2. Types

Based on Practical Python Programming

Numbers

Types of Numbers

Python has 4 types of numbers:

- Booleans
- Integers
- Floating point
- Complex (imaginary numbers)

Booleans (bool)

Booleans have two values: `True`, `False`.

```
a = True
b = False
```

Numerically, they're evaluated as integers with value `1`, `0`.

```
c = 4 + True # 5
d = False
if d == 0:
    print('d is False')
```

Integers (int)

Signed values of arbitrary size and base:

```
a = 37
b = -2993929937277166273771284818122412 31
c = 0x7fa8      # Hexadecimal
d = 0o253       # Octal
e = 0b10001111  # Binary
```

Common operations:

```
x + y      # Add
x - y      # Subtract
x * y      # Multiply
x / y      # Divide (produces a float)
```

```
x // y      # Floor Divide (produces an integer)
x % y       # Modulo (remainder)
x ** y      # Power
x << n      # Bit shift left
x >> n      # Bit shift right
x & y       # Bit-wise AND
x | y       # Bit-wise OR
x ^ y       # Bit-wise XOR
~x          # Bit-wise NOT
abs(x)      # Absolute value
```

Floating point (float)

Use a decimal or exponential notation to specify a floating point value:

```
a = 37.45
b = 4e5 # 4 x 10**5 or 400,000
c = -1.345e-10
```

Floats are represented as double precision using the native CPU representation IEEE 754. This is the same as the `double` type in the programming language C.

> 17 digits of precision
> Exponent from -308 to 308

Be aware that floating point numbers are inexact when representing decimals.

```
>>> a = 2.1 + 4.2
>>> a == 6.3
False
>>> a
6.300000000000001
>>>
```

This is **not a Python issue**, but the underlying floating point hardware on the CPU.

Common Operations:

```
x + y      # Add
x - y      # Subtract
x * y      # Multiply
x / y      # Divide
x // y     # Floor Divide
x % y      # Modulo
x ** y     # Power
abs(x)     # Absolute Value
```

These are the same operators as Integers, except for the bit-wise operators. Additional math functions are found in the `math` module.

```
import math
a = math.sqrt(x)
b = math.sin(x)
c = math.cos(x)
d = math.tan(x)
e = math.log(x)
```

Comparisons

The following comparison / relational operators work with numbers:

```
x < y      # Less than
x <= y     # Less than or equal
x > y      # Greater than
x >= y     # Greater than or equal
x == y     # Equal to
x != y     # Not equal to
```

You can form more complex boolean expressions using

`and`, `or`, `not`

Here are a few examples:

```
if b >= a and b <= c:
    print('b is between a and c')

if not (b < a or b > c):
    print('b is still between a and c')
```

Converting Numbers

The type name can be used to convert values:

```
a = int(x)    # Convert x to integer
b = float(x)  # Convert x to float
```

Try it out.

```
>>> a = 3.14159
>>> int(a)
3
>>> b = '3.14159' # It also works with strings containing numbers
>>> float(b)
3.14159
>>>
```

Strings

This section introduces ways to work with text.

Representing Literal Text

String literals are written in programs with quotes.

```
# Single quote
a = 'Yeah but no but yeah but...'

# Double quote
b = "computer says no"

# Triple quotes
c = '''
Look into my eyes, look into my eyes, the eyes, the eyes, the eyes,
not around the eyes,
don't look around the eyes,
look into my eyes, you're under.
'''
```

Normally strings may only span a single line. Triple quotes capture all text enclosed across multiple lines including all formatting.

There is no difference between using single (‘) versus double (“) quotes. *However, the same type of quote used to start a string must be used to terminate it*.

String escape codes

Escape codes are used to represent control characters and characters that can’t be easily typed directly at the keyboard. Here are some common escape codes:

```
'\n'      # Line feed
'\r'      # Carriage return
'\t'      # Tab
'\''      # Literal single quote
'\"'      # Literal double quote
'\\'      # Literal backslash
```

String Indexing

Strings work like an array for accessing individual characters. You use an integer index, starting at 0. Negative indices specify a position relative to the end of the string.

```
a = 'Hello world'
b = a[0]          # 'H'
```

```
c = a[4]          # 'o'
d = a[-1]         # 'd' (end of string)
```

You can also slice or select substrings specifying a range of indices with `:`.

```
d = a[:5]     # 'Hello'
e = a[6:]     # 'world'
f = a[3:8]    # 'lo wo'
g = a[-5:]    # 'world'
```

The character at the ending index is not included. Missing indices assume the beginning or ending of the string.

String operations

Concatenation, length, membership and replication.

```
# Concatenation (+)
a = 'Hello' + 'World'   # 'HelloWorld'
b = 'Say ' + a          # 'Say HelloWorld'

# Length (len)
s = 'Hello'
len(s)                  # 5

# Membership test (`in`, `not in`)
t = 'e' in s            # True
f = 'x' in s            # False
g = 'hi' not in s       # True

# Replication (s * n)
rep = s * 5             # 'HelloHelloHelloHelloHello'
```

String methods

Strings have methods that perform various operations with the string data.

Example: stripping any leading / trailing white space.

```
s = '  Hello '
t = s.strip()     # 'Hello'
```

Example: Case conversion.

```
s = 'Hello'
l = s.lower()     # 'hello'
u = s.upper()     # 'HELLO'
```

Example: Replacing text.

```
s = 'Hello world'
t = s.replace('Hello' , 'Hallo')   # 'Hallo world'
```

More string methods:

Strings have a wide variety of other methods for testing and manipulating the text data. This is a small sample of methods:

```
s.endswith(suffix)     # Check if string ends with suffix
s.find(t)              # First occurrence of t in s
s.index(t)             # First occurrence of t in s
s.isalpha()            # Check if characters are alphabetic
s.isdigit()            # Check if characters are numeric
s.islower()            # Check if characters are lower-case
s.isupper()            # Check if characters are upper-case
s.join(slist)          # Join a list of strings using s as delimiter
s.lower()              # Convert to lower case
s.replace(old,new)     # Replace text
s.rfind(t)             # Search for t from end of string
s.rindex(t)            # Search for t from end of string
s.split([delim])       # Split string into list of substrings
s.startswith(prefix)   # Check if string starts with prefix
s.strip()              # Strip leading/trailing space
s.upper()              # Convert to upper case
```

String Mutability

Strings are “immutable” or read-only. Once created, the value can’t be changed.

```
>>> s = 'Hello World'
>>> s[1] = 'a'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
```

All operations and methods that manipulate string data, always create new strings.

String Conversions

Use `str()` to convert any value to a string. The result is a string holding the same text that would have been produced by the `print()` statement.

```
>>> x = 42
>>> str(x)
'42'
>>>
```

f-Strings

A string with formatted expression substitution.

```
>>> name = 'IBM'
>>> shares = 100
>>> price = 91.1
>>> a = f'{name:>10s} {shares:10d} {price:10.2f}'
>>> a
'       IBM        100      91.10'
>>> b = f'Cost = ${shares*price:0.2f}'
>>> b
'Cost = $9110.00'
>>>
```

Note: This requires Python 3.6 or newer. The meaning of the format codes is covered later.

None type

```
email_address = None
```

`None` is often used as a placeholder for optional or missing value. It evaluates as `False` in conditionals.

```
if email_address:
    send_email(email_address, msg)
```

Exercises

Exercise 2.1: Dave's mortgage

> Dave has decided to take out a 30-year fixed rate mortgage of $500,000 with Guido's Mortgage, Stock Investment, and Bitcoin trading corporation. The interest rate is 5% per year and the monthly payment is $2684.11.

Here is a program that calculates the total amount that Dave will have to pay over the life of the mortgage:

```
# mortgage.py

principal = 500000.0
rate = 0.05
payment = 2684.11
total_paid = 0.0
MONTHS_IN_YEAR=12

while principal > 0:
    principal = principal * (1 + rate / MONTHS_IN_YEAR) - payment
    total_paid = total_paid + payment

print('Total paid', total_paid)
```

Enter this program and run it. You should get an answer of `966,279.6`.

Exercise 2.2: Extra payments

Suppose Dave pays an extra $1000/month for the first 12 months of the mortgage?

Modify the program to incorporate this extra payment and have it print the total amount paid along with the number of months required.

When you run the new program, it should report a total payment of `929,965.62` over 342 months.

Exercise 2.3: Making an Extra Payment Calculator

Modify the program so that extra payment information can be more generally handled. Make it so that the user can set these variables:

```
extra_payment_start_month = 61
extra_payment_end_month = 108
extra_payment = 1000
```

Make the program look at these variables and calculate the total paid appropriately.

How much will Dave pay if he pays an extra $1000/month for 4 years starting after the first five years have already been paid?

Solution : 883442.32

Exercise 2.4: Making a table

Modify the program to print out a table showing the month, total paid so far, and the remaining principal. The output should look something like this:

```
1 2684.11 499399.22
2 5368.22 498795.94
3 8052.33 498190.15
4 10736.44 497581.83
5 13420.55 496970.98
...
308 874705.88 3478.83
309 877389.99 809.21
310 880074.1 -1871.53
Total paid 880074.1
Months 310
```

Exercise 2.5: Bonus

While you're at it, fix the program to correct for the overpayment that occurs in the last month.

Exercise 2.6: A Mystery

`int()` and `float()` can be used to convert numbers. For example,

```
>>> int("123")
123
>>> float("1.23")
1.23
>>>
```

With that in mind, can you explain this behavior?

```
>>> bool("False")
True
>>>
```

Exercise 2.7: f-strings

Sometimes you want to create a string and embed the values of variables into it.

To do that, use an f-string. For example:

```
>>> name = 'IBM'
>>> shares = 100
>>> price = 91.1
>>> f'{shares} shares of {name} at ${price:0.2f}'
'100 shares of IBM at $91.10'
>>>
```

Modify the `mortgage.py` program from Exercise 2.5 to create its output using f-strings. Try to make it so that output is nicely aligned.