CS 320: Principles of Programming Languages

Mark P Jones and Andrew Tolmach
Portland State University

Spring 2019

Week 5: Paradigms, Scripting, and Python

language = { syntax / semantics }

syntax: the written/spoken/symbolic/physical form; how things are communicated

language = { syntax / semantics }

semantics: what the syntax means or represents

The rest of the course: studying semantics

- Types: classifying values (last week)
  Building blocks for programming language type systems
- Programming language paradigms:
  Procedural, functional, object-oriented, logic, concurrent, …
- Static semantics, program analysis, and type checking
  Fundamentals of static analysis; static vs dynamic typing
- Formalizing programming language semantics
  Denotational, operational, and axiomatics semantics
- Proving program correctness
  Preconditions, invariants, and postconditions

Programming language paradigms

Notions of computation

- Programs are (usually textual) descriptions of computations
- But what, exactly, do we mean by "a computation"?
  Calculating the value of an expression?
  Executing a sequence of commands/instructions?
  Applying logical deduction to reach a conclusion?
  Simulating interacting agents to obtain a result?
  Biological/chemical/physical processes?
- Different notions of computing lead to different paradigms:
  Different ways of thinking about computation
  Different ways of capturing those ideas as programs
  Different tools for solving problems

Multi-paradigm programming

- The lessons that you learn from studying a new language or paradigm can often be adapted to other settings
- If you want to be a better programmer, learn a new language or a new paradigm …
- If you want to design a better programming language, draw inspiration from multiple paradigms …

Example: calculating factorials

Your task: print the factorial of a number n

imperative

```
int r = 1;
int i = 1;
while (i<=n) {
  r = r*i;
  i = i+1;
}
print r;
```

procedural

```
int fact(int i) {
 if (i<2) {
  return i;
 } else {
  return
    i*fact(i-1);
 }
}

print fact(n);
```

functional

```
product [1..n]
```

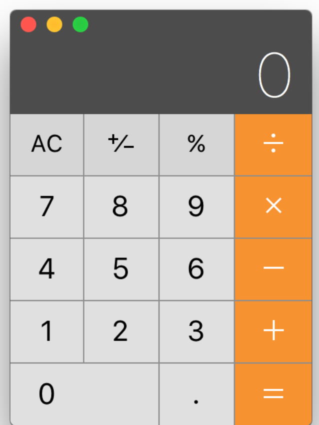
a C programmer might not consider this practical…

… but it opens new paths to solving bigger problems

Computing by calculating

- An old dictionary definition for "computer":

  "a person who makes calculations, especially with a calculating machine"
- A basis for functional programming:
  - Using a wide range of types of values, from numbers and strings to tuples, lists, trees, functions, and more
  - Scaling to large problems



Basic calculator

Good for:

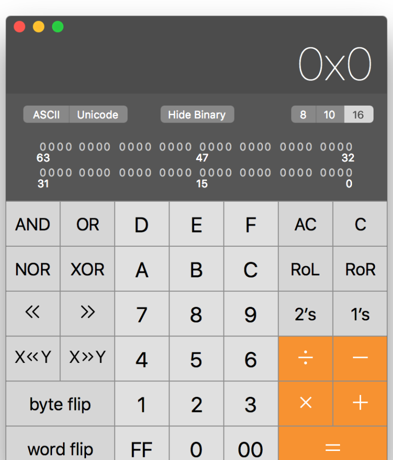
… doing calculations

… with numbers

… and basic arithmetic

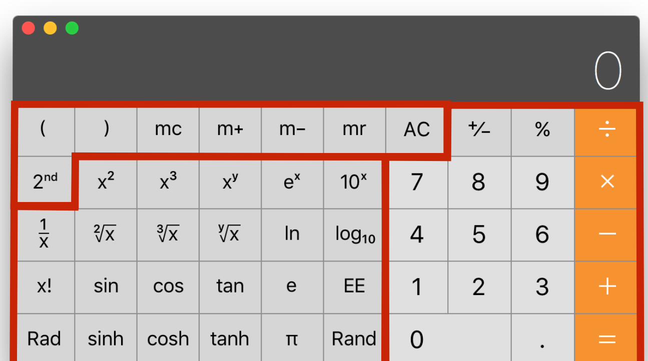
… on small sets of data

Can we do more?

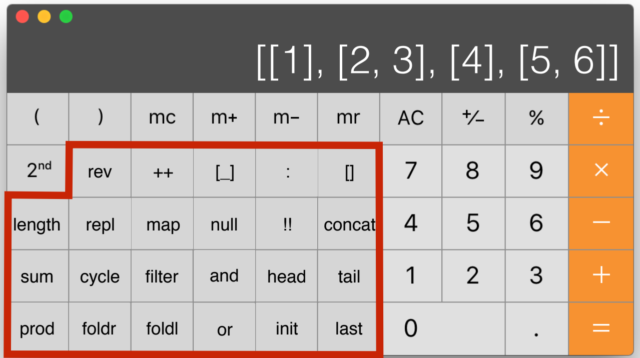


Programmer's calculator

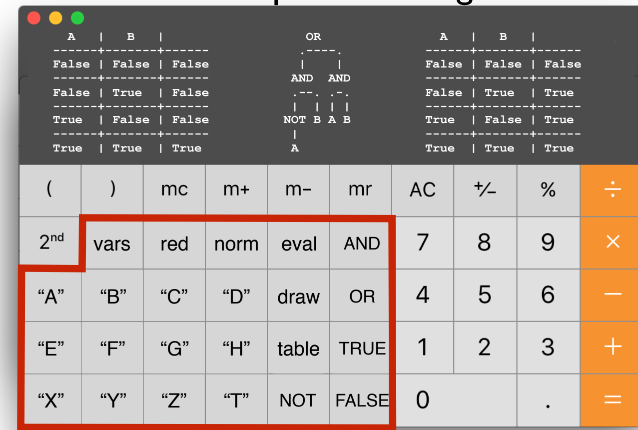
Scientific calculator



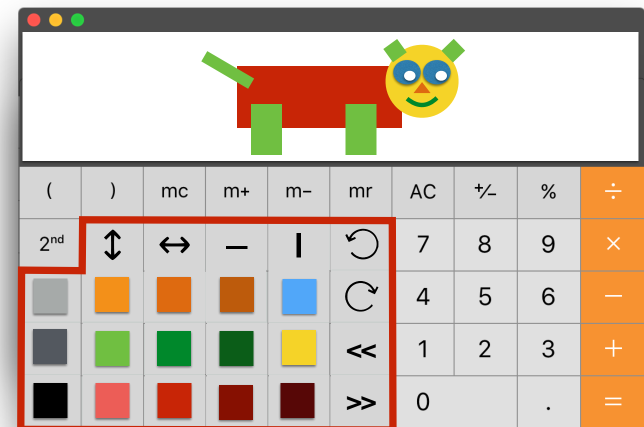
List calculator



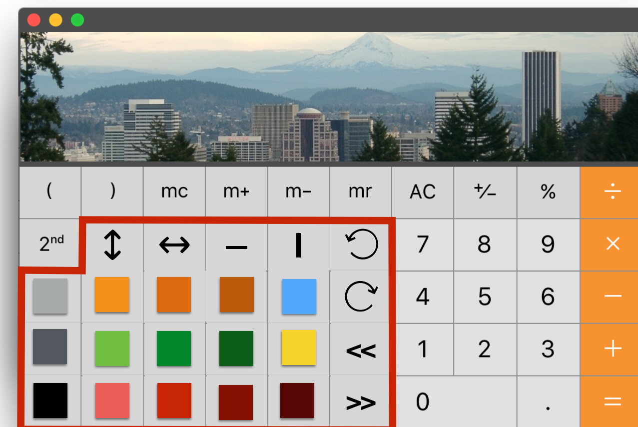
Propositional logic calculator



Picture calculator

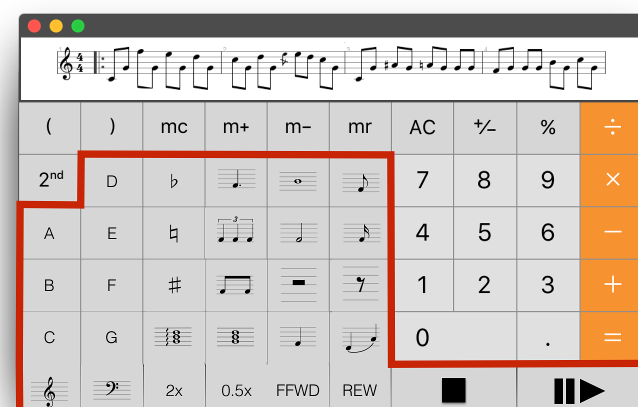


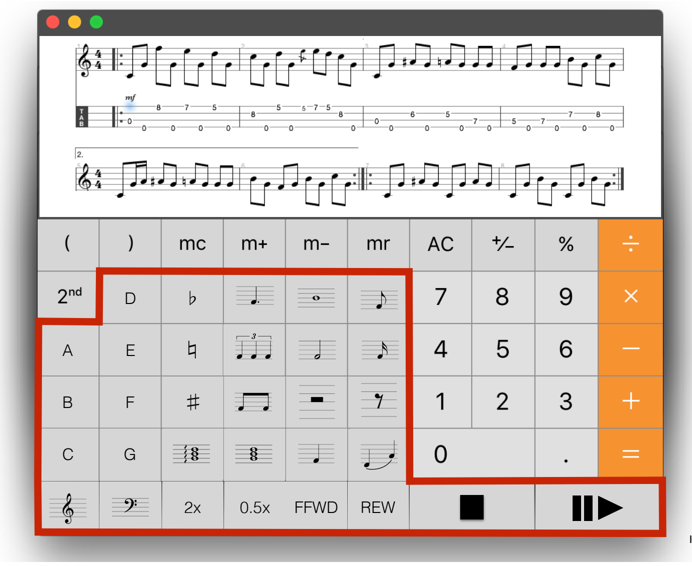
Picture calculator



Music calculator

Music calculator





Computing by calculating

- Basics:
  - Start with some primitive values
  - Add operations for manipulating/combining values
- Multiple types of data:
  - Numbers, Prop, lists, trees, grammars, pictures, music, …
- Extensible:
  - Add new types
  - Add new operations
- Scalable:
  - Work with arbitrarily large values/data structures

Functional language characteristics

- Computation is performed by evaluating expressions
- Functions as **first class values** that can be stored in data structures, passed as arguments, and returned as results
- Pure functional languages:
  - No global variables, no assignment, no side effects
  - Functions as in math (same inputs $\Longrightarrow$ same outputs)
  - How is it possible to do work without side-effects?
    - Higher-level coding: `fact n = product [1..n]`
    - Explicit state: `lexProp ('(':cs) = TOPEN:lexProp cs`

Functional programming languages

- **Lisp** (John McCarthy, 1958)
- **Scheme** (Guy Steele, Gerald Jay Sussman, 1970)
- **ML** (Robin Milner, others, 1973)
- **Haskell** (Committee, 1990)
- **OCaml** (INRIA, 1996)
- **Scala** (Martin Odersky, 2004)
- **F#** (Don Syme, Microsoft, 2005)
- **Clojure** (Rich Hickey, 2007)
- and numerous other examples (Hope, Erlang, Racket, …)

a rich history of functional programming languages!

Imperative and Procedural Programming

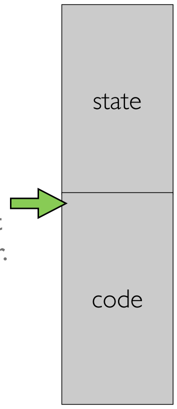
Computing by executing

- A more recent dictionary definition for "computer":

  "an electronic device for storing and processing data, typically in binary form, according to instructions given to it in a variable program."
- A basis for modern computing equipment:
  - programs are described by sequences of primitive machine instructions that are represented by binary data stored in an electronic memory
  - the processor executes the program by decoding and acting on the instructions, one after another

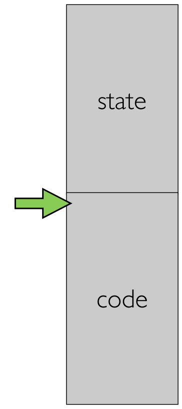
Imperative programming

- Programs specify:
  - the data/state that will be manipulated
  - the instructions that make up the code
- The start of the program is known as the **entry point** (e.g., "main")
- Programs run by starting at the entry point and running commands, one after the other. (Hence "imperative")
- The state will typically change as the program runs



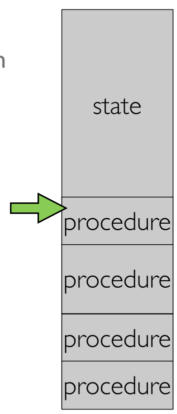
Initialization

- The state must typically be set to some appropriate initial value before the program is executed
- This can be handled by:
  - Loading initial state values in to memory with the rest of the program; or
  - Inserting extra code at the entry point to calculate and save appropriate initial values in the state
- Of course, these techniques can be used in combination too …



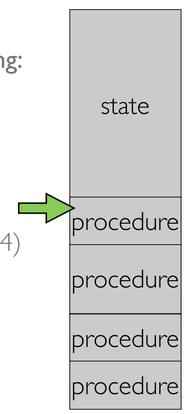
Procedural programming

- A form of imperative programming in which the code is structured as a collection of procedure/subroutine definitions
- Allows abstraction, reuse, and modular construction of software
- One particular procedure is chosen as the **entry point** (e.g., "main")
- Programs run by executing the entry point procedure
- Procedures "call" other procedures in a LIFO (i.e., stack-like) manner



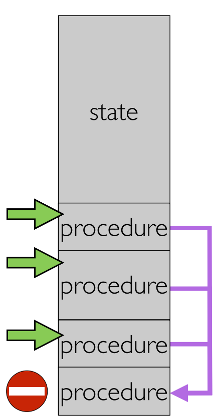
Procedural programming languages

- Many programming languages have been designed to support this approach, including:
  - **Fortran** (John Backus, 1957)
  - **COBOL** (Howard Bromberg, Howard Discount, Vernon Reeves, Jean Sammet, William Selden, Gertrude Tierney, 1959)
  - **BASIC** (John Kemeny, Thomas Kurtz, 1964)
  - **Pascal** (Niklaus Wirth, 1970)
  - **C** (Dennis Ritchie, 1972)
  - **Ada** (Jean Ichbiah, 1983)
  - …



Procedure libraries

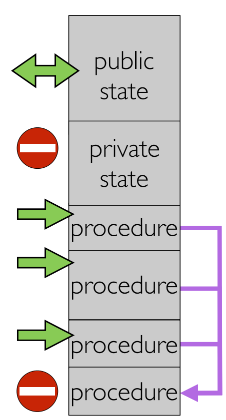
- A **library** is a program with multiple entry points, designed to provide services to some other program (a **client**)
- More opportunities for abstraction, reuse, and modular construction
- Library code is executed only when invoked by the client
- The procedures that are provided to clients define an **interface** to the library
- Some procedures may be for internal use only, excluded from the interface



Public and private state

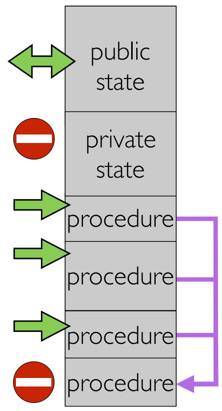
- It is common to partition the state:
- Some parts may be directly accessible to clients (the "**public**" state)
- Some parts may be intended for library/internal use only (the "**private**" state)
- Limiting access to private state helps to isolate bugs and promote modularity:

  If the private state is invalid/inconsistent, this must be the result of a bug in the library itself, *not* the client

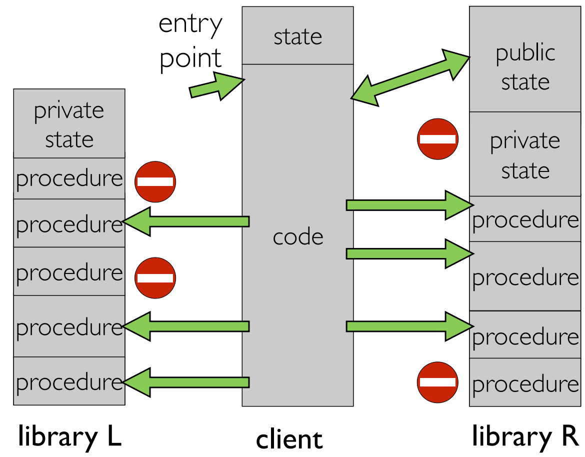


Encapsulated state

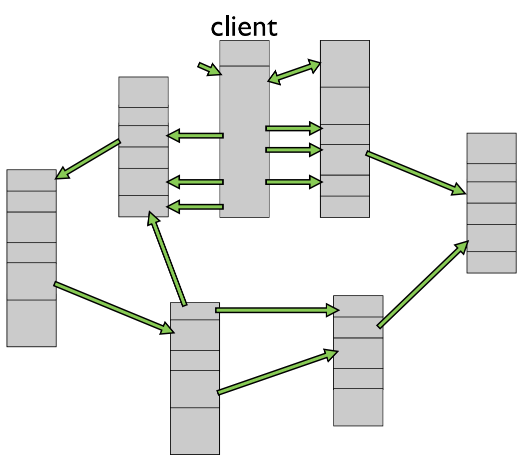
- How can we ensure that state is truly "**encapsulated**" (i.e., that clients do not have access to private state or internal procedures?)
  - Option 1: Rely on programmer discipline (error prone)
  - Option 2: Obfuscate the private components so that they are harder to access
  - Option 3: Use a programming language that enforces these requirements



Using multiple libraries



Using multiple ~~libraries~~ objects



Object-oriented Programming

Object-oriented programming

- Computations are performed by collections of "**objects**" that "**invoke methods**" or "**send messages**" to one another to request a service or deliver a result

- A single program may use multiple forms of object
  ⇒ classes
- Objects store fields
  ⇒ encapsulated state
- New objects can be initialized on demand
  ⇒ constructors
- Every object has an associated set of procedures
  ⇒ methods
- Different classes can share common features
  ⇒ inheritance
- A method may be implemented differently in different classes
  ⇒ dynamic dispatch

Object-oriented programming

- Computations are performed by collections of "**objects**" that "**invoke methods**" or "**send messages**" to one another to request a service or deliver a result

characteristic features of object-oriented languages (not everyone agrees!)

- A single program may u[se] multiple forms of object
  ⇒ classes
- Objects store fields
  ⇒ encapsulated state
- New objects can be initialized on demand
  ⇒ constructors
- ...ssociated
- Different classes can share common features
  ⇒ inheritance
- A method may be implemented differently in different classes
  ⇒ dynamic dispatch

Object-oriented programming languages

- **Simula** (Ole-Johan Dahl and Kristen Nygaard, 1967)
- **Smalltalk** (Alan Kay, Dan Ingalls, Adele Goldberg, 1972)
- **C++** (Bjarne Stroustrup, 1983)
- **Objective-C** (Brad Cox, 1983)
- **Eiffel** (Bertrand Meyer, 1985)
- **Java** (James Gosling, 1995)
- **C#** (Microsoft, 2000)
- and numerous other examples (JavaScript, OCaml, Python, Ruby, Self, Scala, Swift, …)

many different flavors of object-oriented programming!

Scripting Languages

Scripting languages

- Languages designed to compose and coordinate computations rather than to code computations directly
- Sometimes called "glue" languages, because they are used to glue together existing existing programs/tools/libraries/…

Conventional Language	→	Scripting Language
Efficiency		Flexibility
Maintainability		Rapid Development
Portability		Local Customization
Static Checking		Run-time checking
Low-level types		High-level types

Evolution of scripting languages

unix sh
JCL
powershell
applescript

Command shells

```
#!/bin/sh
for i
do
     d=`dirname $i`
     f=`basename $i .fab`
     echo $f:
     if [ -r $d/$f.in ]; then
         java -classpath .:../0 \
           IRInterpDriver $d/$f.fab $d/$f.ir \
            < $d/$f.in > tst/$f.out 2> tst/$f.err
        else
         java -classpath .:../0 \
           IRInterpDriver $d/$f.fab $d/$f.ir \
           > tst/$f.out 2> tst/$f.err
     fi;
     if [ -r tst/$f.out.bak ]; then
         diff tst/$f.out.bak tst/$f.out;
       else cp tst/$f.out tst/$f.out.bak; fi
     if [ -r tst/$f.err.bak ]; then
         diff tst/$f.err.bak tst/$f.err;
       else cp tst/$f.err tst/$f.err.bak; fi
done
exit 0
```

bash

Evolution of scripting languages

unix sed, awk
emacs lisp
RPG

Text editors/generators

```
/<[hH][123]>/ {
    # execute this block if line contains an opening tag
    do {
        open_tag = match($0, /<[hH][123]>/)
        $0 = substr($0, open_tag)       # delete text before opening tag

        while (!/<\/[hH][123]>/) {      # print interior lines
            print                       # in their entirety
            if (getline != 1) exit
        }
        close_tag = match($0, /<\/[hH][123]>/) + 4

        print substr($0, 0, close_tag)  # print through closing tag
        $0 = substr($0, close_tag + 1)  # delete through closing tag
    } while (/<[hH][123]>/)             # repeat if more opening tags
}
```

awk

Evolution of scripting languages

unix sh
JCL
powershell
applescript

Command shells

unix sed, awk
emacs lisp
RPG

Text editors/generators

perl
tcl
python
ruby

Scripting Languages

php
jsp

server

browser

javascript

Embedded Web Languages

Typical scripting language features

- Interactive use (e.g. Read-Eval-Print-Loop)
- Syntax encourages brevity
- Variables have scope but no declarations
- Dynamic typing
- Strong support for string manipulation and pattern matching
- Direct access to OS system facilities and external libraries
- Built-in support for high-level types
- Interpreted execution

  Often “defined” by single official implementation

Python

- Invented ~1990 by Guido van Rossum
- Now one of the most widely used programming languages in the world
- Python 3.X (2008) is mildly backwards-incompatible version

  Now used for new developments

  But lots of Python 2.X code remains

  We will only use 3.X in this course ...

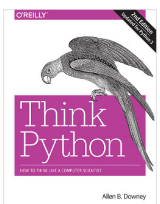
- Our initial focus: procedural programming, but we will use it later for OOP too ...

Python 2.X: print x

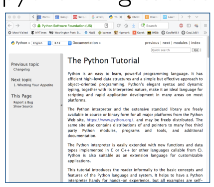
Python 3.X: print(x)

Python resources

- Numerous books and websites (but some for Python 2.X)



https://docs.python.org/3/tutorial/



http://greenteapress.com/wp/think-python-2e/

Prof. Wu-chang Feng's crash course on Python (1 hour video) is available on media.pdx.edu (see link on D2L)

O'Reilly Python books @ humblebundle.com

The Python REPL (Read-Eval-Print-Loop)

```
$ python3
Python 3.7.2 (default, Dec 27 2018, 07:35:45)
[Clang 9.0.0 (clang-900.0.39.2)] on darwin
Type "help", "copyright", "credits" or
"license" for more information.
>>> 2+2
4
>>> x = 15 + 6
>>> y = x * 2
>>> y
42
>>> print('x + y =', x+y)
x + y = 63
>>> ^D
$
```

be sure to specify python3

evaluate an expression

bind a variable to a value

use the value of a variable

display a value

print values of one or more expressions

Python: Batch execution of programs

```
$ cat example.py
# same statements as we typed in before
2+2
x = 15+6
y = x * 2
y
print('x + y =', x+y)
$ python3 example.py
x + y = 63
$
```

comment

file containing python statements

top-level expressions are evaluated invisibly

reads and executes file

print() results go to stdout

Python values and types

- Every value is an object of some class

  Conceptually it lives in a box and is handled by reference

  No distinction between objects and primitive values

- Every value has a type.
- Built-in types include

  primitive types: integers, booleans, floats, strings, ...

  constructed types: lists, tuples, dictionaries, sets, ...

- You can define new classes, but many programs don’t need to

Implicit vs. explicit declarations

- In Python, variables are implicitly declared

  First use of a variable declares it; its type can change:

```
x = 37
y = x + 5
x = 'abc'
```

no declaration needed; created when assigned to

x and y now exist and contain integers

x now contains a string

- Contrast use of explicit variable declarations in Java/C/C++:

  Variables must be named and typed before they are used

```
int x,y;
x = 37;
y = x + 5;
x = "abc";
```

declaration

use

use

static type error

Tradeoffs: Implicit vs. explicit declarations

Explicit declarations:

- More text to type
- Helps prevent typos

Implicit declarations:

- Less text to type
- Easy to mistype variable name
- Scope of variable can be complicated (more later)

```
var = 37
if rare-condition:
  y = vsr + 5
```

only caught when this line actually runs — bug could be lurking for a long time

typo!

Dynamic typing

- Dynamic typing: check type consistency at execution time

  Variables do not themselves have types

  - Often the variables are not even declared

  Each run-time values carries its own type

  Type is checked before use

  Disallowed operations cause run-time exception

- Used in Python, Ruby, Javascript, Lisp/Scheme, …

program may run a long time before error is discovered

```
x = 3
x = "foo"
y = x+1
```

perfectly OK

run-time type error

Static typing

- Static typing: check type consistency before program runs

  Types of all variables and expressions are determined

  Usually, variable types are explicitly declared

  - Sometimes can be inferred by compiler based on use

  Disallowed operations cause compile-time error

- Used in C/C++, Java, C#, Haskell, …

```
int x;
x = 37;
x = "foo";
```

explicit type declaration

not allowed

program doesn't compile

Digging Deeper

How do you learn a new language?

How can you learn the **syntax**, **semantics**, and **idioms** of a new programming language?

- Read a book or tutorial
- Take a class
- Study sample programs
- Talk to other programmers
- Write your own programs
- Practice!

Read the manual!

- Often very technical ...
- Rarely a great work of literature ... 😀
- But an authoritative source of information for understanding key details
- An essential resource for language implementors
- One of the goals of this class is to prepare you for reading this kind of documentation ...

The Python Language Reference
Release 3.7.3

Guido van Rossum
and the Python development team

April 28, 2019

Python Software Foundation
Email: docs@python.org

The Python Language Reference

This reference manual describes the syntax and "core semantics" of the language. It is terse, but attempts to be exact and complete. The semantics of non-essential built-in object types and of the built-in functions and modules are described in The Python Standard Library. For an informal introduction to the language, see The Python Tutorial. For C or C++ programmers, two additional manuals exist: Extending and Embedding the Python Interpreter describes the high-level picture of how to write a Python extension module, and the Python/C API Reference Manual describes the interfaces available to C/C++ programmers in detail.

1. Introduction

This reference manual describes the Python programming language. It is not intended as a tutorial.

While I am trying to be as precise as possible, I chose to use English rather than formal specifications for everything except syntax and lexical analysis. This should make the document more understandable to the average reader, but will leave room for ambiguities. Consequently, if you were coming from Mars and tried to re-implement Python from this document alone, you might have to guess things and in fact you would probably end up implementing quite a different language. On the other hand, if you are using Python and wonder what the precise rules about a particular area of the language are, you should definitely be able to find them here. If you would like to see a more formal definition of the language, maybe you could volunteer your time — or invent a cloning machine :-).

1.2. Notation

The descriptions of lexical analysis and syntax use a modified B
following style of definition:

```
name      ::=  lc_letter (lc_letter | "_")*
lc_letter ::=  "a"..."z"
```

The first line says that a `name` is an `lc_letter` followed by a sequence of zero or more `_letters`
and underscores. An `lc_letter` in turn is any of the single ch
actually adhered to for the names defined in lexical and gramm

Each rule begins with a name (which is the name defined by
used to separate alternatives; it is the least binding operator in
more repetitions of the preceding item; likewise, a plus (+)
phrase enclosed in square brackets ([ ]) means zero or one oc
phrase is optional). The * and + operators bind as tightly
grouping. Literal strings are enclosed in quotes. White space
Rules are normally contained on a single line; rules with many
tively with each line after the first beginning with a vertical bar.

"Backus-Naur Form" (used by John Backus and Peter Naur in the definition of ALGOL 60)

Extended BNF:

```
"tokens"
alt1 | alt2
zero_or_more*
one_or_more+
[optional]
(grouping)
```


2. Lexical analysis

A Python program is read by a *parser*. Input to the parser is a stream of *tokens*, generated by the *lexical analyzer*. This chapter describes how the lexical analyzer breaks a file into tokens.

Python reads program text as Unicode code points; the encoding of a source file can be given by an encoding declaration and defaults to UTF-8, see PEP 3120 for details. If the source file cannot be decoded, a `SyntaxError` is raised.

2.3.1. Keywords

The following identifiers are used as reserved words, or *keywords* of the language, and cannot be used as ordinary identifiers. They must be spelled exactly as written here:

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```


2.4.5. Integer literals

Integer literals are described by the following lexical definitions:

```
integer      ::=  decinteger | bininteger | octinteger | hexinteger
decinteger   ::=  nonzerodigit (["_"] digit)* | "0"+ (["_"] "0")*
bininteger   ::=  "0" ("b" | "B") (["_"] bindigit)+
octinteger   ::=  "0" ("o" | "O") (["_"] octdigit)+
hexinteger   ::=  "0" ("x" | "X") (["_"] hexdigit)+
nonzerodigit ::=  "1"..."9"
digit        ::=  "0"..."9"
bindigit     ::=  "0" | "1"
octdigit     ::=  "0"..."7"
hexdigit     ::=  digit | "a"..."f" | "A"..."F"
```

There is no limit for the length of integer literals apart from what can be stored in available memory.

Underscores are ignored for determining the numeric value of the literal. They can be used to group digits for enhanced readability. One underscore can occur between digits, and after base specifiers like `0x`.

Note that leading zeros in a non-zero decimal number are not allowed. This is for disambiguation with C-style octal literals, which Python used before version 3.0.

Some examples of integer literals:

```
7     2147483647                        0o177    0b100110111
3     79228162514264337593543950336     0o377    0xdeadbeef
      100_000_000_000                   0b_1110_0101
```


2.4.6. Floating point literals

Floating point literals are described by the following lexical definitions:

```
floatnumber   ::=  pointfloat | exponentfloat
pointfloat    ::=  [digitpart] fraction | digitpart "."
exponentfloat ::=  (digitpart | pointfloat) exponent
digitpart     ::=  digit (["_"] digit)*
fraction      ::=  "." digitpart
exponent      ::=  ("e" | "E") ["+" | "-"] digitpart
```

Note that the integer and exponent parts are always interpreted using radix 10. For example, `077e010` is legal, and denotes the same number as `77e10`. The allowed range of floating point literals is implementation-dependent. As in integer literals, underscores are supported for digit grouping.

Some examples of floating point literals:

```
3.14    10.    .001    1e100    3.14e-10    0e0    3.14_15_93
```

2.5. Operators

The following tokens are operators:

```
+       -       *       **      /       //      %      @
<<      >>      &       |       ^       ~
<       >       <=      >=      ==      !=
```

2.6. Delimiters

The following tokens serve as delimiters in the grammar:

```
(       )       [       ]       {       }
,       :       .       ;       @       =       ->
+=      -=      *=      /=      //=     %=      @=
&=      |=      ^=      >>=     <<=     **=
```

The period can also occur in floating-point and imaginary literals. A sequence of three periods has a special meaning as an ellipsis literal. The second half of the list, the augmented assignment operators, serve lexically as delimiters, but also perform an operation.

The following printing ASCII characters have special meaning as part of other tokens or are otherwise significant to the lexical analyzer:

```
'       "       #       \
```

The following printing ASCII characters are not used in Python. Their occurrence outside string literals and comments is an unconditional error:

```
$       ?       `
```


6.9. Binary bitwise operations

Each of the three bitwise operations has a different priority level:

```
and_expr ::=  shift_expr | and_expr "&" shift_expr
xor_expr ::=  and_expr | xor_expr "^" and_expr
or_expr  ::=  xor_expr | or_expr "|" xor_expr
```

The `&` operator yields the bitwise AND of its uments, which must be integers.

What are the relative precedences of `|`, `^`, and `&`?

Do these operations group to the left or to the right?

Example: a typical `or_expr` takes the form

```
((xor_expr "|" xor_expr) "|" xor_expr) "|" xor_expr
```

So we can see that `"|"`:

1. groups to the left; and
2. has lower precedence that xor.

6.11. Boolean operations

```
or_test  ::=  and_test | or_test "or" and_test
and_test ::=  not_test | and_test "and" not_test
not_test ::=  comparison | "not" not_test
```

similar question here ...

6.10. Comparisons

Unlike C, all comparison operations in Python have the same priority, which is lower than that of any arithmetic, shifting or bitwise operation. Also unlike C, expressions like `a < b < c` have the interpretation that is conventional in mathematics:

```
comparison    ::=  or_expr (comp_operator or_expr)*
comp_operator ::=  "<" | ">" | "==" | ">=" | "<=" | "!="
                   | "is" ["not"] | ["not"] "in"
```

Comparisons yield boolean values: `True` or `False`.

Comparisons can be chained arbitrarily, e.g., that `y` is evaluated only once (but in both ca false).

unusual chaining syntax:

$e_1 < e_2 < e_3 < e_4 < e_5$

6.12. Conditional expressions

```
conditional_expression ::=  or_test ["if" or_test "else" expression]
expression             ::=  conditional_expression | lambda_expr
expression_nocond      ::=  or_test | lambda_expr_nocond
```

Conditional expressions (sometimes called a "terna operator") have the lowest priority of all Python operations.

also notable: condition in the middle

`x if x>=y else y`

how do we parse an expression like:

e_1 `if` c_1 `else` e_2 `if` c_2 `else` e_3

left grouping:

(e_1 `if` c_1 `else` e_2) `if` c_2 `else` e_3

✗

right grouping:

e_1 `if` c_1 `else` (e_2 `if` c_2 `else` e_3)

✓

3.2. The standard type hierarchy

Below is a list of the types that are built into Python. Extension modules (written in C, Java, or other languages, depending on the implementation) can define additional types. Future versions of Python may add types to the type hierarchy (e.g., rational numbers, efficiently stored arrays of integers, etc.), although such additions will often be provided via the standard library instead.

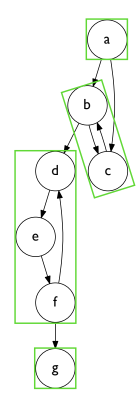
- None
- Numbers
  - Integral
    - int
    - bool
  - real
  - complex
- Sequences
  - Immutable sequences
    - Strings
    - Tuples
    - Bytes
  - Mutable sequences
    - Lists
    - Byte arrays
- Set types
  - Sets (mutable)
  - Frozen sets (immutable)
- Mappings
  - Dictionaries
- Callable types
  - User-defined functions
  - Instance methods
  - Generator functions
  - …
  - Classes
- Modules
- Class instances
- I/O objects
- …

Case study: Strongly connected components

- Python's high-level collection types (lists, sets, dictionaries) and concise syntax make it easy to implement standard algorithms; the results almost look like pseudo code!
- In a directed graph:

  A **strongly connected component** (scc) is a set of vertices in which you can move between any pair of elements by following a path in the graph

- Many practical applications, including numerous examples in programming language analysis and implementation



```python
graph = {'d' : {'e'},
         'a' : {'b', 'c'},
         'e' : {'f'},
         'c' : {'b'},
         'f' : {'g', 'd'},
         'b' : {'c', 'd'},
         'g' : set() }
```

dictionary

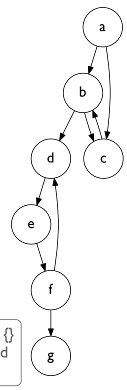
values: sets (of neighbors)

"adjacency list" representation

keys: vertex labels

empty set

To avoid *ambiguity*, we cannot use {} for both the empty dictionary and the empty set…




```python
graph.keys() = {'d', 'a', 'e', 'c', 'f', 'b', 'g'}
```


unvisited

visited

```python
def dfs(graph):
    visited  = set()

    def visit(v):
        if v not in visited:
            visited.add(v)
            for w in graph[v]:
                visit(w)

    for v in graph.keys():
        visit(v)
```

function definition

parameter

local variable (no explicit declaration)

conditional

sets are mutable

nested function definition

loop

recursive call

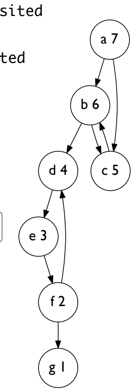
dictionary lookup

loop

indentation determines program structure

procedure, executed for effect, no return values

{'d', 'a', 'e', 'c', 'f', 'b', 'g'}



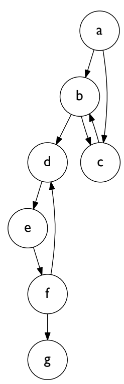

```python
def dfs(graph):
    visited  = set()
    finished = []
    def visit(v):
        if v not in visited:
            visited.add(v)
            for w in graph[v]:
                visit(w)
            finished.append(v)
    for v in graph.keys():
        visit(v)
    return finished
```

list — order matters

lists are mutable

function return value

```python
finished = ['g', 'f', 'e', 'd', 'c', 'b', 'a']
```



flipped

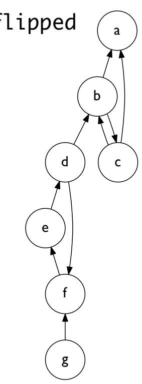
```python
def flipGraph(graph):
    flipped = {}

    for v in graph.keys():
        flipped[v] = set()

    for v in graph.keys():
        for w in graph[v]:
            flipped[w].add(v)

    return flipped
```

empty dictionary

dictionaries are mutable

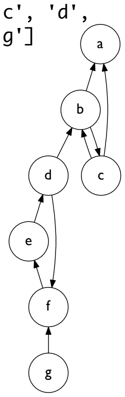
sets are mutable

new graph, original is unchanged



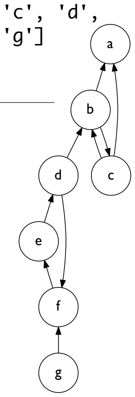

```python
def scc(graph):
    finished = dfs(graph)
    finished.reverse()
```

['a', 'b', 'c', 'd', 'e', 'f', 'g']



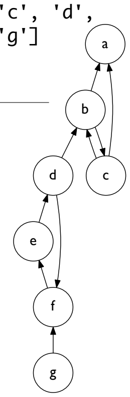

```python
def scc(graph):
    finished = dfs(graph)
    finished.reverse()
    flipped  = flipGraph(graph)
```

['a', 'b', 'c', 'd', 'e', 'f', 'g']



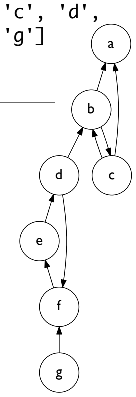
```
def scc(graph):
    finished = dfs(graph)
    finished.reverse()
    flipped  = flipGraph(graph)
    visited  = set()
```

['a', 'b', 'c', 'd',
 'e', 'f', 'g']



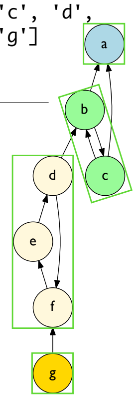

```
def scc(graph):
    finished = dfs(graph)
    finished.reverse()
    flipped  = flipGraph(graph)
    visited  = set()
    def visit(v, scc):
        visited.add(v)
        scc.append(v)
        for w in flipped[v]:
            if w not in visited:
                visit(w, scc)
```

['a', 'b', 'c', 'd',
 'e', 'f', 'g']




```
def scc(graph):
    finished = dfs(graph)
    finished.reverse()
    flipped  = flipGraph(graph)
    visited  = set()
    def visit(v, scc):
        visited.add(v)
        scc.append(v)
        for w in flipped[v]:
            if w not in visited:
                visit(w, scc)
    sccs = []
    for v in finished:
        if v not in visited:
            scc = []
            visit(v, scc)
            sccs.append(scc)
   return sccs
```

['a', 'b', 'c', 'd',
 'e', 'f', 'g']




```
sccs = [['a'],
        ['b', ‘c'],
        ['d', 'f', 'e'],
        [‘g']]
```


Dynamic typing is flexible

dictionary set list

```
scc({'a': {'b'}, 'b':['c'], 'c':[] })

    ==>   [['a'], ['b'], [‘c']]
```

- Our implementation works seamlessly with adjacency lists and adjacency sets (and other "sequence" types too)
- These can be arbitrarily mixed together in a single dictionary representing a graph
- The implementation provides a high degree of polymorphism!

Dynamic typing is dangerous!

dictionary set oops!

```
scc({'a': {'b'}, 'b':3, 'c':[] })
```

```
Traceback (most recent call last):
  File "scc.py", line 83, in <module>
    testScc({'a': {'b'}, 'b':{3}, 'c':[] })
  File "scc.py", line 66, in testScc
    print('are', scc(graph))
  File "scc.py", line 40, in scc
    vertices = dfs(g)
  File "scc.py", line 33, in dfs
    visit(v)
  File "scc.py", line 30, in visit
    visit(w)
  File "scc.py", line 30, in visit
    visit(w)
  File "scc.py", line 29, in visit
    for w in g[v]:
KeyError: 3
```

Errors like this are not caught until the relevant part of the program is executed ...

Summary: Language paradigms

- Languages can be (partially) classified according to paradigms such as functional, procedural, object-oriented, …
- Functional programming emphasizes the evaluation of expressions and the use of functions as first-class values
- Procedural programming views programs as collections of procedures with associated state
- Object-oriented programming views programs as collections of stateful objects that interact by sending messages
- There are no firm boundaries: for example, you can do functional programming in a procedural language
- Many modern language designs are multi-paradigm