Theory of Algorithms

ian.mcloughlin@gmit.ie

Topics

Python

Permutations

Timing Algorithms

Functional Programming

Turing Machines

Complexity Classes

Python

About Python

January 1994 – Python 1.0.0 released.

Guido van Rossum – Designer/Author of Python.

Current versions – 3.5.1 and 2.7.11.

Interpreted – Python implementation must be present at runtime.

Off-side rule – Blocks identified by indentation, as opposed to curly braces.

Popularity – IEEE Spectrum ranks it as the fourth most popular language (July 2015).

Community – Python Enhancement Proposals, notably PEP 8: The Python Style Guide.

spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages

Guido van Rossum

- Started Python as a hobby.
- Worked for Google, half-time spent on Python.
- Now works at Dropbox.
- Benevolent dictator for life (BDFL).

Conditions

```
x = int(raw_input("Please enter an integer: "))
if x < 0:
  x = 0
  print 'Negative changed to zero'
elif x == 0:
  print 'Zero'
elif x == 1:
  print 'Single'
else:
  print 'More'
```

docs.python.org/2/tutorial

Loops

```python
# A for loop.
a = ['Mary', 'had', 'a', 'little', 'lamb']
for i in range(len(a)):
  print(i, a[i])
```

```python
# A while loop.
a, b = 0, 1
while b < 1000:
  print(b)
  a, b = b, a+b
```

docs.python.org/3/tutorial

Functions

```python
# write Fibonacci series up to n
def fib(n):
  """Print a Fibonacci series up to n."""
  a, b = 0, 1
  while a < n:
    print(a)
    a, b = b, a+b
```

docs.python.org/3/tutorial

CPython

Reference implementation – Many different Python implementations exist.

Version 3 – Broke backwards compatibility (somewhat).

Unladen Swallow – Google attempt to fix some Python problems.

Modules – Lots of great Python modules available.

www.python.org

Lists

Lists in Python are usually written as comma-separated values between square brackets.

Types – elements of a list don't have to have the same types.

Slicing is possible, where we take a sublist of the list.

Assignment to slices is possible.

len() is a built-in function that returns the length of a list.

range() is a built-in function that returns a list of numbers. Note: it returns an *iterator*.

```
letters = ['a', 'b', 'c']
letters[1:] = ['c', 'd']
range(10) # [0,1,2,3,4,5,6,7,8,9]
```

docs.python.org/3/tutorial

Strings

Strings are a lot like lists in Python.

Assignment to slices is not allowed, however.

```python
words = "This is a sentence."
words[8]          # a
words[5:7]        # is
words[:7]         # This is
words[10:]        # sentence.
words[17:9:-1]    # ecnetnes

len(words)        # 19
"One" + "Two"     # OneTwo
```

docs.python.org/3/tutorial

Functions

def is the keyword for defining a function.

Parameters can be given defaults, so that they are optional.

```
def axn(x, a=1, n=2):
        return a*(x**n)    # ax^n

axn(3)        # 9
axn(3, 2)     # 18
axn(3, 2, 3)  # 54
axn(3, n=3)   # 27
```

docs.python.org/3/tutorial

List comprehensions

Comprehensions are quick ways of creating lists from other lists.

```
nos = range(5) # [0, 1, 2, 3, 4]
squares = [i*i for i in nos] # [0, 1, 4, 9, 16]
oddsqs = [i*i for i in nos if i % 2 == 1] # [1, 9]
```

docs.python.org/3/tutorial

map()

map() takes a function and a list.

New list – it returns a new generator, which is the original list with the function applied to each element.

```
map(len, words)
list(map(len, words))
```

docs.python.org/3/tutorial

Lambda functions

lambda functions are short, inline functions.

Nameless – lambda functions need not have a name.

```
lambda x: x + n
```

docs.python.org/3/tutorial

Without generators

```python
# Build and return a list
def firstn(n):
    num, nums = 0, []
    while num < n:
        nums.append(num)
        num += 1
    return nums

sum_of_first_n = sum(firstn(1000000))
```

https://wiki.python.org/moin/Generators

With generators

```
# yields items instead of returning a list
def firstn(n):
    num = 0
    while num < n:
        yield num
        num += 1

sum_of_first_n = sum(firstn(1000000))
```

https://wiki.python.org/moin/Generators

Permutations

Permutations

Permutations are rearrangements of ordered collections of items.

Example: "abcd" is a rearrangement of "bacd".

Think of having a four boxes, where we have to place one of the items in each box.

What are all the different ways of doing this?

What are all the different ways of associating items with boxes?

Abstraction

We can consider permutations in abstraction. For instance, if we have four items to rearrange, we can label the first item 1, the second 2, and so on. Then we can represent the various permutations, in terms of the numbers associated with the items. The permutations “abcd” and “bacd” could be represented by:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 3 & 4 \end{pmatrix}$$

In this way we can consider permutations in their own right.

Counting permutations

With four items, how many distinct permutations are there?

Consider having four placeholders where we can place the items:

$$\boxed{?}\ \boxed{?}\ \boxed{?}\ \boxed{?}$$

When we place an item in the first box, we have four choices, then we are left with only three choice for the second, two choices for the third, and one choice (i.e. not a choice at all) for the last. So for there are $4 \times 3 \times 2 \times 1 = 4!$ choices in total.

Counting anagrams

Repitition

What happens when we consider two of our items to be the same? For instance, what if we are looking for all distinct rearrangements of "aacd" as opposed to "abcd"? In that case we need to account for the rearrangements of those items by dividing by the factorial of the number of times each item is repeated: $\frac{4!}{2!}$.

If more than one item is repeated a number of times we just keep dividing by the factorials of the numbers of repititons. The distinct number of rearrangements of "aaabbcd" is $\frac{7!}{3!2!}$.

Exercise

Calculate the number of distinct rearragements of the word "Mississippi".

Heap's algorithm

Heap published an algorithm in November 1963 for generating permutations.

Published in The Computer Journal.

Read the article in the link below – it's an easy read.

Pairs of items are interchanged to generate each new permutation.

Induction is used to show the algorithm works.

comjnl.oxfordjournals.org/content/6/3/293.full.pdf

Heap's algorithm description

- Suppose we know how to permute $(n-1)$ items.
- That is, we know all of the different ways of slotting $(n-1)$ items in $(n-1)$ boxes.
- Let's add another, n^{th} item, and another box, an n^{th} box.
- First, place the n^{th} item in the n^{th} box.
- Permute all the the other items, which you know how to do.
- Then swap another item with the n^{th} item.
- Again permute the items in boxes 1 to $(n-1)$.
- Swap the item in the n^{th} box with another, different item.
- Repeat until all items have been in box n.

comjnl.oxfordjournals.org/content/6/3/293.full.pdf

Steinhaus-Johnson-Trotter algorithm

Johnson published another algorithm in 1963 for generating permutations.

Attributed to three people: Steinhaus, Johnson and Trotter.

See the article in the link below – it's a trickier read.

Pairwise – the algorithm can be done pair-wise, like Heap's.

Induction is used to show the algorithm works.

www.ams.org/journals/mcom/1963-17-083/S0025-5718-1963-0159764-2/S0025-5718-1963-0159764-2.pdf

Steinhaus-Johnson-Trotter algorithm description

- Start with two items, 1 and 2, and generate their list of permutations 12 and 21.
- Use the two-item list to generate the three item list in the following way:
    - Place 3 at the right of the first element in the two-item list.
    - Then move 3 one place to the left continuosly until its on the left.
    - Then place 3 at the left of the next element in the two-item list.
    - move 3 one place to the right continuously until it reaches the right.
- Use the three item list to generate the four item list in the same way, and so on.

comjnl.oxfordjournals.org/content/6/3/293.full.pdf

Timing Algorithms

timeit command line

```
$ python -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 3: 30.2 usec per loop
$ python -m timeit '"-".join([str(n) for n in range(100)])
10000 loops, best of 3: 27.5 usec per loop
$ python -m timeit '"-".join(map(str, range(100)))'
10000 loops, best of 3: 23.2 usec per loop
```

docs.python.org/3/library/timeit.html

timeit module

```python
import timeit

def test():
    """Stupid test function"""
    L = [i for i in range(100)]

if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test()",
                        setup="from __main__ import test"))
```

docs.python.org/3/library/timeit.html

Functional Programming

Turing Machines

Complexity Classes