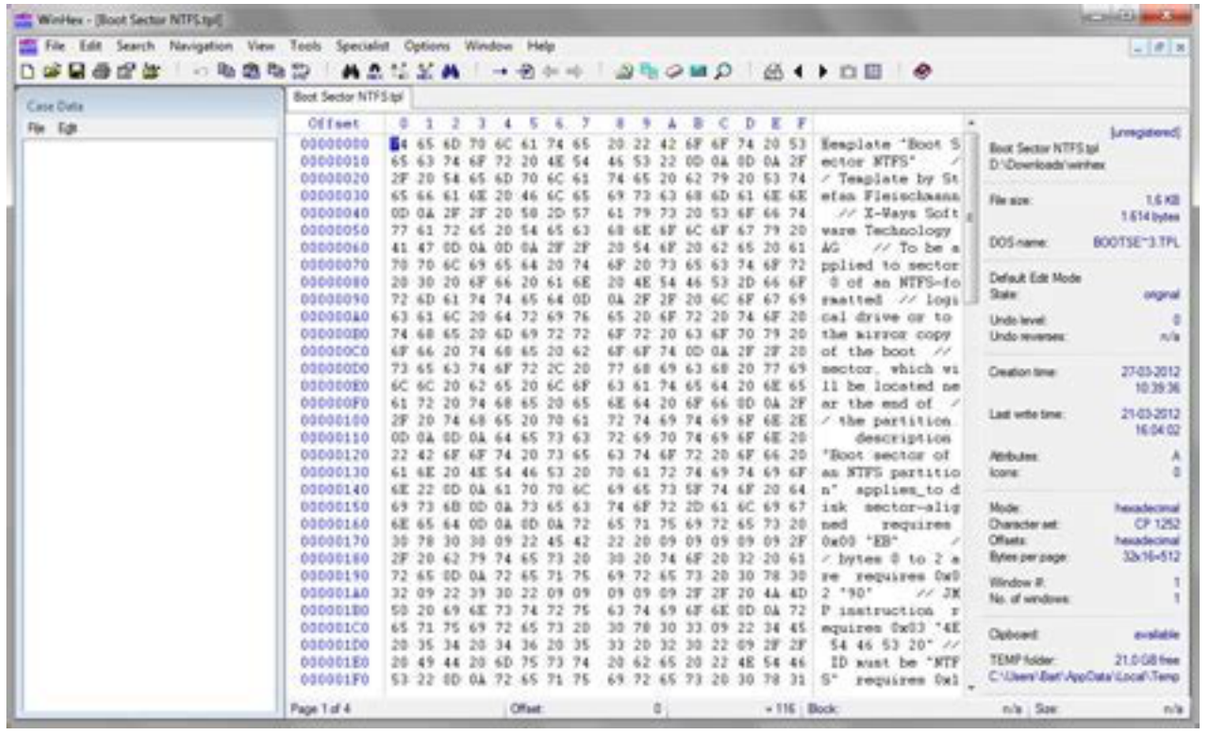
Encodings

Thomas Schwarz, SJ

Encodings

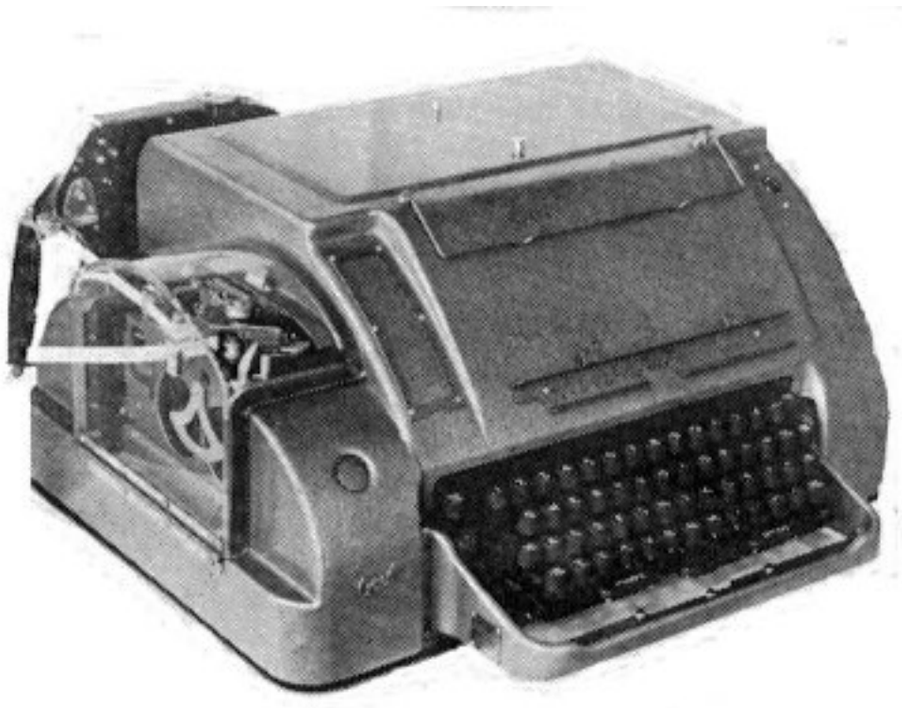
- Information technology has developed a large number of ways of storing particular data
  - Here is some background



Using a forensics tool (Winhex) in order to reveal the bytes actually stored

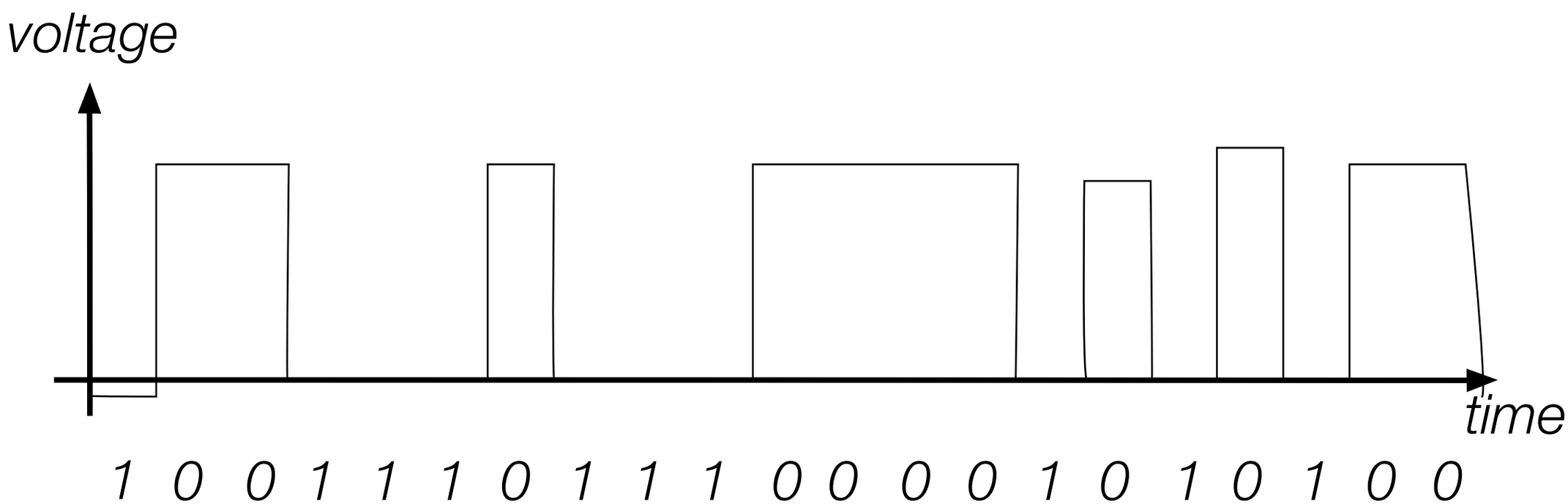
Encodings

- Teleprinters
    - Used to send printed messages
        - Can be done through a single line
        - Use timing to synchronize up and down values



Encodings

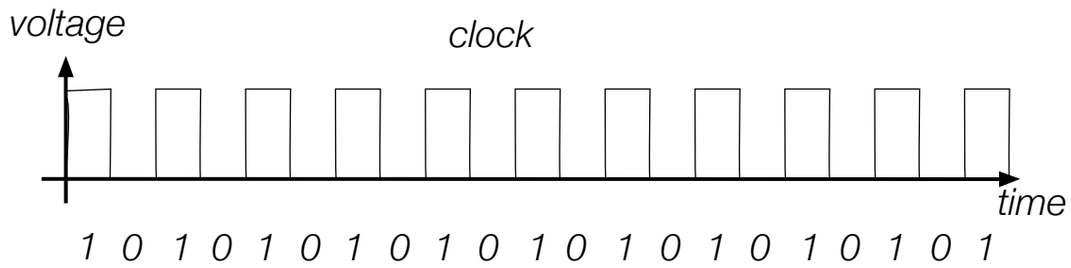
- Serial connection:
  - Voltage level during an interval indicates a bit
  - Digital means that changes in voltage level can be tolerated without information loss

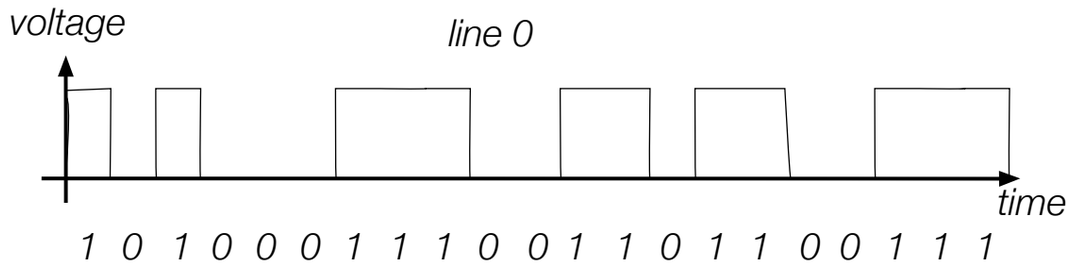


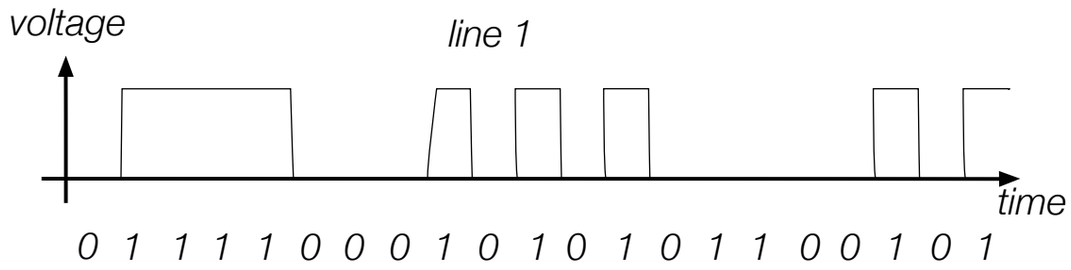
Encodings

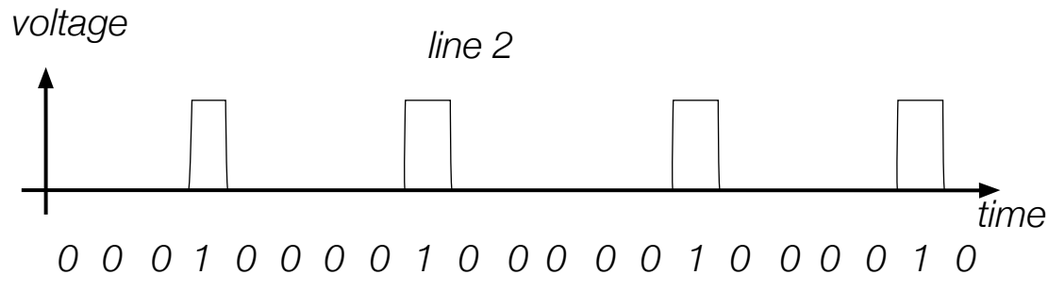
- Parallel Connection
  - Can send more than one bit at a time
  - Sometimes, one line sends a timing signal

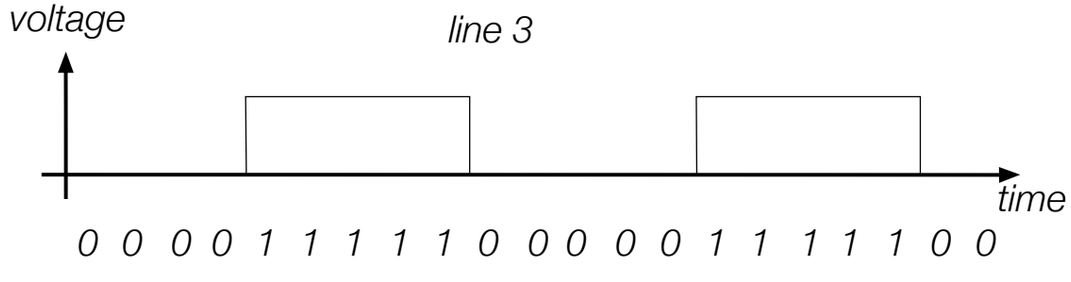
- Sending
  - 1000
  - 0100
  - 1100
  - 0100
  - …
- Small errors in timing and voltage are repaired automatically











Encodings

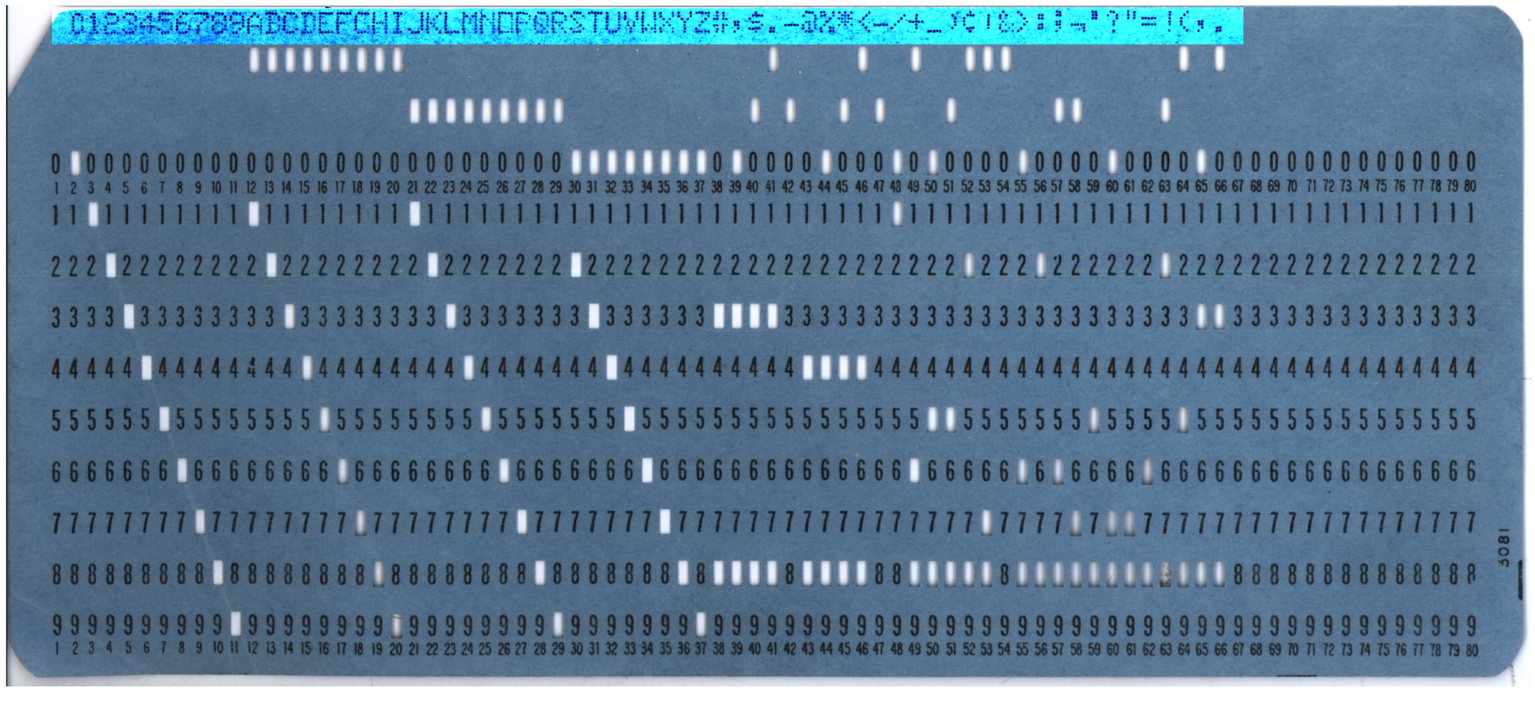
- Need a code to transmit letters and control signals
- Émile Baudot's code 1870
  - 5 bit code
  - Machine had 5 keys, two for the left and three for the right hand
  - Encodes capital letters plus NULL and DEL
  - Operators had to keep a rhythm to be understood on the other side

Encodings

- Many successors to Baudot's code
  - Murray's code (1901) for keyboard
    - Introduced control characters such as Carriage Return (CR) and Line Feed (LF)
  - Used by Western Union until 1950

Encodings

- Computers and punch cards
  - Needed an encoding for strings
    - EBCDIC — 1963 for punch cards by IBM
    - 8b code



Encodings

- ASCII — American Standard Code for Information Interchange — 1963
  - 8b code
    - Developed by American Standard Association, which became American National Standards Institute (ANSI)
    - 32 control characters
    - 91 alphanumerical and symbol characters
    - Used only 7b to encode them to allow local variants
- Extended ASCII
  - Uses full 8b
    - Chooses letters for Western languages

Encodings

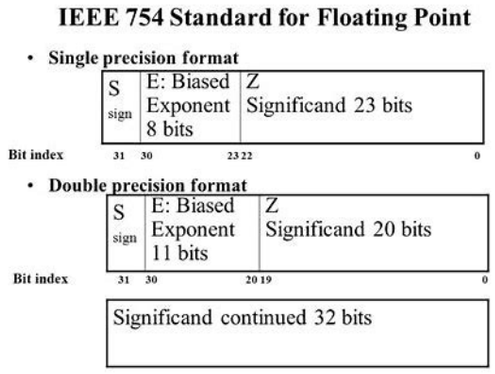
- Unicode - 1991
  - “Universal code” capable of implementing text in all relevant languages
  - 32b-code
  - For compression, uses “language planes”

Encodings

- UTF-7 — 1998
  - 7b-code
    - Invented to send email more efficiently
    - Compatible with basic ASCII
    - Not used because of awkwardness in translating 7b pieces in 8b computer architecture

Encodings

- UTF-8 — Unicode
  - Code that uses
    - 8b for the first 128 characters (basically ASCII)
    - 16b for the next 1920 characters
      - Latin alphabets, Cyrillic, Coptic, Armenian, Hebrew, Arabic, Syriac, Thaana, N’Ko
    - 24b for
      - Chinese, Japanese, Koreans
    - 32b for
      - Everything else



Encodings

- Numbers
  - There is a variety of ways of storing numbers (integers)
    - All based on the binary format
  - For floating point numbers, the exact format has a large influence on the accuracy of calculations
    - All computers use the IEEE standard

Python and Encodings

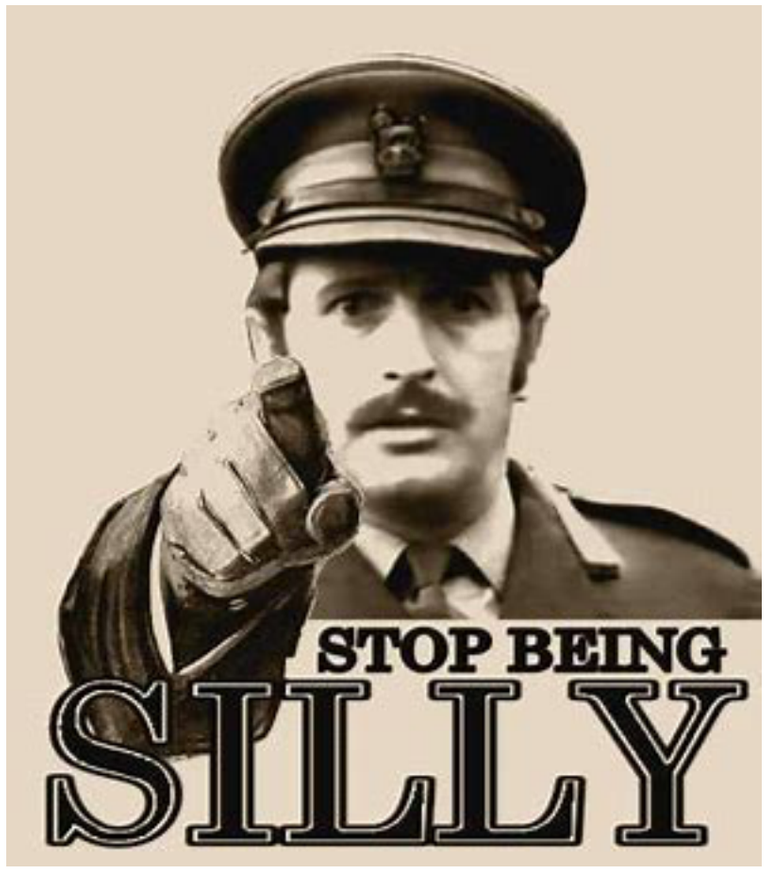
- Python “understands” several hundred encodings
  - Most important
    - ascii (corresponds to the 7-bit ASCII standard)
    - **utf-8** (usually your best bet for data from the Web)
    - latin-1
      - straight-forward interpretation of the 8-bit extended ASCII
      - never throws a “cannot decode” error
      - no guarantee that it read things the right way

Problems with Line Endings

- ASCII code was developed when computers wrote to teleprinters.
  - A new line consisted of a carriage return followed or preceded by a line-feed.
  - UNIX and windows choose to different encodings
    - Unix has just the newline character “\n”
    - Windows has the carriage return: “\r\n”
- By default, Python operates in “universal newline mode”
  - All common newline combinations are understood
  - Python writes new lines just with a “\n”

Python

String Formatting



Formatting Strings

- We really need to learn how to format strings
  - Python has made several attempts before settling on an efficient syntax.
  - You can find information on the previous solutions on the net.
- Use the `format` function
  - Distinguish between the **blueprint**
  - and the **string to be formatted**
  - Result is the formatted string.

Formatting Strings

- Blueprint string
  - Uses { } to denote places for variables
  - Simple example
    - `"{} {}".format('one', 'two')`

Blueprint

Calling format

String to be formatted

- Result `'one two'`

Formatting Strings

- Inside the brackets, we can put indices to select variables
  - 0 means first variable, 1 second, …
  - Can reuse variables

```
>>> "{0}, {0}, {1}, just {0}".format("great", "extraordinary")
'great, great, extraordinary, just great'
```

Formatting Strings

- Additional formatting inside the bracket after a colon
- Can assign the number of characters to print out

```
>>> "{0:10}, {1:10}, {0:10}".format("funny", "nuts")
'funny     , nuts      , funny     '
```

- Default alignment is to the left

Formatting Strings

- Use ^ to center
- Use < to left-align
- Use > to right-align

```
>>> "{0:10}|{1:^10}|{0:>10}".format("sheep", "wolf")
'sheep     |   wolf   |     sheep'
```

Formatting Strings

- Numbers are handled without specifying format instructions.

```
>>> "{} divided by {} is {} modulo {}".format(143, 29, 143//29, 143%29)
'143 divided by 29 is 4 modulo 27'
```

- Or we can insist on special types
  - Use s for string
  - Use d for decimal
  - Use f for floating point
  - Use e for floating point in exponential notation

Formatting Strings

- By specifying “f” we ask for floating point format
- By specifying “e” we ask for scientific format

```
>>> "{0:f}, {0:e}".format(3.141)
'3.141000, 3.141000e+00'
```

Formatting Strings

- Padding
  - If the variable needs more space to print out, it will be provided automatically

```
>>> "{:10s}".format("Pneumonoultramicroscopicsilicovolcanoconiosis")
'Pneumonoultramicroscopicsilicovolcanoconiosis'
```

  - This is actually the longest officially recognized word in English

Formatting Strings

- Padding:
  - On the reverse, we can give the number of significant digits after a period

```
>>> "{:8.2f}".format(3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651328230664709384460955058223172535940812848 1)
'    3.14'
```

  - We only want to keep two decimal digits after the period
  - But use a total of 8 spaces for the number.

Formatting Strings

- Escaping curly brackets:
  - If we want to write strings with format containing the curly brackets “{“ and “}”, we just have to write “{{“ and “}}”

```
>>> "{{ {}, {} }}".format(3, 4)
'{ 3, 4 }'
```

  - A single bracket is a placeholder, a double curly bracket is a single one in the resulting string.

Application: Pretty Printing

- Develop a mortgage payment plan
  - Accountants have formulae for that, but it is fun to do it directly
  - Assume you take out a loan of L$ dollars
    - The loan is financed at a rate of r% annually
    - Interest is paid monthly, i.e. at a rate of r/12%
- Each month you make a repayment
  - Part of the repayment is to pay the interest
  - The remainder pays down the debt

Mortgage Payments

- Use a while-loop
  - Condition is that there is still an outstanding debt
  - Adjust outstanding debt
  - Count the number of payments
- Need to initialize values

Mortgage Payments

- We need values for:
  - Monthly Rate (interest in percent)/1200
  - Principal
  - Repayment
- Get those from the user
  - A true application would contain code that checks whether these numbers make sense.

Mortgage Payments

- Initialization

```
princ = float(input("What is the prinipal “))
rate = float(input("What is the interest rate (in percents)? "))/1200
print("Your minimum rate is ", rate*princ)
paym = float(input("What is the monthly payment? “))
month = 0
```

Mortgage Payments

- We continue until we paid down the principal to zero

```
while princ > 0:
```

Mortgage Payments

- Update the situation in the while loop
- Last payment does not need to be full, so we calculate it

```
intpaid = princ*rate
princ = princ + princ*rate - paym
if princ < 0:
        lastpayment = paym + princ
        princ = 0
month += 1
```

```
**********************************
**  The Ultimate Mortgage Calculator  **
**********************************
What is the prinipal 40000
What is the interest rate (in percents)? 4
Your minimum rate is 133.33
What is the monthly payment? 1950
This is what your mortgage scheme looks like
Month   Interest        Principal
-------------------------------
  1       133.33         38183.33
  2       127.28         36360.61
  3       121.20         34531.81
  4       115.11         32696.92
  5       108.99         30855.91
  6       102.85         29008.76
  7        96.70         27155.46
  8        90.52         25295.98
  9        84.32         23430.30
 10        78.10         21558.40
 11        71.86         19680.26
 12        65.60         17795.86
 13        59.32         15905.18
 14        53.02         14008.20
 15        46.69         12104.89
 16        40.35         10195.24
 17        33.98          8279.22
 18        27.60          6356.82
 19        21.19          4428.01
 20        14.76          2492.77
 21         8.31           551.08
 22         1.84             0.00
You paid of the loan in 22 months, and your last payment was 552.92
>>>
```

Pretty-Printing Tables

- Format Strings revisited:
  - Format string — blueprint
  - Uses { } to denote spots where variables get inserted

Pretty-Printing Tables

- Syntax
  - {a:^10.3f}
    - a — the number of the variable
      - Can be left out
    - : — what follows is the formatting instruction
    - 10 — number of spaces for the variable
    - . — what follows is the precision
    - 3 — precision
    - f — print in floating point format

Pretty-Printing Tables

- If the variable is larger than the space given:
  - Full value is printed out
  - Alignment by default is
    - left (<) for strings
    - right (>) for numbers

Pretty-Printing Tables

- Task:
  - A program that gives a table for the log and the exponential function between 1 and 10
  - Hint: `x=1+i/10`

```
  x    |     exp(x)     |     log(x)
-------------------------------------
 1.00  |        2.71828 |        0.00000
 1.10  |        3.00417 |        0.09531
 1.20  |        3.32012 |        0.18232
 1.30  |        3.66930 |        0.26236
 1.40  |        4.05520 |        0.33647
 1.50  |        4.48169 |        0.40547
 1.60  |        4.95303 |        0.47000
 1.70  |        5.47395 |        0.53063
```