Introduction to Python

Operations and Variables

Topics

1) Arithmetic Operations
2) Floor Division vs True Division
3) Modulo Operator
4) Operator Precedence
5) String Concatenation
6) Augmented Assignment

Arithmetic Operations

Operator	Name	Description
`a + b`	Addition	Sum of a and b
`a - b`	Subtraction	Difference of a and b
`a * b`	Multiplication	Product of a and b
`a / b`	True division	Quotient of a and b
`a // b`	Floor division	Quotient of a and b, removing fractional parts
`a % b`	Modulus	Remainder after division of a by b
`a ** b`	Exponentiation	a raised to the power of b
`-a`	Negation	The negative of a
`+a`	Unary plus	a unchanged (rarely used)

Mixing Types

Any expression that two floats produce a float.

```
x = 17.0 - 10.0
print(x)          # 7.0
```

When an expression's operands are an int and a float, Python automatically converts the int to a float.

```
x = 17.0 - 10
print(x)          # 7.0
y = 17 - 10.0
print(y)          # 7.0
```

True Division vs Floor Division

The operator / is true division and the operator // returns floor division(round down after true divide). True divide / always gives the answer as a float.

```
print(23 // 7)      # 3
print(3 // 9)       # 0
print(-4 // 3)      # -2
print(6 / 5)        # 1.2
print(6 / 3)        # 2.0 NOT 2!
```

Remainder with %

The % operator computes the remainder after floor division.

- `14 % 4` is `2`
- `218 % 5` is `3`

```
    3
4 ) 14
   12
    2
```

```
   43
5 ) 218
   20
    18
    15
     3
```

Applications of % operator:

- Obtain last digit of a number: `230857 % 10` is `7`
- Obtain last 4 digits: `658236489 % 10000` is `6489`
- See whether a number is odd: `7 % 2` is `1`, `42 % 2` is `0`

Modulo Operator

The operator % returns the remainder after floor division.

```
print(18 % 5)          # 3
print(2 % 9)           # 2, if first number is smaller, it's the answer
print(125 % 10)        # 5
print(0 % 10)          # 0
print(10 % 0)          # ZeroDivisionError
```

Why floor and modulo division are useful

Floor division allows us to extract the integer part of the division while the modulo operator extracts the remainder part of the division. Consider the question:

How many weeks and days are there in 25 days?

Answer: 3 weeks plus 4 days.

```
num_weeks = 25 // 7  # extracting number of weeks
print(num_weeks)     # 3

num_days = 25 % 7    # extracting number of days
print(num_days)      # 4
```

Why the modulo operator is useful

If today is a Tuesday, which day is 43 days from today?

Answer: 43 divided by 7 is 6 with a remainder of 1. Thus, it will be Wednesday.

```
print(43 % 7)   # 1
```

Even/odd: A number x is even if x % 2 is 0 and odd if x % 2 is 1.

```
num = int(input('Please enter an integer value: '))
print(num, 'is even:', num % 2 == 0)
```

Output 1:
Please enter an integer value: 4
4 is even: True

Output 2:
Please enter an integer value: 13
13 is even: False

Expressions

Find the exact change for 137 cents using quarters, dimes, nickels and cents. Use the least number of coins.

How many quarters? 137 // 25 = 5 quarters (Floor Division!)

What's leftover? 137 % 25 = 12 cents

How many dimes? 12 // 10 = 1 dime

What's leftover? 12 % 10 = 2 cents

How many nickels? 2 // 5 = 0 nickels.

What's leftover? 2 % 5 = 2 cents.

How many pennies? 2 // 1 = 2 pennies

What's leftover? 2 % 1= 0 cents. Done!

Extracting Digits

Given a three-digit integer. Extract its the ones, tens and hundreds digits.

For example, if the integer is 352. Its ones digit is the 2, its tens digit is the 5 and its hundreds digit is the 3.

```
number = 352
print("ones:", number % 10)          # ones: 2
number = number // 10                # number = 35 (discards last digit)
print("tens:", number % 10)          # tens: 5
number = number // 10                # number = 3 (discards last digit)
print("hundreds:", number % 10)      # hundreds:3
```

Note: Modulo 10 (% 10) extracts the last digit. Floor division (// 10) discards the last digit. Later in another lecture, we will see how to generalize this to any number of digits.

Extracting Digits

Alternatively:

```
number = 352
ones = number % 10
tens = (number // 10) % 10
hundreds = number // 100
print("ones:", ones, "tens", tens, "hundreds:", hundreds)
```

Output:

ones: 2 tens: 5 hundreds: 3

Exponentiation and Negation

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

Negation is a **unary operator**. It applies to only one operand. Other operations such as +, -, *, /, //, % are **binary operators**, they apply to two operands.

```
x = -5
y = --5
print(x)    # -5
print(y)    # 5
```

Operator Precedence

Precedence	Operator	Operation
highest	**	exponentiation
	-	negation
	*, /, //, %	multiplication, division, floor division, modulus (left to right)
lowest	+, -	adding, subtraction(left to right)

Operators on the same row are applied left to right. Exponentiation, however, is applied right to left. Expressions in parenthesis are evaluated first(PEMDAS).

Operator Precedence

```
x = -2 ** 4
print(x)  # -16

y = 7 - 4 * 5 % (1 + 2)
print(y)  # 5
```

```
7 - 4 * 5 % (1 + 2)
  7 - 4 * 5 % 3
    7 - 20 % 3
      7 - 2
        5
```

Augmented Assignment

An **augmented assignment** combines an assignment statement with an operator to make the statement more concise.

Shorthand	Equivalent version
variable += **value**	**variable** = **variable** + **value**
variable -= **value**	**variable** = **variable** - **value**
variable *= **value**	**variable** = **variable** * **value**
variable /= **value**	**variable** = **variable** / **value**
variable %= **value**	**variable** = **variable** % **value**

```
x = 4
x += 1      # equivalent to x = x + 1
print(x)    # 5
```

Augmented Assignment

```
x = 3
x *= 2 + 5
print(x)              # 21

number = 5
number *= number
print(number)         # 25
```

String Concatenation

Two strings can be combined, or **concatenated**, using the + operator:

```
string1 = "abra"
string2 = "cadabra"
magic_string = string1 + string2
```

```
first = "Michael"
last = "Smith"
full_name = first + " " + last
```

Concatenating a string and a number raises a TypeError. Must first cast the number into a string using str().

```
apples = "I have " + 3 + "apples"          # error!
apples = "I have " + str(3) + "apples"     # correct!
```

Errors

Beginning programmers make mistakes writing programs because of inexperience in programming in general or due to unfamiliarity with a programming language.

There are four general kinds of errors: **syntax errors, run-time errors, overflow errors** and **logic errors.**

The interpreter reads the Python source file and translates it into a executable form. This is the *translation phase*. If the interpreter detects an invalid program statement during the translation phase, it will terminate the program's execution and report an error.

Such errors result from the programmer's misuse of the language. A **syntax error** is a common error that the interpreter can detect when attempting to translate a Python statement into machine language.

Syntax Errors

A **syntax error** is a common error that the interpreter can detect when attempting to translate a Python statement into machine language.

```
x = 3
y = x + 5
y + 2 = x              # SyntaxError
print(x                # SyntaxError
z = "hello             # SyntaxError
```

Run-time Errors

A syntactically correct Python program still can have problems. Some language errors depend on the context of the program's execution. Such errors are called **run-time exceptions** or **run-time errors**.

We say the interpreter **raises** an exception. Run-time exceptions arise after the interpreter's translation phase and during the program's execution phase.

Run-time errors will cause the program to immediately terminate.

```
x = 3
y = x + 5
y = w + 1        # Syntactically correct but raises a run-time
                 # error: "name w is not defined"
```

Run-time Errors

Here's another example of a run-time error.

```
# Get two integers from the user
print('Please enter two numbers to divide.')
dividend = int(input('Please enter the dividend: '))
divisor = int(input('Please enter the divisor: '))
# Divide them and report the result
print(dividend, '/', divisor, "=", dividend/divisor)
```

The expression dividend/divisor is potentially dangerous. If the user enters, for example, 32 and 4, the program works nicely.

If the user instead types the numbers 32 and 0, the program reports an error and terminates. Division by zero is undefined in mathematics, and division by zero in Python is illegal.

Run-time Errors

Another example of a run-time error is an **overflow error.** An **overflow error** is an error that occurs when a computer attempts to handle a number that is outside of the defined range of values.

```
print(2.0 ** 10000)  # OverflowError: 'Result too large'
```

Logic Errors

The interpreter can detect syntax errors during the translation phase and uncover run-time exceptions during the execution phase. Both kinds of problems represent violations of the Python language.

Such errors are the easiest to repair because the interpreter indicates the exact location within the source code where it detected the problem.

A **logic error** is a mistake in the algorithm or program that causes it to behave incorrectly or unexpectedly. A logic error is subtle and will not cause the program to terminate.

Logic Errors

Consider the effects of replacing the expression dividend/divisor with divisor/dividend.

The program runs, and unless the user enters a value of zero for the dividend, the interpreter will report no errors. However, the answer it computes is not correct in general.

The only time the program will print the correct answer is when dividend equals divisor.

The program contains an error, but the interpreter is unable detect the problem. An error of this type is known as a **logic error**.

AP Exam Info

Assignment, Display, and Input	
Text: `a ← expression` Block: 	Evaluates `expression` and then assigns a copy of the result to the variable `a`.
Text: `DISPLAY(expression)` Block: 	Displays the value of `expression`, followed by a space.

Text:
`INPUT()`
Block:
`INPUT`
	Accepts a value from the user and returns the input value.

AP Exam Info

Arithmetic Operators and Numeric Procedures	

Text and Block:
`a + b`
`a - b`
`a * b`
`a / b`
	The arithmetic operators `+`, `-`, `*`, and `/` are used to perform arithmetic on `a` and `b`.
	For example, `17 / 5` evaluates to `3.4`.
	The order of operations used in mathematics applies when evaluating expressions.

Text and Block: `a MOD b`	Evaluates to the remainder when `a` is divided by `b`. Assume that `a` is an integer greater than or equal to `0` and `b` is an integer greater than `0`. For example, `17 MOD 5` evaluates to `2`. The `MOD` operator has the same precedence as the `*` and `/` operators.

Note that there is no floor division on the AP exam.
On the AP, like Python, the operator / is true division.

Lab 1: Modulo Operator

Create a new repl on repl.it. Write code to match the following console output. Underline numbers are user inputs; you must use the input() function.

Create the following variables: ones, tens and hundreds

Enter a three-digit number: 245

The sum of the digits of 245 is 11.

Create the following variables: quarters, dimes, nickels and pennies.

Enter amount in cents: 137

Number of quarters: 5

Number of dimes: 1

Number of nickels: 0

Number of pennies: 2

References

1) Vanderplas, Jake, A Whirlwind Tour of Python, O'Reilly Media.
This book is completely free and can be downloaded online at O'reilly's site.