Flattening process
* Radon is a package (module) that gives you CC
* pip install radon

Ternary operator

* In C++, **(n1 > n2)? n1: n2**

>>> n1 = 10

>>> n2 = 8

>>> n1 if (n1 > n2) else n2

10

* Expression evaluated before assignment

>>n1 = 3

>>> n2 = 5

>>> big = n1 if n1 > n2 else n2

>>> big

5

* Other methods

>>> n1, n2 = n2, n1

>>> print(n1, n2)

8 7

>>> a, b = 10, 20

>>> print((b, a)[a < b])

10

>>> print({True: a, False: b}[a < b])

10

* Nesting of inline operator

>>> a, b = 1, 2

>>> 1 if a > b else -1 if a < b else 0

-1

* Python Visualizer at csecircles.com
* In Python 2, range is a list; in Python 3, range is a range object
* Eager evaluation and lazy evaluation
* Based on needs, it uses it (generates value from range generator)
* Range object is a lazy evaluation
* Eager evaluation is implemented in functions; will not delay evaluation, along with short circuit evaluation
* Short circuit evaluation vs lazy evaluation
* In Python 2, xrange(5) gives range object, similar to range(5) in Python 3

07.09.19

* xrange() vs range()
* Both are iterable

>>> for i in range(3):

... print(i)

...

0

1

2

* Both support len function

>>> from collections import Sequence

\_\_main\_\_:1: DeprecationWarning: Using or importing the ABCs from 'collections' instead of from 'collections.abc' is deprecated, and in 3.8 it will stop working

>>> isinstance(range(10), Sequence)

True

* Equality in Python3

>>> r1 = range(10)

>>> r2 = range(10)

>>> r1 == r2

True

* is operator

>>> r1 is r2

False

* in python2

>>> r1 = xrange(10)

>>> r2 = xrange(10)

>>> r1 == r2

False

* Python 2 xrange does not support equality unless they two xranges are the exact same objects
* Slicing range/list/strings

>>> for i in range(1, 11)[2:]:

... print(i)

...

3

4

5

6

7

8

9

10

* xrange doesn’t support slicing
* containment check (in operator) works on both python2 xrange and python3 range
* Python3 supports big numbers, xrange supports smaller numbers

## Repetitive control structures – loops

* Loop index variable – usually i

>>> st = 'hello'

>>> for ch in st:

... print(ch)

...

h

e

l

l

o

* Factorial programme

>>> def fact(n):

... prod = 1

... for i in range(n, 0, -1):

... prod \*= i

... return prod

...

>>> fact(4)

24

* Factorial of negative value: recent research paper, read it
* Use sys.exit()
* Terminates programme
* Eval function

>>> a, b, c = eval(input("Enter 3 numbers: "))

Enter 3 numbers: 3, 4, 6

>>> a, b, c

(3, 4, 6)

* Evaluate based on context
* Can also use split() based on ‘ ‘
* Can use delimiter comma
* Read up on map()

11.09.19

* Two kinds of loops
* Pre test and post test loops
* Emulating post test in Python; Python only has pre test
* No do-while loop
* Boolean controlled loop or count controlled loop
* Numeric range for loops, count controlled for loop (3 expression; C++, Java etc), vectorised for loops, iterator based for loop (Python)
* Using for construct, count and iterator based can be simulated
* from coffee import python
* Double all even natural numbers from 1 to n

>>> def doub(n):

... for i in range(1, n + 1):

... if (i % 2 == 0):

... print(2\*i)

... else:

... print(i)

* Single line if else

>>> def doub(n):

... for i in range(1, n + 1):

... print(i \* 2 if i % 2 == 0 else i)

* Write a programme for bound delimited summation

>>> def bound(s, e):

... sumel = 0

... for i in range(s, e + 1):

... sumel += i

... return sumel

* Indentation in CBT very important
* Dynamic yet strongly typed
* Cannot start a variable with += operator
* DO NOT USE KEYWORD SUM

>>> def bound(s, e):

... return sum(range(s, e))

* Event controlled loops: sig module
* For event handlers
* Called special sentinel controlled values
* While loops are indefinite but not infinite (conditional based loops)
* Counter, accumulator, incrementor

03.10.19

* Dictionaries take up too much memory
* Unordered lists
* To deal with this, named tuples are used
* NamedTuple -> typing module (type hints, sub-typing, supported by this module, only python 3 and above)
* namedtuple -> collections

from collections import namedtuple

print (dir(namedtuple))

namedtuple("Student", "srn name sem")

print(a)

* a is a class template
* namedtuple is immutable
* Field names can be string (comma separated or space separated), list of strings, dictionary with values initialised to empty strings
* Look up deep copy; Vinay sir is a joke
* copy.copy and copy.copy

**04.10.19**

* Sets are hashable?Or not?Idk
* PLEASE LOOK UP
* Unordered sets for non-integers??????
* Mutable unhashable structures cannot have dicts, sets, lists as members?
* Sets don’t allow mutable objects\
* Try out all methods
* Copy – shallow copy
* Shorthand operation
* difference, difference\_update

**09.10.19**

* Inline nested for loops
* Dictionaries from lists, strings, tuples
* List of tuples

**10.10.19**

* Argument: passed to function call
* Parameter: placeholder for argument in function definition
* Types of parameters: 5 mapped to types of arguments: 4

Functions:

1. Modular
2. Abstraction
3. Reusable
4. Sharing
5. Security
6. Simplified

No function declaration; only definition

Type annotations (hints)

Dictionary of dictionaries not allowed; not hashable?

**11.10.19**

* POSIX standard – all functions must return a value (C/C++)
* In Python, return None
* Referential transparency – function call to method can be replaced with its return value
* Input does not affect the code

## Global variables

* Use global keyword inside function
* Not recommended to use global variables in any programme
* Tokenisation?
* Parsing, syntax analysis, semantic analysis
* Interpreter scans code like us??

## Sorted function

* Case-sensitive
* key (use lambda function)
* Key doesn’t perform any transformation
* Lambda t:t[0].lower()