Python

Type casting, operators, errors, scope and lifetime of variable, named constants, and exceptions

Situation (type mismatching)

The output of the following simple code segment given after it, is not what we are expecting.

```
i = 13
b = True
f = 2.5
r = (i-b)/ f
print(“(13 - True) / 2.5 is “, r)
```

```
(13 - True) / 2.5 is  4.8
```

Type Conversion and Type Casting?

Type castings

- **Implicit** type casting
  - Shorter sized data types are promoted to longer sized data types in expressions of operands with mismatched (but appropriate) data types.
  - No lose of data
- **Explicit** type casting
  - Programmer specified promotion or demotion of values for operands before being used in expression.
  - Type Casting functions: int(), str(), bool(), …
  - Data may loose, but on programmers wish

Operators

- Categories
  - Arithmetic
  - Relational
  - Identity and membership
  - Logical or Boolean
  - Bitwise
  - Assignment
  - Etc, etc

PEDMAS

- Precedence
  - Which operation will perform earlies than other in same expression

- Associativity
  - Same precedence operation will operate left to right or right to left

Common operators

```
Operators         Assoc    Description
----------------------------------------------------------Non Category
()                  --  Parentheses (grouping), function call
[]                  --  indexing, slicing
.                   --  attribute reference
------------------------------------------------------------Arithmetic
+ -                 RL  unary versions
**                  RL  power or exponent
*  /  //  %         LR  multiplication and division
+    -              LR  addition and subtraction
-----------------------------------------------------------Conditional
<    <=   >   >=    LR  inequality relational
==   !=             LR  equality relational
is,in,not is,not in LR  identity and membership
---------------------------------------------------------------Logical
not                 RL  logical NOT
and                 LR  logical AND
or                  LR  logical OR
------------------------------------------------------------Assignment
=                   RL  assignment
*=                  RL  multiplication and assignment
/=                  RL  division and assignment
%=                  RL  modulus (remainder) and assignment
+=                  RL  addition and assignment
-=                  RL  subtraction and  assignment
```

Overloaded operators

Operator are defined for types other than Numbers and Booleans

e.g.

- `str + str`
- `str * number`
- `str [not] in str`
- `list * number`
- `list + list`

Errors

- Syntax errors
  - If exist, compiler/interpreter is unable to get the statement recognized and interpret it, and you have no choice but to remove them.
- Warnings (usually in compilers)
  - May tell you about your possible logical errors, program can run with them. MUST understand them and remove them, if these WARNINGS are harmful
- Logical Errors
  - If exist, behavior of the program is unexpected, wrong results, missing values, infinite execution
- Runtime errors / Exceptions
  - Occurs when data is not appropriate for an operation, e.g. divide by ZERO, crossing arrays limits
- Exceptions
  - Raised by the code through special instructions

Variables – Scope and Lifetime

- Global variables
  - Avoid them (almost) in all cases
- Automatic or Local variables
  - Functions parameters
  - Defined with a block
- Scope
  - Accessible within the block, they are defined
- Lifetime
  - Created when first time assigned a value
  - Exists till execution exits form their scope's blocks

Named constants

Just Variables in Python

- Name
  - Usually CAPITALIZED/Global
- Helps in avoiding magic numbers

- Examples
  - PI = 3.14159265358979323846
  - DATASIZE = 100
  - KM_PER_MILE = 1.60934
  - BEEP = “\a”

Situation (exceptional cases)

Function that divide first parameter with second

```
def divide(num, den):
    if (den == 0):
        // what code should be here
    return num/den
```

Solutions?

- If through coding, exceptional case can be handled, only then they should be dealt.
- Printing messages, exiting, returning special values may lead other problems

Exceptional case

- Handling not possible
  - If handing of exceptional case is not locally possible, then it is better to raise as Exception, that may lead the caller function to catch it and handle if possible.
  - raise Exception(str_expr)

```
def divide(num, den):
    if (den == 0):
        raise Exception(“Den 0”)
    return num/den
```

try, except, else and finally

```
try:
    print(divide(45, 5))  # (4, 0)
except Exception as e:
    print(“in except, div zero”, e)
else:
    print(“in else, div zero”)
finally:
    print(“in finally, div zero”)
```

https://www.tutorialspoint.com/python-exception-base-classes