Python in 3 parts

A pandemic-adapted professional development workshop

Mark Galassi

Space Science and Applications group
Los Alamos National Laboratory

Part I – 2020-06-23; part II – 2020-06-24; part III – 2020-06-25

Last built 2020-06-24T22:35:48

Outline

Outline

Goals and path

In the educational industrial complex we are required to state our goals before we start.
It might even be a good idea.

Goals

Goals and path

In the educational industrial complex we are required to state our goals before we start.
It might even be a good idea.

Goals

- Practical hands-on work in Python.

Goals and path

In the educational industrial complex we are required to state our goals before we start.
It might even be a good idea.

Goals

- Practical hands-on work in Python.
- A deep awareness of how programming and Python fit in what we do.

Goals and path

In the educational industrial complex we are required to state our goals before we start.
It might even be a good idea.

Goals

- Practical hands-on work in Python.
- A deep awareness of how programming and Python fit in what we do.

The path

Goals and path

In the educational industrial complex we are required to state our goals before we start.
It might even be a good idea.

Goals

- Practical hands-on work in Python.
- A deep awareness of how programming and Python fit in what we do.

The path

- The "K&R" approach.

Goals and path

In the educational industrial complex we are required to state our goals before we start.
It might even be a good idea.

Goals

- Practical hands-on work in Python.
- A deep awareness of how programming and Python fit in what we do.

The path

- The "K&R" approach.
- Tutorial and examples followed by insights.

Goals and path

In the educational industrial complex we are required to state our goals before we start.
It might even be a good idea.

Goals

- Practical hands-on work in Python.
- A deep awareness of how programming and Python fit in what we do.

The path

- The "K&R" approach.
- Tutorial and examples followed by insights.

Style

Goals and path

In the educational industrial complex we are required to state our goals before we start.
It might even be a good idea.

Goals

- Practical hands-on work in Python.
- A deep awareness of how programming and Python fit in what we do.

The path

- The "K&R" approach.
- Tutorial and examples followed by insights.

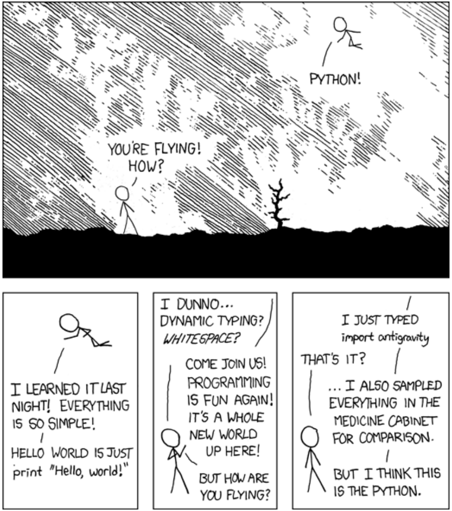
Style

- Slides are placeholders for work in an editor.
- We will have a URL for monitoring my editor.

(dude, we're not programming yet)

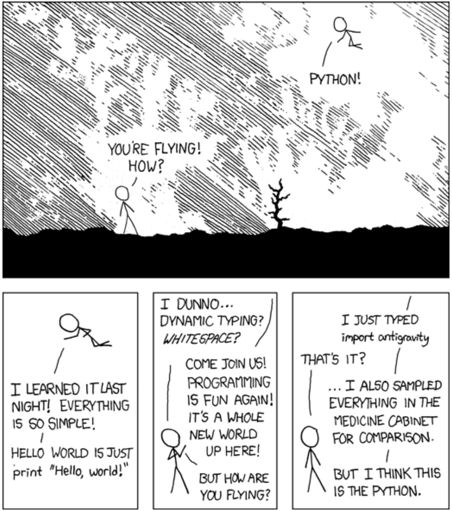
Fear and loathing in programming languages – love

Naturalmente ...xkcd: `https://xkcd.com/353/`



Fear and loathing in programming languages – love

Naturalmente . . . xkcd: `https://xkcd.com/353/`



I wrote 20 short programs in Python yesterday. It was wonderful. Perl, I'm leaving you.

(dude, we're not programming yet)

Where does Python fit?

Classifications of programming languages

imperative Lower-level, functions tell computer how to manipulate data.

- procedural FORTRAN, Pascal, C
- object-oriented Smalltalk
- multi-paradigm C++, Python

declarative State relationships, language "makes it happen."

- logical Prolog
- functional Lisp, Haskell

Where does Python fit?

Classifications of programming languages

imperative Lower-level, functions tell computer how to manipulate data.

- procedural FORTRAN, Pascal, C
- object-oriented Smalltalk
- multi-paradigm C++, Python

declarative State relationships, language "makes it happen."

- logical Prolog
- functional Lisp, Haskell

In truth most languages are multi-paradigm, these are fanciful classifications, sometimes useful, sometimes misleading. Think of striking versus grappling in martial arts.

(dude, we're not programming yet)

Terminology

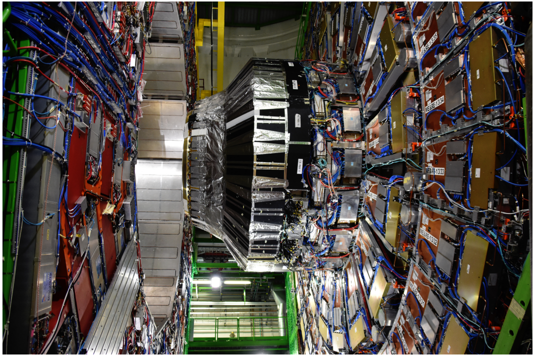
When talking about computer programming:

Attitude toward terminology
: Suspend one's uncertainty.

Complexity
: Software is enormously more complex than even the most elaborate hardware.

Growth of the field
: The field grows so quickly that it is daunting to keep up with the terminology.

Longevity of concepts
: Need to develop a talent to latch on to ideas that last (Neil Young's "coin that won't get tossed".)



Terminology

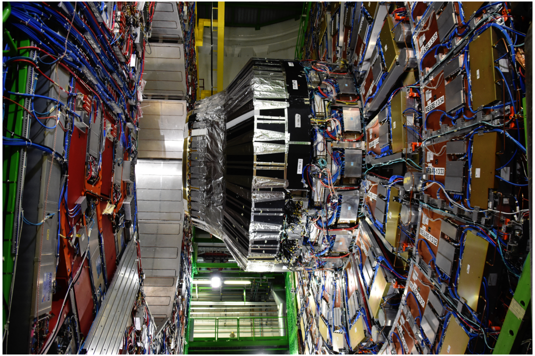
When talking about computer programming:

Attitude toward terminology Suspend one's uncertainty.

Complexity Software is enormously more complex than even the most elaborate hardware.

Growth of the field The field grows so quickly that it is daunting to keep up with the terminology.

Longevity of concepts Need to develop a talent to latch on to ideas that last (Neil Young's "coin that won't get tossed".)



A tiny bit of the Large Handron Collider (LHC) at CERN: the hardware is complex.

(dude, we're not programming yet)

Outline

Motivation, Goals, and plan

Elementary python

- Tutorial
- Our program
- Skeletons

Outline

Eearly examples – 1

At the interpreter prompt

Hello world

```
$ python3
>>> print('hello, world')
```

Eearly examples – 1

At the interpreter prompt

Hello world

```
$ python3
>>> print('hello, world')
```

Python as a calculator

```
>>> print(7*4)
>>> 7*4
>>> 125 / 13.5
>>> import math
>>> math.sqrt(1.7 + 32/17.1)
```

Eearly examples – 1

At the interpreter prompt

Hello world

```
$ python3
>>> print('hello, world')
```

introducing variables

```
>>> x = 7
>>> y = 4
>>> x*y
>>> print(x*y)
```

Python as a calculator

```
>>> print(7*4)
>>> 7*4
>>> 125 / 13.5
>>> import math
>>> math.sqrt(1.7 + 32/17.1)
```

Eearly examples – 1

At the interpreter prompt

Hello world

```
$ python3
>>> print('hello, world')
```

introducing variables

```
>>> x = 7
>>> y = 4
>>> x*y
>>> print(x*y)
```

Python as a calculator

```
>>> print(7*4)
>>> 7*4
>>> 125 / 13.5
>>> import math
>>> math.sqrt(1.7 + 32/17.1)
```

pause: are we all here?

- This is the time to make sure that everyone is helping their neighbor get the interpreter going on their system.

Eearly examples – 2

At the interpreter prompt

for loop

```
>>> for i in range(16):
...     print(i, '   ', i*i, '   ', i*i*i)
```

Eearly examples – 2

At the interpreter prompt

for loop

```
>>> for i in range(16):
...     print(i, '    ', i*i, '    ', i*i*i)
```

Celsius to Fahrenheit

```
>>> for degC in range(101):
...     degF = 32 + (9.0/5.0) * degC
...     print(degC, '    ', degF)
```

Eearly examples – 2

At the interpreter prompt

for loop

```
>>> for i in range(16):
...     print(i, '   ', i*i, '   ', i*i*i)
```

Celsius to Fahrenheit

```
>>> for degC in range(101):
...     degF = 32 + (9.0/5.0) * degC
...     print(degC, '    ', degF)
```

pause and early lessons

Eearly examples – 2

At the interpreter prompt

for loop

```
>>> for i in range(16):
...     print(i, '   ', i*i, '   ', i*i*i)
```

Celsius to Fahrenheit

```
>>> for degC in range(101):
...     degF = 32 + (9.0/5.0) * degC
...     print(degC, '    ', degF)
```

pause and early lessons

- Check on your neighbor again.

Eearly examples – 2

At the interpreter prompt

for loop

```
>>> for i in range(16):
...     print(i, '    ', i*i, '    ', i*i*i)
```

Celsius to Fahrenheit

```
>>> for degC in range(101):
...     degF = 32 + (9.0/5.0) * degC
...     print(degC, '    ', degF)
```

pause and early lessons

- Check on your neighbor again.
- **The purpose of computers is to automate repetitive tasks.**

Eearly examples – 2

At the interpreter prompt

for loop

```
>>> for i in range(16):
...     print(i, '   ', i*i, '   ', i*i*i)
```

Celsius to Fahrenheit

```
>>> for degC in range(101):
...     degF = 32 + (9.0/5.0) * degC
...     print(degC, '   ', degF)
```

pause and early lessons

- Check on your neighbor again.
- **The purpose of computers is to automate repetitive tasks.**
- We use the interpreter for quickies: two or three lines.

Early examples – 3

Using an editor - Geany is an OK default if you don't have a favorite

Gaussian sum: file gauss-sum.py

```
N = 100
sum = 0
for i in range(1, N+1):
    sum = sum + i
print('sum was:', sum)
print('gauss says:', N*(N+1) / 2)
```

for loop with arithmetic: file for-loop.py

```
import math
for i in range(16):
    print(i, '   ', i*i, '   ', i*i*i, '   ', math.sqrt(i))
```

To run it

```
$ python3 gauss-sum.py
$ python3 for-loop.py
```

Introducing functions – in the interpreter

Functions in the interpreter

```
>>> def sum_gauss(N):
...     return (N*(N+1)) / 2
... ## [hit enter a second time]
>>> sum_gauss(100), sum_gauss(1000)
>>> def factorial(n):
...     if n == 0:
...         return 1
...     else:
...         return n*factorial(n-1)
...
>>> for i in range(13):
...     print(i, factorial(i))
...
0 1
1 1
2 2
3 6
4 24
5 120
6 720
7 5040
8 40320
9 362880
10 3628800
12 479001600
```

Terminology related to functions

Introducing functions – in the interpreter

Functions in the interpreter

```
>>> def sum_gauss(N):
...     return (N*(N+1)) / 2
... ## [hit enter a second time]
>>> sum_gauss(100), sum_gauss(1000)
>>> def factorial(n):
...     if n == 0:
...         return 1
...     else:
...         return n*factorial(n-1)
...
>>> for i in range(13):
...     print(i, factorial(i))
...
0 1
1 1
2 2
3 6
4 24
5 120
6 720
7 5040
8 40320
9 362880
10 3628800
12 479001600
```

Terminology related to functions

function block
: In this case the bock is the body of the function: that part which depends on the "`def sum_gauss(N):`"

Introducing functions – in the interpreter

Functions in the interpreter

```
>>> def sum_gauss(N):
...     return (N*(N+1)) / 2
... ## [hit enter a second time]
>>> sum_gauss(100), sum_gauss(1000)
>>> def factorial(n):
...     if n == 0:
...         return 1
...     else:
...         return n*factorial(n-1)
...
>>> for i in range(13):
...     print(i, factorial(i))
...
0 1
1 1
2 2
3 6
4 24
5 120
6 720
7 5040
8 40320
9 362880
10 3628800
12 479001600
```

Terminology related to functions

function block In this case the bock is the body of the function: that part which depends on the "`def sum_gauss(N):`"

general block In general every python construct of which ends with a colon and start some indented lines of code.

Introducing functions – in the interpreter

Functions in the interpreter

```
>>> def sum_gauss(N):
...     return (N*(N+1)) / 2
... ## [hit enter a second time]
>>> sum_gauss(100), sum_gauss(1000)
>>> def factorial(n):
...     if n == 0:
...         return 1
...     else:
...         return n*factorial(n-1)
...
>>> for i in range(13):
...     print(i, factorial(i))
...
0 1
1 1
2 2
3 6
4 24
5 120
6 720
7 5040
8 40320
9 362880
10 3628800
12 479001600
```

Terminology related to functions

function block In this case the bock is the body of the function: that part which depends on the "`def sum_gauss(N):`"

general block In general every python construct of which ends with a colon and start some indented lines of code.

argument Information that you pass to the function so it knows what to work on.

Introducing functions – in the interpreter

Functions in the interpreter

```
>>> def sum_gauss(N):
...     return (N*(N+1)) / 2
... ## [hit enter a second time]
>>> sum_gauss(100), sum_gauss(1000)
>>> def factorial(n):
...     if n == 0:
...         return 1
...     else:
...         return n*factorial(n-1)
...
>>> for i in range(13):
...     print(i, factorial(i))
...
0 1
1 1
2 2
3 6
4 24
5 120
6 720
7 5040
8 40320
9 362880
10 3628800
12 479001600
```

Terminology related to functions

function block In this case the bock is the body of the function: that part which depends on the "`def sum_gauss(N):`"

general block In general every python construct of which ends with a colon and start some indented lines of code.

argument Information that you pass to the function so it knows what to work on.

return value Information passed back to you by the function.

Functions and program structure

Gaussian sum program with functions

```
def main():
    gsum = sum_gauss(100)
    bfsum = sum_brute_force(100)
    print('sum was:', bfsum)
    print('gauss says:', gsum)
    if gsum == bfsum:
        print('they were the same')
    else:
        print('they were different')

def sum_gauss(N):
    """Gauss's sum rule to calculate the sum of the first N numbers."""
    return (N*(N+1)) / 2

def sum_brute_force(N):
    """Calculate the sum of the first N numbers with brute force."""
    sum = 0
    for i in range(1, N+1):
        sum = sum + i
    return sum

main()
```

More things to notice

Functions and program structure

Gaussian sum program with functions

```
def main():
    gsum = sum_gauss(100)
    bfsum = sum_brute_force(100)
    print('sum was:', bfsum)
    print('gauss says:', gsum)
    if gsum == bfsum:
        print('they were the same')
    else:
        print('they were different')

def sum_gauss(N):
    """Gauss's sum rule to calculate the sum of the first N numbers."""
    return (N*(N+1)) / 2

def sum_brute_force(N):
    """Calculate the sum of the first N numbers with brute force."""
    sum = 0
    for i in range(1, N+1):
        sum = sum + i
    return sum

main()
```

More things to notice

- Documentation blocks using Python's `"""`.

Functions and program structure

Gaussian sum program with functions

```
def main():
    gsum = sum_gauss(100)
    bfsum = sum_brute_force(100)
    print('sum was:', bfsum)
    print('gauss says:', gsum)
    if gsum == bfsum:
        print('they were the same')
    else:
        print('they were different')

def sum_gauss(N):
    """Gauss's sum rule to calculate the sum of the first N numbers."""
    return (N*(N+1)) / 2

def sum_brute_force(N):
    """Calculate the sum of the first N numbers with brute force."""
    sum = 0
    for i in range(1, N+1):
        sum = sum + i
    return sum

main()
```

More things to notice

- Documentation blocks using Python's `"""`.
- We have a `main()` function!

Functions and program structure

Gaussian sum program with functions

```
def main():
    gsum = sum_gauss(100)
    bfsum = sum_brute_force(100)
    print('sum was:', bfsum)
    print('gauss says:', gsum)
    if gsum == bfsum:
        print('they were the same')
    else:
        print('they were different')

def sum_gauss(N):
    """Gauss's sum rule to calculate the sum of the first N numbers."""
    return (N*(N+1)) / 2

def sum_brute_force(N):
    """Calculate the sum of the first N numbers with brute force."""
    sum = 0
    for i in range(1, N+1):
        sum = sum + i
    return sum

main()
```

More things to notice

- Documentation blocks using Python's `"""`.
- We have a `main()` function!
- Python's use of indentation instead of `{block}` or **begin** `block` **end** can cause the "`return sum`" statement to get mis-indented.

Functions and program structure

Gaussian sum program with functions

```
def main():
    gsum = sum_gauss(100)
    bfsum = sum_brute_force(100)
    print('sum was:', bfsum)
    print('gauss says:', gsum)
    if gsum == bfsum:
        print('they were the same')
    else:
        print('they were different')

def sum_gauss(N):
    """Gauss's sum rule to calculate the sum of the first N numbers."""
    return (N*(N+1)) / 2

def sum_brute_force(N):
    """Calculate the sum of the first N numbers with brute force."""
    sum = 0
    for i in range(1, N+1):
        sum = sum + i
    return sum

main()
```

More things to notice

- Documentation blocks using Python's `"""`.
- We have a `main()` function!
- Python's use of indentation instead of `{block}` or **begin** `block` **end** can cause the "`return sum`" statement to get mis-indented.
- The function `sum_brute_force` uses an "accumulator" paradigm. Let's remember that one.

Data types

Exploring data types at the interpreter - gleaning from examples

Numbers

```
$ python3
>>> a = 27
>>> b = 12
>>> a*b
>>> a/b
>>> a // b
>>> a % b
>>> x = 7.2
>>> a*x
>>> y = 3.141592654
>>> x*y
>>> type(a)
>>> type(a*x)
>>> type(a*b)
>>> type(x*y)
```

Introducing strings

```
>>> s = 'hello'
>>> t = 'world'
>>> print(s, t)
>>> s + t
>>> s + ' ' + t
>>> s[0], s[1]
>>> (s + ' ' + t)[8]
>>> (s + ' ' + t)[42]
```

Function on strings

```
>>> def prepend_first_letter(s):
...     s = s[0] + s[0] + s[0] + s
...     return s
>>> my_str = 'dude'
>>> result = prepend_first_letter(my_str)
>>> my_str, result
```

Introducing lists

```
$ python3
>>> mylist = [2.5, 17, 'dude']
>>> print(mylist)
>>> mylist
>>> mylist[0]
>>> mylist[1]
>>> mylist[2]
```

AAAARGHH: repetitive task alert!!

```
>>> for i in range(3):
...     print(i, mylist[i])
>>> for item in mylist:
...     print('item is:', item)
>>> print(len(mylist))
>>> for i in range(len(mylist)):
...     print(i, mylist[i])
```

More play with types

```
$ python3 # not putting >>> prompt here
type(4)
n = 42
type(n)
type(4.4)
x = 3.141592654
type(x)
type(2.0), type(2)
type('hello world')
s = 'hello world'
type(s)
mylist = [2.5, 17, 'dude']
mylist
type(mylist)
mylist[0]
type(mylist[0])
len(mylist)
type(len(mylist))
mylist
for i, item in enumerate(mylist):
    print('ind:', i, 'list-item:', item,
          'type:', type(item))
```

More language features, and converting types

Logic

```
>>> if 2 > 3:
...     print('the impossible just happened')
... else:
...     print('phew: 2 is not greater than 3')
>>> x = 7
>>> y = 8
>>> if x*y < (x+1)*(y+1):
...     print('that made sense')
>>> x, y
>>> x == y
>>> x, y
>>> x = y
>>> x, y
>>> x == y
```

More language features, and converting types

Logic

```
>>> if 2 > 3:
...     print('the impossible just happened')
... else:
...     print('phew: 2 is not greater than 3')
>>> x = 7
>>> y = 8
>>> if x*y < (x+1)*(y+1):
...     print('that made sense')
>>> x, y
>>> x == y
>>> x, y
>>> x = y
>>> x, y
>>> x == y
```

Type conversions

```
>>> ns = '42'
>>> n = 42
>>> print(n)
>>> print(ns)
>>> n == ns
>>> type(n), type(ns)
>>> n, str(n)
>>> str(n) == ns
>>> ns, int(ns)
>>> n == int(ns)
>>> type(str(n)), type(ns)
```

More language features, and converting types

Logic

```
>>> if 2 > 3:
...     print('the impossible just happened')
... else:
...     print('phew: 2 is not greater than 3')
>>> x = 7
>>> y = 8
>>> if x*y < (x+1)*(y+1):
...     print('that made sense')
>>> x, y
>>> x == y
>>> x, y
>>> x = y
>>> x, y
>>> x == y
```

Type conversions

```
>>> ns = '42'
>>> n = 42
>>> print(n)
>>> print(ns)
>>> n == ns
>>> type(n), type(ns)
>>> n, str(n)
>>> str(n) == ns
>>> ns, int(ns)
>>> n == int(ns)
>>> type(str(n)), type(ns)
```

Taking stock

More language features, and converting types

Logic

```
>>> if 2 > 3:
...     print('the impossible just happened')
... else:
...     print('phew: 2 is not greater than 3')
>>> x = 7
>>> y = 8
>>> if x*y < (x+1)*(y+1):
...     print('that made sense')
>>> x, y
>>> x == y
>>> x, y
>>> x = y
>>> x, y
>>> x == y
```

Type conversions

```
>>> ns = '42'
>>> n = 42
>>> print(n)
>>> print(ns)
>>> n == ns
>>> type(n), type(ns)
>>> n, str(n)
>>> str(n) == ns
>>> ns, int(ns)
>>> n == int(ns)
>>> type(str(n)), type(ns)
```

Taking stock

- Are we comfortable with the syntax? (Commas, indentation, . . .)

More language features, and converting types

Logic

```
>>> if 2 > 3:
...     print('the impossible just happened')
... else:
...     print('phew: 2 is not greater than 3')
>>> x = 7
>>> y = 8
>>> if x*y < (x+1)*(y+1):
...     print('that made sense')
>>> x, y
>>> x == y
>>> x, y
>>> x = y
>>> x, y
>>> x == y
```

Type conversions

```
>>> ns = '42'
>>> n = 42
>>> print(n)
>>> print(ns)
>>> n == ns
>>> type(n), type(ns)
>>> n, str(n)
>>> str(n) == ns
>>> ns, int(ns)
>>> n == int(ns)
>>> type(str(n)), type(ns)
```

Taking stock

- Are we comfortable with the syntax? (Commas, indentation, . . .)
- Are we comfortable with the data types we have seen so far? (integers, floats, strings, lists)

More language features, and converting types

Logic

```
>>> if 2 > 3:
...     print('the impossible just happened')
... else:
...     print('phew: 2 is not greater than 3')
>>> x = 7
>>> y = 8
>>> if x*y < (x+1)*(y+1):
...     print('that made sense')
>>> x, y
>>> x == y
>>> x, y
>>> x = y
>>> x, y
>>> x == y
```

Type conversions

```
>>> ns = '42'
>>> n = 42
>>> print(n)
>>> print(ns)
>>> n == ns
>>> type(n), type(ns)
>>> n, str(n)
>>> str(n) == ns
>>> ns, int(ns)
>>> n == int(ns)
>>> type(str(n)), type(ns)
```

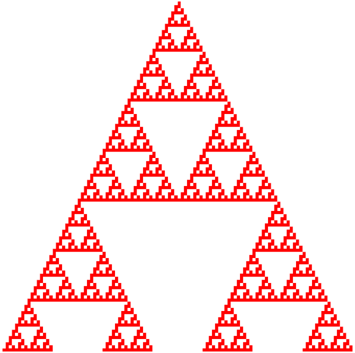
Taking stock

- Are we comfortable with the syntax? (Commas, indentation, ...)
- Are we comfortable with the data types we have seen so far? (integers, floats, strings, lists)
- Shall we start writing a program?

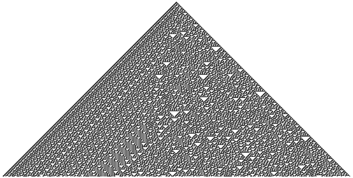
Outline

Our program

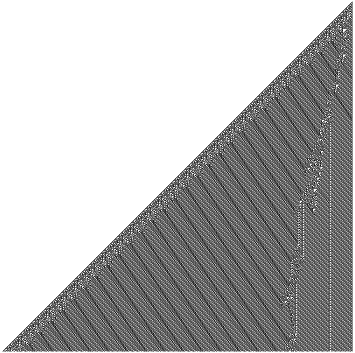
Visualizing cellular automata



Rule 90: the Sierpiński gasket.



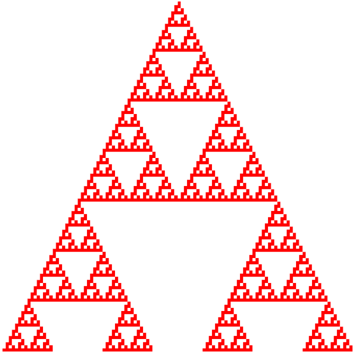
Rule 30



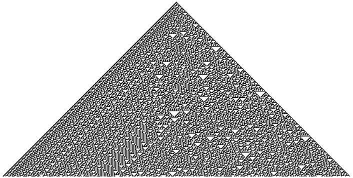
Rule 110

Our program

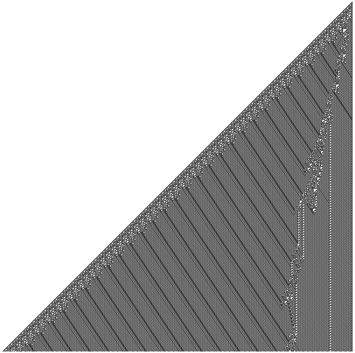
Visualizing cellular automata



Rule 90: the Sierpiński gasket.



Rule 30



Rule 110

(Shift to a window to show an animation of 1D and 2D cellular automata.)

Outline

The complexities we handle as beginners

The complexities we handle as beginners

Getting comfortable with syntax

The complexities we handle as beginners

Getting comfortable with syntax

Lots of hello-world-ish examples.

The complexities we handle as beginners

Getting comfortable with syntax

Lots of hello-world-ish examples.

Getting good with tools

The complexities we handle as beginners

Getting comfortable with syntax
Lots of hello-world-ish examples.

Getting good with tools
Roll up your sleeves and do the lonely work of the full emacs tutorial (or other programming editor).

The complexities we handle as beginners

Getting comfortable with syntax
: Lots of hello-world-ish examples.

Getting good with tools
: Roll up your sleeves and do the lonely work of the full emacs tutorial (or other programming editor).

Overcoming the "activation barrier"

The complexities we handle as beginners

Getting comfortable with syntax
Lots of hello-world-ish examples.

Getting good with tools
Roll up your sleeves and do the lonely work of the full emacs tutorial (or other programming editor).

Overcoming the “activation barrier”
Use the skeleton approach.

Start with a skeleton - ca-skel-0.py

```
#! /usr/bin/env python3
# first attempt: just starting
def main():
    print('future home of cellular automata code')

main()
```

Listing 13: ca-skel-0.py

The complexities we handle as beginners

Getting comfortable with syntax
Lots of hello-world-ish examples.

Getting good with tools
Roll up your sleeves and do the lonely work of the full emacs tutorial (or other programming editor).

Overcoming the "activation barrier"
Use the skeleton approach.

Start with a skeleton - ca-skel-0.py

```
#! /usr/bin/env python3
# first attempt: just starting
def main():
    print('future home of cellular automata code')

main()
```

Listing 15: ca-skel-0.py

First actions: I want to see some output!

```
#! /usr/bin/env python3
# next attempt: explore the data representation for a CA row
def main():
    print('for now just printing out a single row')
    n_cells = 79
    row = [0]*n_cells           # row is a list of 0 or 1 values
    row[7] = 1
    row[24] = 1
    row[50] = 1
    row[75] = 1
    print(row)
    for cell in row:
        if cell == 0:
            print(' ', end="")
        else:
            print('x', end="")
    print()

main()
```

Listing 16: ca-skel-1.py

The “English language narrative”

The “English language narrative”

Modularize it

```
#! /usr/bin/env python3
# next attempt - make it modular: write some functions
def main():
    n_steps = 100
    n_cells = 79
    row = first_row_empty(n_cells)
    set_some_cells(row, [7, 24, 50, 75])
    print_row(row)
    for i in range(n_steps):
        row = take_step(row)

def first_row_empty(n_cells):
    """Make a first row where all cells are 0."""
    row = [0]*n_cells           # row is a list of 0 or 1 values
    return row

def set_some_cells(row, cell_list):
    """Modifies row by setting to 1 all the cells listed in cell_list."""
    for cell_no in cell_list:
        row[cell_no] = 1

def print_row(row):
    """Prints a cellular automaton row, a blank for 0 and an 'x' for 1."""
    for cell in row:
        if cell == 0:
            print(' ', end="")
        else:
            print('x', end="")
    print_row()


main()
```

Listing 18: ca-skel-2.py

Our main function

```
def main():
    n_steps = 100
    n_cells = 79
    row = first_row_empty(n_cells)
    set_some_cells(row, [7, 24, 50, 75])
    print_row(row)
    for i in range(n_steps):
```

The “English language narrative”

Modularize it

```
#! /usr/bin/env python3
# next attempt - make it modular: write some functions
def main():
    n_steps = 100
    n_cells = 79
    row = first_row_empty(n_cells)
    set_some_cells(row, [7, 24, 50, 75])
    print_row(row)
    for i in range(n_steps):
        row = take_step(row)

def first_row_empty(n_cells):
    """Make a first row where all cells are 0."""
    row = [0]*n_cells           # row is a list of 0 or 1 values
    return row

def set_some_cells(row, cell_list):
    """Modifies row by setting to 1 all the cells listed in cell_list."""
    for cell_no in cell_list:
        row[cell_no] = 1

def print_row(row):
    """Prints a cellular automaton row, a blank for 0 and an 'x' for 1."""
    for cell in row:
        if cell == 0:
            print(' ', end="")
        else:
            print('x', end="")
    print_row()


main()
```

Listing 19: `ca-skel-2.py`

Our main function

```
def main():
    n_steps = 100
    n_cells = 79
    row = first_row_empty(n_cells)
    set_some_cells(row, [7, 24, 50, 75])
    print_row(row)
    for i in range(n_steps):
```

Telling the story

The size of our cellular space is 79. We create a row of deactivated cells and we activate a few of those cells. Then we print what that row looks like.

The "English language narrative"

Modularize it

```
#! /usr/bin/env python3
# next attempt - make it modular: write some functions
def main():
    n_steps = 100
    n_cells = 79
    row = first_row_empty(n_cells)
    set_some_cells(row, [7, 24, 50, 75])
    print_row(row)
    for i in range(n_steps):
        row = take_step(row)

def first_row_empty(n_cells):
    """Make a first row where all cells are 0."""
    row = [0]*n_cells           # row is a list of 0 or 1 values
    return row

def set_some_cells(row, cell_list):
    """Modifies row by setting to 1 all the cells listed in cell_list."""
    for cell_no in cell_list:
        row[cell_no] = 1

def print_row(row):
    """Prints a cellular automaton row, a blank for 0 and an 'x' for 1."""
    for cell in row:
        if cell == 0:
            print(' ', end="")
        else:
            print('x', end="")
    print_row()


main()
```

Listing 20: `ca-skel-2.py`

Our main function

```
def main():
    n_steps = 100
    n_cells = 79
    row = first_row_empty(n_cells)
    set_some_cells(row, [7, 24, 50, 75])
    print_row(row)
    for i in range(n_steps):
```

Telling the story

The size of our cellular space is 79. We create a row of deactivated cells and we activate a few of those cells. Then we print what that row looks like.

Every program should look like a `main()` function that calls other functions. This is called a "top-down" view of the program.

Expanding our program to take steps

Updating main()

```
def main():
    n_steps = 100
    n_cells = 150
    row = first_row_empty(n_cells)
    set_some_cells(row, [7, 24, 50, 75]) # initial values
    print_row(row)
    for i in range(n_steps):
        row = take_step_sierpinski(row) # new row from rule 30
        print_row(row)
```

Expanding our program to take steps

Updating main()

```
def main():
    n_steps = 100
    n_cells = 150
    row = first_row_empty(n_cells)
    set_some_cells(row, [7, 24, 50, 75]) # initial values
    print_row(row)
    for i in range(n_steps):
        row = take_step_sierpinski(row) # new row from rule 30
        print_row(row)
```

Taking a step

```
def take_step_sierpinski(row):
    """a single iteration of the cellular automaton"""
    n_cells = len(row)
    new_row = [0]*n_cells  # paradigm: make it blank, then fill it
    for i in range(n_cells):
        # new python ideas: modular arithmetic to wrap around the
        # ends of the list
        neighbors = [row[(i - 1 + n_cells) % n_cells], row[i], row
        [(i + 1) % n_cells]]
        if neighbors in [[1,1,1], [1,0,1], [0,1,0], [0,0,0]]:
            new_cell_value = 1
        else:
            new_cell_value = 0
        new_row[i] = new_cell_value
    return new_row
```

New features

Expanding our program to take steps

Updating main()

```
def main():
    n_steps = 100
    n_cells = 150
    row = first_row_empty(n_cells)
    set_some_cells(row, [7, 24, 50, 75]) # initial values
    print_row(row)
    for i in range(n_steps):
        row = take_step_sierpinski(row) # new row from rule 30
        print_row(row)
```

Taking a step

```
def take_step_sierpinski(row):
    """a single iteration of the cellular automaton"""
    n_cells = len(row)
    new_row = [0]*n_cells  # paradigm: make it blank, then fill it
    for i in range(n_cells):
        # new python ideas: modular arithmetic to wrap around the
        # ends of the list
        neighbors = [row[(i - 1 + n_cells) % n_cells], row[i], row
        [(i + 1) % n_cells]]
        if neighbors in [[1,1,1], [1,0,1], [0,1,0], [0,0,0]]:
            new_cell_value = 1
        else:
            new_cell_value = 0
        new_row[i] = new_cell_value
    return new_row
```

New features

- New way of making a list: `[0]*n_cells`.

Expanding our program to take steps

Updating main()

```
def main():
    n_steps = 100
    n_cells = 150
    row = first_row_empty(n_cells)
    set_some_cells(row, [7, 24, 50, 75]) # initial values
    print_row(row)
    for i in range(n_steps):
        row = take_step_sierpinski(row) # new row from rule 30
        print_row(row)
```

Taking a step

```
def take_step_sierpinski(row):
    """a single iteration of the cellular automaton"""
    n_cells = len(row)
    new_row = [0]*n_cells  # paradigm: make it blank, then fill it
    for i in range(n_cells):
        # new python ideas: modular arithmetic to wrap around the
        # ends of the list
        neighbors = [row[(i - 1 + n_cells) % n_cells], row[i], row
        [(i + 1) % n_cells]]
        if neighbors in [[1,1,1], [1,0,1], [0,1,0], [0,0,0]]:
            new_cell_value = 1
        else:
            new_cell_value = 0
        new_row[i] = new_cell_value
    return new_row
```

New features

- New way of making a list: `[0]*n_cells`.
- **in** operator for lists

What are we unhappy about?

Expanding our program to take steps

Updating main()

```
def main():
    n_steps = 100
    n_cells = 150
    row = first_row_empty(n_cells)
    set_some_cells(row, [7, 24, 50, 75]) # initial values
    print_row(row)
    for i in range(n_steps):
        row = take_step_sierpinski(row) # new row from rule 30
        print_row(row)
```

Taking a step

```
def take_step_sierpinski(row):
    """a single iteration of the cellular automaton"""
    n_cells = len(row)
    new_row = [0]*n_cells  # paradigm: make it blank, then fill it
    for i in range(n_cells):
        # new python ideas: modular arithmetic to wrap around the
        # ends of the list
        neighbors = [row[(i - 1 + n_cells) % n_cells], row[i], row
        [(i + 1) % n_cells]]
        if neighbors in [[1,1,1], [1,0,1], [0,1,0], [0,0,0]]:
            new_cell_value = 1
        else:
            new_cell_value = 0
        new_row[i] = new_cell_value
    return new_row
```

New features

- New way of making a list: `[0]*n_cells`.
- **in** operator for lists

What are we unhappy about?

- Hard-coded function to only do the Sierpiński rule.

Expanding our program to take steps

Updating main()

```
def main():
    n_steps = 100
    n_cells = 150
    row = first_row_empty(n_cells)
    set_some_cells(row, [7, 24, 50, 75]) # initial values
    print_row(row)
    for i in range(n_steps):
        row = take_step_sierpinski(row) # new row from rule 30
        print_row(row)
```

Taking a step

```
def take_step_sierpinski(row):
    """a single iteration of the cellular automaton"""
    n_cells = len(row)
    new_row = [0]*n_cells  # paradigm: make it blank, then fill it
    for i in range(n_cells):
        # new python ideas: modular arithmetic to wrap around the
        # ends of the list
        neighbors = [row[(i - 1 + n_cells) % n_cells], row[i], row
        [(i + 1) % n_cells]]
        if neighbors in [[1,1,1], [1,0,1], [0,1,0], [0,0,0]]:
            new_cell_value = 1
        else:
            new_cell_value = 0
        new_row[i] = new_cell_value
    return new_row
```

New features

- New way of making a list: `[0]*n_cells`.
- **in** operator for lists

What are we unhappy about?

- Hard-coded function to only do the Sierpiński rule.
- Checking if `neighbors` is in a hard-coded list of neighbor triplets is not beautiful programming.

Run it!

```
$ python3 ca-first-steps.py
```

How to encode them

The lonely work of programming: representations

Generalizing

The tables below show how to represent **any** CA rule (for 2 states and a single neighbor on each side) as a **string of 8 binary digits.**

Cellular automata rules: rule 30, i.e. 00011110

current pattern	111	110	101	100	011	010	001	000
new state for center cell	0	0	0	1	1	1	1	0

Cellular automata rules: rule 90, i.e. 01011010, the Sierpiński gasket

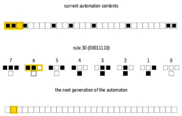
current pattern	111	110	101	100	011	010	001	000
new state for center cell	0	1	0	1	1	0	1	0

Cellular automata rules: rule 110, i.e. 01101110

current pattern	111	110	101	100	011	010	001	000
new state for center cell	0	1	1	0	1	1	1	0

How to encode them

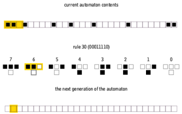
Mapping a neighborhood into a digit.



Rule 30: details of the mapping.

How to encode them

Mapping a neighborhood into a digit.



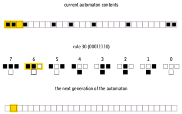
Rule 30: details of the mapping.

Naïve Python code for rule 30

```python
neighbors = [row[(center - 1 + n_cells) % n_cells],
             row[center], row[(center + 1) % n_cells]]
if neighbors in [[1,0,0], [0,1,1], [0,1,0]]:
    new_cell_value = 1
else:
    new_cell_value = 0
new_row[center] = new_cell_value
```

How to encode them

Mapping a neighborhood into a digit.



Rule 30: details of the mapping.

Naïve Python code for rule 30

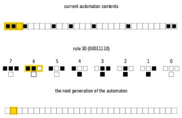
```
neighbors = [row[(center - 1 + n_cells) % n_cells],
             row[center], row[(center + 1) % n_cells]]
if neighbors in [[1,0,0], [0,1,1], [0,1,0]]:
    new_cell_value = 1
else:
    new_cell_value = 0
new_row[center] = new_cell_value
```

More general implementation for any rule

```
def new_cell_with_rule(rule, neighbors):
    """Applies a rule encoded as a binary string -- since a neighborhood
    of 3 binary cells can have 8 possible patterns, it's a string of 8
    bits.  You can modify it to be any of the 256 possible strings of
    8 bits.  I provide a couple of examples.  You can try many others."""
    if not rule:
        rule = '01101000'       # the default rule
    rule_index = neighbors[0] + 2*neighbors[1] + 4*neighbors[2]
    cell = int(rule[rule_index])
    return cell
```

How to encode them

Mapping a neighborhood into a digit.



Rule 30: details of the mapping.

Naïve Python code for rule 30

```
neighbors = [row[(center - 1 + n_cells) % n_cells],
             row[center], row[(center + 1) % n_cells]]
if neighbors in [[1,0,0], [0,1,1], [0,1,0]]:
    new_cell_value = 1
else:
    new_cell_value = 0
new_row[center] = new_cell_value
```

More general implementation for any rule

```
def new_cell_with_rule(rule, neighbors):
    """Applies a rule encoded as a binary string -- since a neighborhood
    of 3 binary cells can have 8 possible patterns, it's a string of 8
    bits.  You can modify it to be any of the 256 possible strings of
    8 bits.  I provide a couple of examples.  You can try many others."""
    if not rule:
        rule = '01101000'       # the default rule
    rule_index = neighbors[0] + 2*neighbors[1] + 4*neighbors[2]
    cell = int(rule[rule_index])
    return cell
```

This is all put together in the file `full-ca-program.py`

Outline

Dictionaries: Python's "killer feature"

Basics of object-oriented python

- Stories of programming languages
- Object Oriented Programming (OOP)

The need for dictionaries

The need for dictionaries

Accessing within aggregate types

- `print(my_list[7], my_list[-1])`
- `print(my_str[2], my_str[7:12])`

The need for dictionaries

Accessing within aggregate types

- `print(my_list[7], my_list[-1])`
- `print(my_str[2], my_str[7:12])`

Structured data with a list

Describe a person as a list of their characteristics:

```
def main():
    boyd_record = ['Boyd', 1971,
                   '543-81-5481', '+1-606-555-6173']
    print_person(boyd_record)

def print_person(person):
    print('==== record for', person[0], '=====')
    print('name:', person[0])
    print('birth-year:', person[1])
    print('SSN:', person[2])
    print('phone:', person[3])

main()
```

The need for dictionaries

Accessing within aggregate types

- `print(my_list[7], my_list[-1])`
- `print(my_str[2], my_str[7:12])`

Structured data with a list

Describe a person as a list of their characteristics:

```
def main():
    boyd_record = ['Boyd', 1971,
                   '543-81-5481', '+1-606-555-6173']
    print_person(boyd_record)

def print_person(person):
    print('==== record for', person[0], '=====')
    print('name:', person[0])
    print('birth-year:', person[1])
    print('SSN:', person[2])
    print('phone:', person[3])

main()
```

Goes south quickly

The need for dictionaries

Accessing within aggregate types

- `print(my_list[7], my_list[-1])`
- `print(my_str[2], my_str[7:12])`

Structured data with a list

Describe a person as a list of their characteristics:

```
def main():
    boyd_record = ['Boyd', 1971,
                   '543-81-5481', '+1-606-555-6173']
    print_person(boyd_record)

def print_person(person):
    print('==== record for', person[0], '=====')
    print('name:', person[0])
    print('birth-year:', person[1])
    print('SSN:', person[2])
    print('phone:', person[3])

main()
```

Goes south quickly

- You realize you should also have a surname for your record:

```
def main():
    boyd_record = ['Boyd', 'Crowder', 1971,
                   '543-81-5481',
                   '+1-606-555-6173']
    print_person(boyd_record)
```

The need for dictionaries

Accessing within aggregate types

- `print(my_list[7], my_list[-1])`
- `print(my_str[2], my_str[7:12])`

Structured data with a list

Describe a person as a list of their characteristics:

```
def main():
    boyd_record = ['Boyd', 1971,
                   '543-81-5481', '+1-606-555-6173']
    print_person(boyd_record)

def print_person(person):
    print('==== record for', person[0], '=====')
    print('name:', person[0])
    print('birth-year:', person[1])
    print('SSN:', person[2])
    print('phone:', person[3])

main()
```

Goes south quickly

- You realize you should also have a surname for your record:

```
def main():
    boyd_record = ['Boyd', 'Crowder', 1971,
                   '543-81-5481',
                   '+1-606-555-6173']
    print_person(boyd_record)
```

- Can you just add a `print('surname:', person[1])` to your `print_person()` function?

The need for dictionaries

Accessing within aggregate types

- `print(my_list[7], my_list[-1])`
- `print(my_str[2], my_str[7:12])`

Structured data with a list

Describe a person as a list of their characteristics:

```
def main():
    boyd_record = ['Boyd', 1971,
                   '543-81-5481', '+1-606-555-6173']
    print_person(boyd_record)

def print_person(person):
    print('==== record for', person[0], '=====')
    print('name:', person[0])
    print('birth-year:', person[1])
    print('SSN:', person[2])
    print('phone:', person[3])

main()
```

Goes south quickly

- You realize you should also have a surname for your record:

```
def main():
    boyd_record = ['Boyd', 'Crowder', 1971,
                   '543-81-5481',
                   '+1-606-555-6173']
    print_person(boyd_record)
```

- Can you just add a `print('surname:', person[1])` to your `print_person()` function?
- Requiring fiddly changes in disparate places - Murphy's law is lying in wait.

Introducing dictionaries – Python's "killer feature"

Index by string instead of int

```
>>> boyd_record = {'name' : 'Boyd',
                   'birth-year' : 1971}
>>> print(boyd_record)
>>> import pprint
>>> pprint.pprint(boyd_record)
>>> print(boyd_record['name'])
>>> print(boyd_record['birth-year'])
>>> print(boyd_record.keys())
>>> print(boyd_record.values())
```

Introducing dictionaries – Python's "killer feature"

Index by string instead of int

```
>>> boyd_record = {'name' : 'Boyd',
                   'birth-year' : 1971}
>>> print(boyd_record)
>>> import pprint
>>> pprint.pprint(boyd_record)
>>> print(boyd_record['name'])
>>> print(boyd_record['birth-year'])
>>> print(boyd_record.keys())
>>> print(boyd_record.values())
```

Terminology

Introducing dictionaries – Python's "killer feature"

Index by string instead of int

```
>>> boyd_record = {'name' : 'Boyd',
                   'birth-year' : 1971}
>>> print(boyd_record)
>>> import pprint
>>> pprint.pprint(boyd_record)
>>> print(boyd_record['name'])
>>> print(boyd_record['birth-year'])
>>> print(boyd_record.keys())
>>> print(boyd_record.values())
```

Terminology

key
: The string (or sometimes other object) you use to access the specific data item.

Introducing dictionaries – Python's "killer feature"

Index by string instead of int

```
>>> boyd_record = {'name' : 'Boyd',
                   'birth-year' : 1971}
>>> print(boyd_record)
>>> import pprint
>>> pprint.pprint(boyd_record)
>>> print(boyd_record['name'])
>>> print(boyd_record['birth-year'])
>>> print(boyd_record.keys())
>>> print(boyd_record.values())
```

Terminology

key
: The string (or sometimes other object) you use to access the specific data item.

value
: The value associated with (and retrieved by) that key.

Introducing dictionaries – Python's "killer feature"

Index by string instead of int

```
>>> boyd_record = {'name' : 'Boyd',
                   'birth-year' : 1971}
>>> print(boyd_record)
>>> import pprint
>>> pprint.pprint(boyd_record)
>>> print(boyd_record['name'])
>>> print(boyd_record['birth-year'])
>>> print(boyd_record.keys())
>>> print(boyd_record.values())
```

Terminology

key The string (or sometimes other object) you use to access the specific data item.

value The value associated with (and retrieved by) that key.

key-value pair For example `('name','Boyd')`

Introducing dictionaries – Python's "killer feature"

Index by string instead of int

```
>>> boyd_record = {'name' : 'Boyd',
                   'birth-year' : 1971}
>>> print(boyd_record)
>>> import pprint
>>> pprint.pprint(boyd_record)
>>> print(boyd_record['name'])
>>> print(boyd_record['birth-year'])
>>> print(boyd_record.keys())
>>> print(boyd_record.values())
```

Terminology

key The string (or sometimes other object) you use to access the specific data item.

value The value associated with (and retrieved by) that key.

key-value pair For example `('name','Boyd')`

other names Hash table, associative list.

Pro tips

Introducing dictionaries – Python's "killer feature"

Index by string instead of int

```
>>> boyd_record = {'name' : 'Boyd',
                   'birth-year' : 1971}
>>> print(boyd_record)
>>> import pprint
>>> pprint.pprint(boyd_record)
>>> print(boyd_record['name'])
>>> print(boyd_record['birth-year'])
>>> print(boyd_record.keys())
>>> print(boyd_record.values())
```

Terminology

key The string (or sometimes other object) you use to access the specific data item.

value The value associated with (and retrieved by) that key.

key-value pair For example ('name','Boyd')

other names Hash table, associative list.

Pro tips

- Always use dictionaries: find ways to fit them.

Introducing dictionaries – Python's "killer feature"

Index by string instead of int

```
>>> boyd_record = {'name' : 'Boyd',
                   'birth-year' : 1971}
>>> print(boyd_record)
>>> import pprint
>>> pprint.pprint(boyd_record)
>>> print(boyd_record['name'])
>>> print(boyd_record['birth-year'])
>>> print(boyd_record.keys())
>>> print(boyd_record.values())
```

Terminology

key The string (or sometimes other object) you use to access the specific data item.

value The value associated with (and retrieved by) that key.

key-value pair For example `('name','Boyd')`

other names Hash table, associative list.

Pro tips

- Always use dictionaries: find ways to fit them.
- `dir(boyd_record)`

Introducing dictionaries – Python's "killer feature"

Index by string instead of int

```
>>> boyd_record = {'name' : 'Boyd',
                   'birth-year' : 1971}
>>> print(boyd_record)
>>> import pprint
>>> pprint.pprint(boyd_record)
>>> print(boyd_record['name'])
>>> print(boyd_record['birth-year'])
>>> print(boyd_record.keys())
>>> print(boyd_record.values())
```

Terminology

key The string (or sometimes other object) you use to access the specific data item.

value The value associated with (and retrieved by) that key.

key-value pair For example `('name','Boyd')`

other names Hash table, associative list.

Pro tips

- Always use dictionaries: find ways to fit them.
- `dir(boyd_record)`
- `help(boyd_record)`

Introducing dictionaries – Python's "killer feature"

Index by string instead of int

```
>>> boyd_record = {'name' : 'Boyd',
                   'birth-year' : 1971}
>>> print(boyd_record)
>>> import pprint
>>> pprint.pprint(boyd_record)
>>> print(boyd_record['name'])
>>> print(boyd_record['birth-year'])
>>> print(boyd_record.keys())
>>> print(boyd_record.values())
```

Terminology

key The string (or sometimes other object) you use to access the specific data item.

value The value associated with (and retrieved by) that key.

key-value pair For example `('name','Boyd')`

other names Hash table, associative list.

Pro tips

- Always use dictionaries: find ways to fit them.
- `dir(boyd_record)`
- `help(boyd_record)`

Reads better – and try to add a field!

```
def main():
    boyd_record = {'name' : 'Boyd',
                   'birth-year' : 1971,
                   'SSN' : '543-81-5481',
                   'phone' : '+1-606-555-6173'}
    print_person(boyd_record)

def print_person(person):
    print('==== record for', person['name'], '=====')
    print('name:', person['name'])
    print('birth-year:', person['birth-year'])
    print('SSN:', person['SSN'])
    print('phone:', person['phone'])

main()
```

Dictionaries making a job trivial

A program to analyze text

Dictionaries making a job trivial

A program to analyze text

- Project gutenberg: `https://www.gutenberg.org/`

Dictionaries making a job trivial

A program to analyze text

- Project gutenberg: `https://www.gutenberg.org/`
- Remote retrieval.

Dictionaries making a job trivial

A program to analyze text

- Project gutenberg: `https://www.gutenberg.org/`
- Remote retrieval.
- Analyzing rank-frequency relations.

```
wget --continue --output-document swanns-way-english.txt \
    http://www.gutenberg.org/cache/epub/1128/pg1128.txt
```

Dictionaries making a job trivial

A program to analyze text

- Project gutenberg: `https://www.gutenberg.org/`
- Remote retrieval.
- Analyzing rank-frequency relations.

```
wget --continue --output-document swanns-way-english.txt \
     http://www.gutenberg.org/cache/epub/1128/pg1128.txt
```

The use of a dictionary: frequency counting

```
# read all the words into a list of words
# loop through words
#     if word is *not* in dictionary: freq_map[word] = 1
#     if word *is* in dictionary: freq_map[word] += 1
# [snippet from word-freq-rank.py]
        for word in word_list:
            if word in word_freq_map.keys():
                word_freq_map[word] += 1
            else:
                word_freq_map[word] = 1
```

Dictionaries making a job trivial

A program to analyze text

- Project gutenberg: `https://www.gutenberg.org/`
- Remote retrieval.
- Analyzing rank-frequency relations.

```
wget --continue --output-document swanns-way-english.txt \
     http://www.gutenberg.org/cache/epub/1128/pg1128.txt
```

The use of a dictionary: frequency counting

```
# read all the words into a list of words
# loop through words
#     if word is *not* in dictionary: freq_map[word] = 1
#     if word *is* in dictionary: freq_map[word] += 1
# [snippet from word-freq-rank.py]
        for word in word_list:
            if word in word_freq_map.keys():
                word_freq_map[word] += 1
            else:
                word_freq_map[word] = 1
```

Top-down `main()`

```
"""
Reads all the words in a file and prints information about the
rank and frequence of occurrence of words in the file.

The file should be a rather long file with a typical sampling of
words.  The ideal file would be a book downloaded from Project
Gutenberg in ascii text format.
"""

def main():
    if len(sys.argv) == 1:
        f = sys.stdin
    elif len(sys.argv) == 2:
        fname = sys.argv[1]
        f = open(fname, 'r')
    else:
        sys.stderr.write('error: use 0 or 1 arguments\n')
        sys.exit(1)

    sorted_words, word_freq_map = read_words_from_file(f)
    f.close()
    print('##  rank  word                frequency')
    for i, word in enumerate(sorted_words):
        print('%8d  %-16s  %8d' % (i+1, word, word_freq_map[word]))
```

Carry out the analysis

The full program is in the file `word-freq-rank.py`

```
wget --continue --output-document swanns-way-english.txt \
    http://www.gutenberg.org/cache/epub/1128/pg1128.txt
python3 word-freq-rank.py swanns-way.txt
## other way to run python:
chmod +x word-freq-rank.py swanns-way.txt
./word-freq-rank.py swanns-way.txt
```

Carry out the analysis

The full program is in the file `word-freq-rank.py`

```
wget --continue --output-document swanns-way-english.txt \
    http://www.gutenberg.org/cache/epub/1128/pg1128.txt
python3 word-freq-rank.py swanns-way.txt
## other way to run python:
chmod +x word-freq-rank.py swanns-way.txt
./word-freq-rank.py swanns-way.txt
```

Output

Carry out the analysis

The full program is in the file `word-freq-rank.py`

```
wget --continue --output-document swanns-way-english.txt \
    http://www.gutenberg.org/cache/epub/1128/pg1128.txt
python3 word-freq-rank.py swanns-way.txt
## other way to run python:
chmod +x word-freq-rank.py swanns-way.txt
./word-freq-rank.py swanns-way.txt
```

Output

```
##  file: swanns-way.txt
##  rank  word          frequency
       1  the             10051
       2  of               7169
       3  to               6749
       4  and              4631
       5  a                4440
       6  in               4160
       7  that             3632
       8  had              2712
       9  which            2686
      10  he               2648
      11  i                2405
```

Carry out the analysis

The full program is in the file `word-freq-rank.py`

```
wget --continue --output-document swanns-way-english.txt \
    http://www.gutenberg.org/cache/epub/1128/pg1128.txt
python3 word-freq-rank.py swanns-way.txt
## other way to run python:
chmod +x word-freq-rank.py swanns-way.txt
./word-freq-rank.py swanns-way.txt
```

Output

```
##  file: swanns-way.txt
##  rank  word          frequency
       1  the             10051
       2  of               7169
       3  to               6749
       4  and              4631
       5  a                4440
       6  in               4160
       7  that             3632
       8  had              2712
       9  which            2686
      10  he               2648
      11  i                2405
```

```
      12  was              2395
      13  her              2288
      14  it               2201
      15  as               1884
      16  she              1830
      17  for              1773
      18  with             1761
      19  would            1554
      20  my               1492
      21  his              1487
      22  not              1434
      23  at               1422
      24  but              1171
```

Carry out the analysis

The full program is in the file word-freq-rank.py

```
wget --continue --output-document swanns-way-english.txt \
    http://www.gutenberg.org/cache/epub/1128/pg1128.txt
python3 word-freq-rank.py swanns-way.txt
## other way to run python:
chmod +x word-freq-rank.py swanns-way.txt
./word-freq-rank.py swanns-way.txt
```

Output

```
## file: swanns-way.txt
## rank  word          frequency
      1  the               10051
      2  of                 7169
      3  to                 6749
      4  and                4631
      5  a                  4440
      6  in                 4160
      7  that               3632
      8  had                2712
      9  which              2686
     10  he                 2648
     11  i                  2405
```

```
     12  was                2395
     13  her                2288
     14  it                 2201
     15  as                 1884
     16  she                1830
     17  for                1773
     18  with               1761
     19  would              1554
     20  my                 1492
     21  his                1487
     22  not                1434
     23  at                 1422
     24  but                1171
```

```
  13447  rambling              1
  13448  laboured              1
  13449  quimperle             1
  13450  e-mail                1
  13451  deceiving             1
  13452  crescendos            1
  13453  vercingetorix         1
  13454  coils                 1
  13455  apprehended           1
  13456  embed                 1
  13457  laid-out              1
  13458  chartreuse            1
  13459  resolute              1
```

Discussion and take-aways about dictionaries

Discussion and take-aways about dictionaries

- Natural fit for this kind of histogram and much more.

Discussion and take-aways about dictionaries

- Natural fit for this kind of histogram and much more.
- Text files are cool.

Discussion and take-aways about dictionaries

- Natural fit for this kind of histogram and much more.
- Text files are cool.
- Did Proust really use the word email? How do we improve the program?

Discussion and take-aways about dictionaries

- Natural fit for this kind of histogram and much more.
- Text files are cool.
- Did Proust really use the word email? How do we improve the program?
- Discussion.

Outline

Grand challenges for programming language design

Terminology

Attitude toward terminology Suspend one's uncertainty.

Interpreter Slow and flexible.

Compiler Fast: compiles to machine code. And what is that machine code, with its fabled ones and zeros? See ▸ Machine language – 6502

Controlling complexity of large programs

Cutoff at about 100 tounsand lines of code.

Performance

Language features are related to how well you can optimize.

Memory safety

Avoiding memory corruption while keeping high performance.

Outline

The story of programming languages

From https://www.scriptol.com/programming/chronology.php

The story of programming languages

From https://www.scriptol.com/programming/chronology.php

Prehistory

1840 Analytical Engine (Charles Babbage and Ada Lovelace)

1943 ENIAC coding system

1947-1949 Assembly language

1955 FLOW-MATIC (Grace Hopper)

The story of programming languages

From https://www.scriptol.com/programming/chronology.php

Prehistory

1840 Analytical Engine (Charles Babbage and Ada Lovelace)

1943 ENIAC coding system

1947-1949 Assembly language

1955 FLOW-MATIC (Grace Hopper)

The 1950s

1957 FORTRAN (John Backus)

1958 LISP (John McCarthy)

1959 COBOL (CODASYL group)

The story of programming languages

From `https://www.scriptol.com/programming/chronology.php`

Prehistory

1840 Analytical Engine (Charles Babbage and Ada Lovelace)

1943 ENIAC coding system

1947-1949 Assembly language

1955 FLOW-MATIC (Grace Hopper)

The 1950s

1957 FORTRAN (John Backus)

1958 LISP (John McCarthy)

1959 COBOL (CODASYL group)

The 1960s

1960 ALGOL 60

1962 APL

1964 BASIC

1964 Simula

1969 PL/1, B

The story of programming languages

From https://www.scriptol.com/programming/chronology.php

Prehistory

1840 Analytical Engine (Charles Babbage and Ada Lovelace)

1943 ENIAC coding system

1947-1949 Assembly language

1955 FLOW-MATIC (Grace Hopper)

The 1950s

1957 FORTRAN (John Backus)

1958 LISP (John McCarthy)

1959 COBOL (CODASYL group)

The 1960s

1960 ALGOL 60

1962 APL

1964 BASIC

1964 Simula

1969 PL/1, B

The 1970s

1970 Pascal

1972 C

1973 FORTH, ML

1975 Scheme

1977 Bourne shell

The story of programming languages

From `https://www.scriptol.com/programming/chronology.php`

Prehistory

- 1840 Analytical Engine (Charles Babbage and Ada Lovelace)
- 1943 ENIAC coding system
- 1947-1949 Assembly language
- 1955 FLOW-MATIC (Grace Hopper)

The 1950s

- 1957 FORTRAN (John Backus)
- 1958 LISP (John McCarthy)
- 1959 COBOL (CODASYL group)

The 1960s

- 1960 ALGOL 60
- 1962 APL
- 1964 BASIC
- 1964 Simula
- 1969 PL/1, B

The 1970s

- 1970 Pascal
- 1972 C
- 1973 FORTH, ML
- 1975 Scheme
- 1977 Bourne shell

The 1980s

- 1980 Smalltalk
- 1983 Ada
- 1985 Postscript, C++
- 1987 Perl
- 1988 Tcl

The story of programming languages

From `https://www.scriptol.com/programming/chronology.php`

Prehistory

- 1840 Analytical Engine (Charles Babbage and Ada Lovelace)
- 1943 ENIAC coding system
- 1947-1949 Assembly language
- 1955 FLOW-MATIC (Grace Hopper)

The 1950s

- 1957 FORTRAN (John Backus)
- 1958 LISP (John McCarthy)
- 1959 COBOL (CODASYL group)

The 1960s

- 1960 ALGOL 60
- 1962 APL
- 1964 BASIC
- 1964 Simula
- 1969 PL/1, B

The 1970s

- 1970 Pascal
- 1972 C
- 1973 FORTH, ML
- 1975 Scheme
- 1977 Bourne shell

The 1980s

- 1980 Smalltalk
- 1983 Ada
- 1985 Postscript, C++
- 1987 Perl
- 1988 Tcl

The 1990s

- 1990 Haskell
- 1991 Python
- 1995 Java, javascript, Ruby, PHP

The story of programming languages

From https://www.scriptol.com/programming/chronology.php

Prehistory

1840 Analytical Engine (Charles Babbage and Ada Lovelace)

1943 ENIAC coding system

1947-1949 Assembly language

1955 FLOW-MATIC (Grace Hopper)

The 1950s

1957 FORTRAN (John Backus)

1958 LISP (John McCarthy)

1959 COBOL (CODASYL group)

The 1960s

1960 ALGOL 60

1962 APL

1964 BASIC

1964 Simula

1969 PL/1, B

The 1970s

1970 Pascal

1972 C

1973 FORTH, ML

1975 Scheme

1977 Bourne shell

The 1980s

1980 Smalltalk

1983 Ada

1985 Postscript, C++

1987 Perl

1988 Tcl

The 1990s

1990 Haskell

1991 Python

1995 Java, javascript, Ruby, PHP

The "aughts"

2000 C#

2004 Scala

2006 Rust

2007 Scratch

2009 Go

The story of programming languages

From https://www.scriptol.com/programming/chronology.php

Prehistory

1840 Analytical Engine (Charles Babbage and Ada Lovelace)

1943 ENIAC coding system

1947-1949 Assembly language

1955 FLOW-MATIC (Grace Hopper)

The 1950s

1957 FORTRAN (John Backus)

1958 LISP (John McCarthy)

1959 COBOL (CODASYL group)

The 1960s

1960 ALGOL 60

1962 APL

1964 BASIC

1964 Simula

1969 PL/1, B

The 1970s

1970 Pascal

1972 C

1973 FORTH, ML

1975 Scheme

1977 Bourne shell

The 1980s

1980 Smalltalk

1983 Ada

1985 Postscript, C++

1987 Perl

1988 Tcl

The 1990s

1990 Haskell

1991 Python

1995 Java, javascript, Ruby, PHP

The "aughts"

2000 C#

2004 Scala

2006 Rust

2007 Scratch

2009 Go

The 2010s

2010 Julia

2012 Kotlin

2017 WebAssembly

The story of programming languages

From `https://www.scriptol.com/programming/chronology.php`

Prehistory

1840 Analytical Engine (Charles Babbage and Ada Lovelace)

1943 ENIAC coding system

1947-1949 Assembly language

1955 FLOW-MATIC (Grace Hopper)

The 1950s

1957 FORTRAN (John Backus)

1958 LISP (John McCarthy)

1959 COBOL (CODASYL group)

The 1960s

1960 ALGOL 60

1962 APL

1964 BASIC

1964 Simula

1969 PL/1, B

The 1970s

1970 Pascal

1972 C

1973 FORTH, ML

1975 Scheme

1977 Bourne shell

The 1980s

1980 Smalltalk

1983 Ada

1985 Postscript, C++

1987 Perl

1988 Tcl

The 1990s

1990 Haskell

1991 Python

1995 Java, javascript, Ruby, PHP

The "aughts"

2000 C#

2004 Scala

2006 Rust

2007 Scratch

2009 Go

The 2010s

2010 Julia

2012 Kotlin

2017 WebAssembly

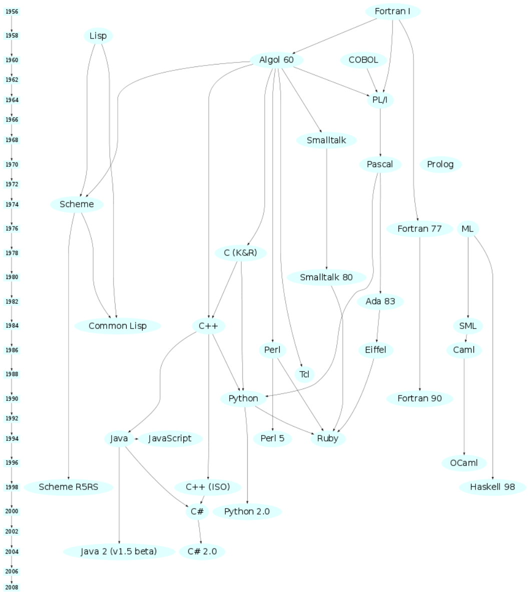
The future (created by Santa Fe youngsters)

2027 greenchile

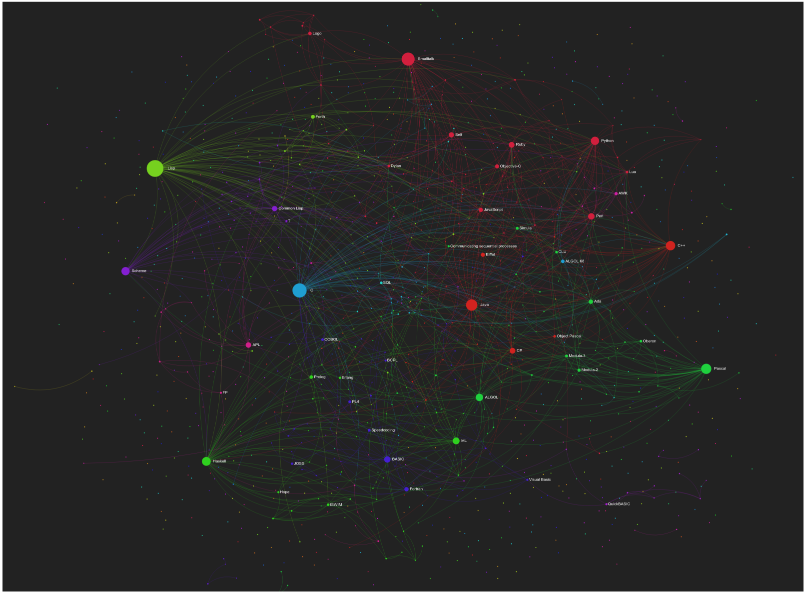
2030 joemama

2032 updog

The story of programming languages – timeline



The story of programming languages – influence



Outline

What is Object Oriented Programming (OOP)?

Objects vs. messages

What is Object Oriented Programming (OOP)?

Objects vs. messages

- Alan Kay coins the term "object-oriented programming" and invents the ultra-OOP language Smalltalk.

What is Object Oriented Programming (OOP)?

Objects vs. messages

- Alan Kay coins the term “object-oriented programming” and invents the ultra-OOP language Smalltalk.
- “I’m sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea.”

What is Object Oriented Programming (OOP)?

Objects vs. messages

- Alan Kay coins the term “object-oriented programming” and invents the ultra-OOP language Smalltalk.
- “I’m sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea.”
- “The big idea is "messaging" – that is what the kernal of Smalltalk/Squeak is all about (and it’s something that was never quite completed in our Xerox PARC phase). The Japanese have a small word – ma – for "that which is in between" – perhaps the nearest English equivalent is ’interstitial’.”

What is Object Oriented Programming (OOP)?

Objects vs. messages

- Alan Kay coins the term “object-oriented programming” and invents the ultra-OOP language Smalltalk.
- “I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea.”
- “The big idea is "messaging" – that is what the kernal of Smalltalk/Squeak is all about (and it's something that was never quite completed in our Xerox PARC phase). The Japanese have a small word – ma – for "that which is in between" – perhaps the nearest English equivalent is 'interstitial'.”
- Inspired by Kay's previous experience in cell biology.

Classes

- Python is an object oriented programming language.
- Almost everything in Python is an object, with its properties and methods.
- A Class is like an object constructor, or a “blueprint” for creating objects.

Make a class with:

```
>>> class MyClass:
>>>     x = 5
```

Then create an object from that class with:

```
>>> p1 = MyClass()
>>> print(p1.x)
```

The person description with a class

Defining the class

```
class Person:
    def __init__(self, name, birth_year, SSN, phone):
        self.name = name
        self.birth_year = birth_year
        self.SSN = SSN
        self.phone = phone

pb = Person('Boyd', 1971, '543-81-5481',
            '+1-606-555-6173')

print(pb.name)
print(pb.birth_year)
```

Adding methods

```
class Person:
    def __init__(self, name, surname, birth_year, SSN
      , phone):
        self.name = name
        self.surname = surname
        self.birth_year = birth_year
        self.SSN = SSN
        self.phone = phone
    def example_function(self):
        print('this is an example function for dude',
              self.name)

pb = Person('Boyd', 1971, '543-81-5481',
            '+1-606-555-6173')

print(pb.name)
print(pb.birth_year)
pb.example_function()
```

More methods

Represent yourself as a string

Put this code in a `person-oop.py` file and run it:

```
class Person:
    def __init__(self, name, surname, birth_year, SSN, phone):
        self.name = name
        self.surname = surname
        self.birth_year = birth_year
        self.SSN = SSN
        self.phone = phone
    def __str__(self):
        return ('name: %s\nsurname: %s\nborn: %d\nSSN: %s\nphone: %s\n'
                % (self.name, self.surname, self.birth_year, self.SSN,
     self.phone))
pb = Person('Boyd', 'Crowder', 1971,
            '543-81-5481', '+1-606-555-6173')
print(pb)                   # note the magic of the __str__() method
```

Check the `__str__()` method

```
$ python3 person-oop.py
name: Boyd
born: 1971
SSN: 543-81-5481
phone: +1-606-555-6173
```