```
from __future__ import print_function, division

class Kangaroo:
    """A Kangaroo is a marsupial."""

    def __init__(self, name, contents=[]):
        """Initialize the pouch contents.
        name: string
        contents: initial pouch contents.
        """
        self.name = name
        self.pouch_contents = contents

    def __str__(self):
        """Return a string representaion of this Kangaroo.
        """
        t = [ self.name + ' has pouch contents:' ]
        for obj in self.pouch_contents:
            s = '    ' + object.__str__(obj)
            t.append(s)
        return '\n'.join(t)

    def put_in_pouch(self, item):
        """Adds a new item to the pouch contents.
        item: object to be added
        """
        self.pouch_contents.append(item)
```

पायथॉन विचार

शिका संगणक वैज्ञानिकाप्रमाणे विचार करायला

```
def submod_max():
    for e in stream:
        w[e] = f(S + e) - f(S)
        if w[e] >= 2 * w[argmin]:
            S -= argmin
            S += e
            update_argmin(S, argmin, e)
    return S
```

2nd Edition, Version 2.4.0

Copyright © 2023 Sagar Sudhir Kale (सागर सुधीर काळे).
IN DRAFT STAGE. Last updated: 31 January 2023.

```
kanga = Kangaroo('Kanga')
roo = Kangaroo('Roo')
kanga.put_in_pouch('wallet')
kanga.put_in_pouch('car keys')
kanga.put_in_pouch(roo)

print(kanga)

# If you run this program as is, it seems to work.
# To see the problem, trying printing roo.

# Hint: to find the problem try running pylint.
```

IN DRAFT STAGE. Last updated: 31 January 2023.

पायथॉन विचार

(Think Python)

शिका संगणक वैज्ञानिकाप्रमाणे विचार करायला

(How to Think Like a Computer Scientist)

2nd Edition, Version 2.4.0

मूळ लेखक: ॲलन डाउनी

अनुवाद: सागर सुधीर काळे

Original Author: Allen Downey

Translation: Sagar Sudhir Kale

sagark4.github.io think.python.marathi@gmail.com

ग्रीन टी प्रेस (Green Tea Press)

नीडम, मॅसेचुसेट्स (Needham, Massachusetts)

Copyright notice for the Marathi translation:

Copyright © 2023 Sagar Sudhir Kale.

A-5, Jehangir Apts.,
Syndicate, Murbad Rd.,
Kalyan (W), Dist. Thane,
Maharashtra, India, 421301
Email: sagar.sudhir.kale@gmail.com

Permission is granted to copy, distribute, and/or modify this document under the terms of the Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0), which is available at https://creativecommons.org/licenses/by-nc-sa/4.0/.

वरील लायसन्सद्वारे, व्यावसायिक किंवा व्यापारी वापर सोडून, कोणत्याही प्रकारच्या आर्थिक नफ्यासाठी वापर सोडून, आणि लायसन्समधील सर्व अटींची पूर्तता केल्यास, इतर वापरांसाठी ह्या पुस्तकात बदल करण्याची आणि/किंवा ह्या पुस्तकाच्या प्रती बनवून त्यांचे वितरण करण्याची आपल्याला परवानगी देण्यात येत आहे. संपूर्ण अटींसाठी वरील लायसन्सचे इंग्रजीमधील स्वरूपच अंतिमपणे लागू होईल ज्याची माहिती पुढील लिंकवर दिलेली आहे: https://creativecommons.org/licenses/by-nc-sa/4.0/.

Copyright notice for the original book:

Copyright © 2015 Allen Downey.

Green Tea Press
9 Washburn Ave
Needham MA 02492

Permission is granted to copy, distribute, and/or modify this document under the terms of the Creative Commons Attribution-NonCommercial 3.0 Unported License, which is available at http://creativecommons.org/licenses/by-nc/3.0/.

The original form of this book is LaTeX source code. Compiling this LaTeX source has the effect of generating a device-independent representation of a textbook, which can be converted to other formats and printed.

The LaTeX source for this book is available from http://www.thinkpython2.com

बाबांच्या स्मृतीस.

कर्मण्येवाधिकारस्ते मा फलेषु कदाचन।

प्रस्तावना

ह्या पुस्तकाचा विचित्र इतिहास

जानेवारी १९९९ मध्ये मी प्राथमिक प्रोग्रामिंग (introductory programming) हा विषय जावा (Java) मध्ये शिकवण्याची तयारी करत होतो. हा विषय मी तीन वेळा शिकवला होता, आणि माझा अनुभव निराशाजनक होता. वर्गात अयशस्वी होण्याचे प्रमाण खूप जास्त होते, आणि त्यातही जे विद्यार्थी यशस्वी झाले, त्यांची एकूण कामगिरी खूपच साधारण होती.

पुस्तके ही एक मोठी समस्या होती. जावा (Java) बद्दलच्या अतिअनावश्यक तपशीलांमुळे ती पुस्तके खूप मोठी होती, आणि वरून प्रोग्रामिंग कसे करावे याबद्दल पुरेसे सामान्य मार्गदर्शन त्यांत नव्हते. आणि त्या सर्व पुस्तकांमध्ये ट्रॅप-डोअर इफेक्ट (trap-door effect) आढळून आला, जो की: त्यांची सुरुवात सहज सोपी असे, हळूहळू ती पुढे जात, आणि नंतर कुठेतरी पाचव्या प्रकरणापाशी अचानक कोसळत. विद्यार्थ्यांवर माहितीचा खूप जास्ती भडिमार होई, आणि माझी उर्वरित सेमिस्टर त्यांना बरोबर नेण्याच्या मेटाकुटीत जाई.

ह्या विषयाच्या पहिल्या वर्गाच्या दिवसाच्या दोन आठवड्यांपूर्वी मी ठरवले की मीच माझे स्वतःचे पुस्तक लिहीन. माझी उद्दिष्टे होती:

- हे पुस्तक लहान असावे. विद्यार्थ्यांनी ५० पाने *न* वाचण्यापेक्षा १० पाने वाचलेली बरी.
- ह्या पुस्तकात किचकट शब्द कमी असावेत. मी कमीतकमी आगंतुक शब्द वापरण्याचा प्रयत्न केला आहे आणि प्रत्येक नवीन संज्ञेची/शब्दाची व्याख्या तिच्या पहिल्या वापराच्या आधी दिली आहे.
- विषयाला हळूहळू आकार दिला जावा. ट्रॅप डोर्स (trap doors) टाळण्यासाठी, मी सर्वांत अवघड भाग घेऊन त्यांचे छोट्या भागात विभाजन केले.
- प्रोग्रामिंग वर भर असावा, प्रोग्रामिंग लँग्वेज (programming language) वर नाही.

पुस्तकासाठी एक नाव गरजेचे होते, तर मी एका हुक्कीवर *How to Think Like a Computer Scientist* हे नाव निवडले.

माझी पहिली आवृत्ती ओबडधोबड होती, पण चालून गेली. विद्यार्थ्यांनी तिचे वाचन केले, आणि त्यांना ते इतके पुरेसे समजले की मला वर्गातला वेळ अवघड पण कुतूहल जागवणाऱ्या मुद्द्यांवर आणि विद्यार्थ्यांचा सराव घेण्यात घालवता आला.

मी हे पुस्तक GNU Free Documentation License द्वारे प्रकाशित केले ज्यामुळे कोणालाही ह्या पुस्तकाच्या प्रती बनवण्यास, त्यात बदल करण्यास, आणि त्याचे वितरण करण्यास मुभा देता आली.

जे पुढे झाले, ते मजेशीर होते. जेफ एल्क्नर ह्या व्हर्जिनियातील (Virginia) एका शाळेच्या शिक्षकाने माझे पुस्तक आपल्या छत्राखाली घेतले आणि त्याचे पायथॉन (Python) मध्ये भाषांतर केले. त्याने मला त्याच्या भाषांतराची प्रत पाठवली, आणि मला स्वतःचेच पुस्तक वाचून पायथॉन (Python) शिकण्याचा एक विचित्र पण आनंददायी अनुभव मिळाला. ग्रीन टी प्रेस (Green Tea Press) मार्फत, मी पहिली पायथॉन आवृत्ती २००१ मध्ये प्रकाशित केली.

२००३ साली, मी ओलिन कॉलेज (Olin College) मध्ये शिकवायला सुरुवात केली, आणि मला पहिल्यांदा पायथॉन शिकवायला मिळाले. जावा बरोबरचा विरोधाभास खूपच जाणवण्यासारखा होता. विद्यार्थ्यांची विषयाशी झटापट कमी झाली, त्यांना जास्ती शिकायला मिळाले, त्यांनी गमतीदार प्रोजेक्ट्सवर काम केले, एकंदरीत त्यांना खूप जास्ती मजा आली.

तेव्हापासून मी पुस्तकावर काम चालू ठेवले आहे, चुका दुरूस्त करणे, काही उदाहरणांचे चांगले स्पष्टीकरण देणे, आणि अधिक माहितीची भर घालणे (मुख्यतः स्वअभ्यासाच्या प्रश्नसंचातील प्रश्न).

सगळ्याचा परिणाम म्हणजे हे पुस्तक, आता *Think Python* ह्या कमी भव्य नावासहित. ह्यातले काही बदल म्हणजे:

- प्रत्येक प्रकरणाच्या शेवटी डीबगिंग वर एक विभाग जोडला आहे. हे विभाग बग शोधण्याची आणि टाळण्याची ढोबळ तंत्रे, आणि पायथॉनच्या काही अज्ञात धोक्यांच्या सूचना मांडतात.
- काही छोट्या चाचण्यांपासून ते काही मोठ्या प्रोजेक्ट्स पर्यंत प्रश्नांची भर घातली. बहुतांश प्रश्नांमध्ये तुम्हाला माझ्या उत्तरांची लिंक (वेबसाइटची) आढळून येईल.
- विविध केस स्टडीजचा (case studies), म्हणजेच मोठी उदाहरणे, प्रश्नोत्तरे, आणि त्यावरील चर्चा ह्यांचा समावेश केला.
- मी प्रोग्राम डेव्हेलपमेंट प्लान्स (program development plans) आणि मूलभूत डिझाइन पॅटर्न्स (design patterns) ह्यांवरच्या चर्चेचा विस्तार केला.
- डीबगिंग आणि अल्गोरिदम (algorithm) विश्लेषण ह्यावर परिशिष्ट जोडले.

Think Python च्या दुसऱ्या आवृत्तीची ही वैशिष्ट्ये आहेत:

- पुस्तक आणि बरोबरचा सगळा कोड (code) पायथॉन ३ (Python 3) वर नेला आहे.
- ब्राउझरवर (browser), उदा., मोझिला फायरफॉक्स (Mozilla Firefox), गूगल क्रोम (Google Chrome), पायथॉन कसे चालवता येऊ शकते ह्यावर ह्या पुस्तकात काही विभागांचा आणि संलग्न वेबसाइटवर (website) थोड्या माहितीचा समावेश केला.
- प्रकरण ४.१ मध्ये माझ्या स्वतःचे ग्राफिक्स पॅकेज Swampy सोडून जास्ती प्रसिद्ध असलेले पायथॉनचे `turtle` मोड्युल (module) वापरायला घेतले; हे इन्स्टॉल (install) करायला सोपे आणि जास्ती काम करणारे आहे.
- 'The Goodies' नावाच्या प्रकरणाचा समावेश केला; ह्यात पायथॉनच्या अजून काही वैशिष्ट्यांशी परिचय केला गेला आहे जी एकदम गरजेची नसली तरी भरपूर उपयोगी पडू शकतील अशी आहेत.

मला आशा आहे की तुम्हाला ह्या पुस्तकाचा वापर करताना खूप आनंद मिळेल, आणि ह्या पुस्तकाचा तुम्हाला प्रोग्रामिंग कसे करावे आणि संगणक वैज्ञानिकाप्रमाणे विचार कसा करावा हे शिकायला थोडा का होईना उपयोग होईल.

ॲलन डाउनी (Allen B. Downey)

ओलिन कॉलेज (Olin College)

आभार

जेफ एल्क्नर (Jeff Elkner) ह्यांचे अनेक आभार; त्यांनी माझे जावाच्या पुस्तकाचे पायथॉनमध्ये भाषांतर केले, ज्यामुळे हा प्रकल्प सुरू झाला आणि माझा परिचय माझ्या होणाऱ्या आवडत्या (प्रोग्रामिंग) लँग्वेजशी करून दिला.

क्रिस मायर्स (Chris Meyers) ह्यांचेसुद्धा धन्यवादआभार; ह्यांनी *How to Think Like a Computer Scientist* मध्ये अनेक विभागांचे योगदान दिले.

Free Software Foundation चे GNU Free Documentation License बनवण्यासाठी आभार; ह्यामुळे जेफ आणि क्रिस बरोबरचा माझा सहयोग शक्य झाला; आणि Creative Commons सध्या वापरत असलेल्या लायसन्सबद्दल आभार.

Lulu च्या *How to Think Like a Computer Scientist* वर काम केलेल्या एडिटर्सना धन्यवाद.

O'Reilly Media च्या *Think Python* वर काम केलेल्या एडिटर्सना धन्यवाद.

पूर्वीच्या आवृत्त्यांवर काम करणाऱ्या सर्व विद्यार्थ्यांचे आणि सर्व दुरुस्त्या आणि सुधार पाठवणाऱ्यांचे आभार.

अनुवादकाचे मनोगत

माझ्या मराठी भाषेवरील आणि महाराष्ट्रवासीयांवरील प्रेमापोटीच हा उपक्रम. माझे स्वतःचे शिक्षण मराठी माध्यमातून झाल्यामुळे मला इंग्रजी *न* जमणे म्हणजे काय ह्याची चांगलीच कल्पना आहे. शिकण्याची खूप इच्छा असूनही एकतर शिक्षक चांगले नाही मिळत किंवा पुस्तके फक्त इंग्रजीमधून आणि किचकट भाषेत असतात अशी इकडे आड तिकडे विहीर अवस्था खूप लोकांची होते. मूळ इंग्रजी पुस्तकाचे मराठीत भाषांतर करण्यामागे माझा मुख्य उद्देश म्हणजे: प्रोग्रामिंगसारखा मूलभूत विषय मराठीतून शिकता यावा आणि इंग्रजी भाषेकडे जाण्याचा मार्ग सुकर व्हावा.

इंग्रजी ही जगाची भाषा असल्यामुळे ती वापरणे अनिवार्यच आहे. विशेषतः संगणक विज्ञानाच्या जगात इंग्रजीशिवाय काहीही अशक्य आहे. त्यामुळेच ह्या पुस्तकात मी सर्व इंग्रजी संज्ञांचे किंवा शब्दांचे मराठीकरण केलेले नाही. असा हट्ट धरणे अनुचित आहे. जे तांत्रिकी शब्द तुम्हाला पुढे जाऊन इंग्रजीमध्येच वापरायचे आहेत आणि ज्यांना एकदम योग्य असा मराठी शब्द नाहीये असल्या शब्दांना मी इंग्रजीमध्येच ठेवले आहे. माझ्या भावाचे शिक्षणही मराठी माध्यमातूनच झाले; त्याने हे सुचवले की काही तांत्रिकी शब्दांबरोबर त्यांचे इंग्रजी स्वरूप (स्पेलिंगसह) दिल्यास ह्या पुस्तकाचा पायथॉनच नाही तर इंग्रजी शिकण्यासाठीदेखील फायदा होईल. त्यामुळे मी वारंवार कंसांमध्ये इंग्रजी स्वरूप दिले आहे. उदा., एक वाक्य असे आहे, 'कोड-चा एक भाग एका फंक्शनमध्ये जमा करण्याला **एन्कॅप्सुलेशन** (**encapsulation, विभागीकरण**) म्हणतात.' असे वारंवार करण्याचा एकमेव उद्देश म्हणजे पुनरावृत्ती केल्याने तुम्हाला ते शब्द लक्षात ठेवणे सोपे जाईल.

अजून एक मुद्दा पुढीलप्रमाणे. काही इंग्रजी शब्दांना मूळ लॅटिन लिपीमध्ये *न* लिहिता त्यांचे देवनागरीकरण करण्याचे (उदा., लिस्ट, डिक्शनरी, इत्यादीचे) कारण म्हणजे वाचताना एका लिपीतून दुसऱ्या लिपीत संदर्भ बदलताना आपल्या मेंदूला काही कार्य करावे लागते. नवीन माहिती शिकताना ह्यामुळे थोडी अडचण होऊ शकते.

संलग्न इंग्रजी मजकूरसाठी तुम्ही मूळ पुस्तक नक्की बघा.

गेली १६ वर्षे व्यावसायिकपणे (आणि वैयक्तिक आयुष्यातदेखील गेली १०-१२ वर्षे) मी जवळजवळ १००% इंग्रजीच वापरल्यामुळे हे भाषांतर करताना खूप श्रम लागले हे खरे. चूकभूल माफ असावी.

मूळ इंग्रजी पुस्तकास योगदान देणाऱ्यांची यादी

शंभरहून अधिक चलाख नजरेच्या आणि दक्ष वाचकांनी गेल्या काही वर्षांत सूचना आणि सुधारणा पाठवल्या आहेत. त्यांचे ह्या प्रकल्पासाठीचे योगदान आणि उत्साह खूप महत्त्वाचे ठरले आहे.

- Lloyd Hugh Allen ह्यांनी विभाग ८.४ मध्ये एक दुरुस्ती पाठवली.
- Yvon Boulianne ह्यांनी प्रकरण ५ मध्ये सिमँटिक एररची दुरुस्ती पाठवली.
- Fred Bremmer ह्यांनी विभाग २.१ मध्ये दुरुस्ती पाठवली.
- Jonah Cohen ह्यांनी पुस्तकाच्या LaTeX कोड-चे सुंदर HTML मध्ये रुपांतर करणाऱ्या Perl स्क्रिप्ट्स लिहिल्या.
- Michael Conlon ह्यांनी प्रकरण २ मधील एक व्याकरणातील चूक पाठवली आणि प्रकरण १ मधील शैलीची दुरुस्ती पाठवली. त्यांनी इंटरप्रिटर्सच्या तांत्रिक बाबींवर चर्चा सुरू केली.

- Benoît Girard ह्यांनी विभाग ५.६ मधील एका गमतीशीर चुकीची दुरुस्ती पाठवली.
- Courtney Gleason आणि Katherine Smith ह्यांनी `horsebet.py` लिहिली जी पुस्तकाच्या आधीच्या आवृत्तीमध्ये केस स्टडी म्हणून वापरली गेली होती. त्यांचा प्रोग्राम पुस्तकाच्या वेबसाइटवर मिळेल.
- Lee Harr ह्यांनी इतक्या दुरुस्त्या पाठवल्या की त्यांची यादी दाखवण्यासाठी आपल्याकडे इथे जागा नाही. खरे तर त्यांना पुस्तकाचे मुख्य संपादक म्हटले पाहिजे.
- James Kaylin ह्यांनी अनेक दुरुस्त्या पाठवल्या.
- David Kershaw ह्यांनी विभाग ३.१० मधील `catTwice` फंक्शनमध्ये असलेल्या चुका दुरूस्त केल्या.
- Eddie Lam ह्यांनी प्रकरण १, २, आणि ३ आणि Makefile मध्ये अनेक दुरुस्त्या पाठवल्या; आणि versioning scheme बनवली.
- Man-Yong Lee ह्यांनी विभाग २.४ मध्ये दुरुस्ती पाठवली.
- David Mayo ह्यांनी प्रकरण १ मध्ये 'unconsciously' हा शब्द 'subconsciously' असला पाहिजे ह्याची आठवण दिली.
- Chris McAloon ह्यांनी विभाग ३.९ आणि ३.१० मध्ये अनेक दुरुस्त्या पाठवल्या.
- Matthew J. Moelter ह्यांनी खूप काळापासून योगदान दिले आहे आणि अनेक दुरुस्त्या पाठवल्या आहेत.
- Simon Dicon Montford ह्यांनी प्रकरण ३ मध्ये आणि प्रकरण १३ मध्ये अनेक दुरुस्त्या पाठवल्या.
- John Ouzts ह्यांनी प्रकरण ३ मधील 'return value' ची डेफनिशन सुधारली.
- Kevin Parks ह्यांनी पुस्तकाचे वितरण चांगले करण्यासाठी अनेक मोलाच्या सूचना दिल्या.
- David Pool ह्यांनी प्रकरण १ च्या शब्दार्थांमधील एक टाइपिंगची चूक प्रोत्साहनाचे शब्द आणि पाठवले.
- Michael Schmitt ह्यांनी फाइल्स आणि एक्सेप्शन्स वरील प्रकरणात एक दुरुस्ती पाठवली.
- Robin Shaw ह्यांनी विभाग १३.१ मध्ये printTime फंक्शन डिफाइन करायच्या आधीच वापरले जात आहे ह्याची नोंद दिली.
- Paul Sleigh ह्यांनी प्रकरण ७ मध्ये एक चूक शोधली आणि Jonah Cohen ह्यांच्या HTML निर्माण करणाऱ्या Perl स्क्रिप्टमध्ये एक बग शोधला.
- Craig T. Snydal ह्यांनी Drew University मध्ये एका वर्गात हे पुस्तक वापरले आणि अनेक मोलाच्या सूचना आणि दुरुस्त्या पाठवल्या.
- Ian Thomas आणि त्यांचे विद्यार्थी एका प्रोग्रामिंगच्या वर्गात हे पुस्तक वापरत आहेत. उत्तरार्धातील प्रकरणांची तपासणी करणारे ते पहिलेच असून त्यांनी अनेक दुरुस्त्या आणि सूचना केल्या आहेत.
- Keith Verheyden ह्यांनी प्रकरण ३ मध्ये एक दुरुस्ती पाठवली.
- Peter Winstanley ह्यांनी प्रकरण ३ मधील खूप वेळ राहिलेली Latin बद्दलची चूक पाठवली.
- Chris Wrobel ह्यांनी फाइल आणि एक्सेप्शनवरील प्रकरणात अनेक दुरुस्त्या पाठवल्या.
- Moshe Zadka ह्यांनी ह्या प्रकल्पाला मोलाचे योगदान दिले आहे. डिक्शनरीवरील प्रकरणाचा मसुदा त्यांनी लिहिला आणि सुरुवातीच्या काळात सतत मार्गदर्शन केले.
- Christoph Zwerschke ह्यांनी अनेक दुरुस्त्या सुचवल्या आणि *gleich* आणि *selbe* मधील फरक सांगितला.
- James Mayer ह्यांनी स्पेलिंगच्या अनेक चुका पाठवल्या, काही ह्या योगदानकर्त्यांच्या यादीतल्याही.
- Hayden McAfee ह्यांनी दोन उदाहरणांमधील संभ्रमात टाकू शकणारी विसंगती पकडली.
- Angel Arnal हे पुस्तकाच्या स्पॅनिश आवृत्तीवर काम करत होते. त्यांनी इंग्रजी आवृत्तीत अनेक चुका शोधल्या.

- Tauhidul Hoque आणि Lex Berezhny ह्यांनी प्रकरण १ मधील आकृत्या बनवल्या आणि इतर अनेक आकृत्या सुधारल्या.
- Dr. Michele Alzetta ह्यांनी प्रकरण ८ मधील एक चूक पकडली आणि Fibonacci आणि Old Maid बद्दल कल्पक सूचना पाठवल्या.
- Andy Mitchell ह्यांनी प्रकरण १ मधील एक टाइपिंगची चूक शोधली आणि प्रकरण २ मधील एक चुकीचे उदाहरण.
- Kalin Harvey ह्यांनी प्रकरण ७ मध्ये एक सुधारण सुचवली आणि काही टाइपिंगच्या चुका पकडल्या.
- Christopher P. Smith ह्यांनी अने टाइपिंगच्या चुका शोधल्या आणि पुस्तक पायथॉन २.२ ला अपडेट करण्यासाठी मदत केली.
- David Hutchins ह्यांनी प्रस्तावनेत एक टाइपिंगची चूक शोधली.
- Gregor Lingl हे व्हिएना, ऑस्ट्रिया येथे एका शाळेत पायथॉन शिकवतात; ते पुस्तकाचे जर्मन भाषांतर करत आहेत आणि त्यांनी प्रकरण ५ मध्ये थोड्या गंभीर अशा चुका शोधल्या.
- Julie Peters ह्यांनी प्रस्तावनेत एक टाइपिंगची चूक शोधली.
- Florin Oprina ह्यांनी `makeTime` आणि `printTime` मध्ये सुधार सुचवले, आणि अजून एक छान टाइपिंगची चूक शोधली.
- D. J. Webre ह्यांनी प्रकरण ३ मध्ये एक सुधारणा सुचवली.
- Ken found ह्यांनी प्रकरण ८, ९, आणि ११ मध्ये मूठभर चुका शोधल्या.
- Ivo Wever ह्यांनी प्रकरण ५ मध्ये एक चूक पकडली आणि प्रकरण ३ मध्ये एक सुधारणा सुचवली.
- Curtis Yanko ह्यांनी प्रकरण २ मध्ये एक सुधारणा सुचवली.
- Ben Logan sent ह्यांनी टाइपिंगच्या चुका आणि HTML भाषांतरतल्या अडचणी पाठवल्या.
- Jason Armstrong ह्यांनी प्रकरण २ मधील एक राहिलेला शब्द नोंदला.
- Louis Cordier ह्यांनी प्रकरण १६ मधील कोड आणि मजकुरातील विसंगती नोंदली.
- Brian Cain ह्यांनी प्रकरण २ आणि ३ मध्ये अनेक सुधारणा सुचवल्या.
- Rob Black ह्यांनी दुरुस्त्यांचा साठा पाठवला ज्यात पायथॉन २.२ साठीच्या बदलांचा समावेश देखील होता.
- Jean-Philippe Rey जे École Centrale Paris इथे असतात त्यांनी अनेक पॅचेस पाठवले ज्यात पायथॉन २.२ साठीच्या बदलांचा आणि अनेक विचारपूर्वक सूचनांचा देखील समावेश होता.
- Jason Mader जे George Washington University इथे असतात त्यांनी अनेक फायदेशीर सूचना आणि दुरुस्त्या पाठवल्या.
- Jan Gundtofte-Bruun ह्यांनी आठवण करून दिली की " 'a error' is an error."
- Abel David आणि Alexis Dinno ह्यांनी आठवण करून दिली की 'matrix' चे अनेकवचन 'matrices' आहे, 'matrixes' नाही. ही चूक पुस्तकात अनेक वर्षे होती पण नावाचे सारखेच initials असणाऱ्या दोन जणांनी एकाच दिवशी ती पाठवली. विचित्र योगायोग.
- Charles Thayer ह्यांनी काही स्टेटमेंट्सच्या शेवटी लावलेले अर्धविराम (semicolons) काढायला आणि 'argument' आणि 'parameter' ह्यांचा नीट वापर करायला प्रोत्साहन दिले.
- Roger Sperberg ह्यांनी प्रकरण ३ मधील एका विचित्र लॉजिकची नोंद केली.
- Sam Bull ह्यांनी प्रकरण २ मधील एका संभ्रमात टाकणाऱ्या परिच्छेदाची नोंद केली.
- Andrew Cheung ह्यांनी 'डेफनिशनच्या आधीच वापरा'च्या दोन चुका नोंदल्या.

- C. Corey Capel ह्यांनी प्रकरण ४ मधील एक टाइपिंगची चूक आणि 'Third Theorem of Debugging' मधील एक राहिलेला शब्द हे पाठवले.
- Alessandra ह्यांनी Turtle चा संभ्रम दूर केला.
- Wim Champagne ह्यांनी डिक्शनरीच्या एका उदाहरणात मेंदूतील चूक शोधली.
- Douglas Wright ह्यांनी `arc` मधील एक चूक शोधली.
- Jared Spindor ह्यांनी एका वाक्याची शेवटी असलेल्या केराची नोंद केली.
- Lin Peiheng ह्यांनी अनेक फायदेशीर सूचना पाठवल्या.
- Ray Hagtvedt ह्यांनी दोन चुका आणि एक नसलेली चूक हे पाठवले.
- Torsten Hübsch ह्यांनी Swampy मधील एक विसंगती पाठवली.
- Inga Petuhhov ह्यांनी प्रकरण १४ मधील एका उदाहरणात एक दुरुस्ती पाठवली.
- Arne Babenhauserheide ह्यांनी अनेक उपयोगी दुरुस्त्या पाठवल्या.
- Mark E. Casida हे हे शब्दांची पुनरावृत्ती शोधण्यात निपुण आहेत.
- Scott Tyler ह्यांनी एक राहिलेला a भरला. आणि नंतर दुरुस्त्यांचा ढिगारा पाठवला.
- Gordon Shephard ह्यांनी अनेक दुरुस्त्या पाठवल्या, सर्व वेगवेगळ्या इमेल्समध्ये.
- Andrew Turner `spotted` an error in Chapter 8.
- Adam Hobart ह्यांनी `arc` मधील एक चूक दुरूस्त केली.
- Daryl Hammond आणि Sarah Zimmerman ह्यांनी नोंद केली की `math.pi` लवकरच वाढला. आणि Zim ने एक टाइपिंगची चूक शोधली.
- George Sass ह्यांनी एका डीबगिंग विभागात एक बग शोधला.
- Brian Bingham ह्यांनी प्रश्न ११.५ सुचवला.
- Leah Engelbert-Fenton ह्यांनी नोंद केली की ह्या पुस्तकात दिलेल्या सूचनेच्या उलट जाऊन `tuple` नावाचे व्हेरिएबल बनवले गेले. आणि नंतर टाइपिंगच्या चुकांचा गठ्ठा पाठवला आणि एका 'डेफनिशनच्या आधीच वापरा'ची नोंद केली.
- Joe Funke ह्यांनी एक टाइपिंगची चूक हेरली.
- Chao-chao Chen ह्यांनी फिबोनाचीच्या उदाहरणातील एक विसंगती शोधली.
- Jeff Paine ह्यांना space आणि spam मधील फरक माहीत आहे.
- Lubos Pintes ह्यांनी एक टाइपिंगची चूक पाठवली.
- Gregg Lind आणि Abigail Heithoff ह्यांनी प्रश्न १४.३ सुचवला.
- Max Hailperin ह्यांनी अनेक दुरुस्त्या आणि सूचना पाठवल्या. ते *Concrete Abstractions* ह्या उत्तम पुस्तकाचे लेखक आहेत; हे पुस्तक झाल्यावर तुम्ही ते वाचू शकता.
- Chotipat Pornavalai ह्यांना एका एरर मेसेजमध्ये एरर मिळाला.
- Stanislaw Antol ह्यांनी अनेक उपयोगी सूचनांची यादी पाठवली.
- Eric Pashman ह्यांनी प्रकरण ४--११ मध्ये अनेक दुरुस्त्या पाठवल्या.
- Miguel Azevedo ह्यांनी काही टाइपिंगच्या चुका पाठवल्या.
- Jianhua Liu ह्यांनी दुरुस्त्यांची एक मोठी यादी पाठवली.

- Nick King ह्यांना एक राहिलेला शब्द मिळाला.
- Martin Zuther ह्यांनी सूचनांची एक मोठी यादी पाठवली.
- Adam Zimmerman ह्यांना 'instance' च्या एका instance मध्ये एक विसंगती मिळाली आणि इतर अनेक चुका मिळाल्या.
- Ratnakar Tiwari ह्यांनी degenerate triangles बद्दल एक तळटीप सुचवली.
- Anurag Goel ह्यांनी `is_abecedarian` साठी अजून एक उत्तर पाठवले आणि इतर काही दुरुस्त्या पाठवल्या. आणि त्यांना Jane Austen ची स्पेलिंग माहीत आहे.
- Kelli Kratzer ह्यांनी एक टाइपिंगची चूक शोधली.
- Mark Griffiths ह्यांनी प्रकरण ३ मधील एका संभ्रमात टाकणाऱ्या उदाहरणाची नोंद केली.
- Roydan Ongie ह्यांना Newton च्या पद्धतीतील एक चूक मिळाली.
- Patryk Wolowiec ह्यांनी HTML आवृत्तीतील एका समस्येबद्दल मदत केली.
- Mark Chonofsky ह्यांनी मला पायथॉन ३ मधील एका नवीन कीवर्ड सांगितला.
- Russell Coleman ह्यांनी भूमितीत मदत केली.
- Nam Nguyen ह्यांनी एक टाइपिंगची चूक शोधली आणि सांगितले की एके ठिकाणी Decorator पॅटर्न वापरला जात आहे पण त्याचा नावाने उल्लेख नाही.
- Stéphane Morin ह्यांनी अनेक दुरुस्त्या आणि सूचना पाठवल्या.
- Paul Stoop ह्यांनी `uses_only` मधील टाइपिंगची चूक दुरूस्त केली.
- Eric Bronner ह्यांनी ऑपरेटर्सच्या अनुक्रमाच्या चर्चेतील एक संभ्रमात टाकणारा मुद्दा नोंदला.
- Alexandros Gezerlis ह्यांनी खूप खूप आणि खूप उच्च दर्जाच्या सूचना पाठवल्या. मनापासून आभार!
- Gray Thomas ह्यांना डावा-उजवा फरक माहीत आहे.
- Giovanni Escobar Sosa ह्यांनी टाइपिंगच्या चुका आणि सूचनांची मोठी यादी पाठवली.
- Daniel Neilson ह्यांनी ऑपरेटर्सच्या अनुक्रमाबद्दलची एक चूक दुरूस्त केली.
- Will McGinnis ह्यांनी ही नोंद केली की `polyline` दोन वेगळ्या ठिकाणी वेगळ्या प्रकारे वापरले जात होते.
- Frank Hecker ह्यांनी एका प्रश्नातील अस्पष्टपणा आणि काही तुटलेल्या लिंक्स नोंदल्या.
- Animesh B ह्यांनी एक संभ्रमात टाकणारे उदाहरण ठीक करण्यात मदत केली.
- Martin Caspersen ह्यांना दोन round-off एरर्स मिळाले.
- Gregor Ulm ह्यांनी अनेक दुरुस्त्या आणि सूचना पाठवल्या.
- Dimitrios Tsirigkas ह्यांनी एक प्रश्न स्पष्ट करण्याची सूचना केली.
- Carlos Tafur ह्यांनी पानभर टाइपिंगच्या चुका आणि सूचना पाठवल्या.
- Martin Nordsletten ह्यांना एका प्रश्नाच्या उत्तरात बग मिळाला.
- Sven Hoexter ह्यांनी नोंद केली की `input` नावाचे व्हेरिएबल एका बिल्ट-इन फंक्शनला झाकते.
- Stephen Gregory ह्यांनी पायथॉन ३ मध्ये `cmp` विषयीची अडचण नोंदली.
- Ishwar Bhat ह्यांनी फर्मॅटच्या शेवटचे प्रमेयात एक दुरुस्ती पाठवली.
- Andrea Zanella ह्यांनी पुस्तकाचे इटालियनमध्ये भाषांतर केले आणि अनेक दुरुस्त्या पाठवल्या.

- Melissa Lewis आणि Luciano Ramalho ह्यांना दुसऱ्या आवृत्तीवर उत्तम सूचना देण्याबद्दल अनेक अनेक आभार.
- PythonAnywhere च्या Harry Percival ह्यांचे लोकांना पायथॉन ब्राउझरवर चालवता यावे म्हणून केलेल्या मदतीबद्दल धन्यवाद.
- Xavier Van Aubel ह्यांनी दुसऱ्या आवृत्तीत अनेक फायदेशीर दुरुस्त्या केल्या.
- William Murray ह्यांनी floor division ची डेफनिशन सुधारली.
- Per Starbäck ह्यांनी पायथॉन ३ मधील universal newlines ताजी माहिती दिली.
- Laurent Rosenfeld आणि Mihaela Rotaru ह्यांनी ह्या पुस्तकाचे फ्रेंचमध्ये भाषांतर केले आणि अनेक दुरुस्त्या पाठवल्या.

  आणि पुढील लोकांनीदेखील टाइपिंगच्या चुका शोधल्या किंवा दुरुस्त्या केल्या: Czeslaw Czapla, Dale Wilson, Francesco Carlo Cimini, Richard Fursa, Brian McGhie, Lokesh Kumar Makani, Matthew Shultz, Viet Le, Victor Simeone, Lars O.D. Christensen, Swarup Sahoo, Alix Etienne, Kuang He, Wei Huang, Karen Barber, आणि Eric Ransom.

अनुक्रमणिका

प्रकरण १

प्रोग्रामचा मार्ग

तुम्हाला संगणक वैज्ञानिकाप्रमाणे विचार करण्यास शिकवणे हा ह्या पुस्तकाचा उद्देश आहे. अशा विचारसरणीमध्ये गणित, इंजिनीयरींग, आणि नैसर्गिक विज्ञानांमधील गहन कल्पनांचा सुंदर मिलाफ होतो. गणितज्ञांप्रमाणे, संगणक वैज्ञानिकसुद्धा तर्कशुद्ध भाषेमध्ये कल्पना मांडतात (मुख्यतः गणन म्हणजेच, computation बद्दल). इंजिनीअर प्रमाणे, ते डिझाइन करतात, एका रचनेमध्ये असेंबल (assemble) करतात, आणि अशा प्रकारे विविध पर्याय जमा करून त्यातला उत्तम पर्याय निवडतात. वैज्ञानिकांप्रमाणे, ते किचकट रचनांचे निरीक्षण करतात आणि त्यांचा अभ्यास करतात.

संगणक वैज्ञानिकांचे सर्वांत महत्त्वाचे कौशल्य म्हणजे: **प्रॉब्लेम-सॉल्व्हिंगचे कौशल्य** (**problem-solving skills**). प्रॉब्लेम म्हणजे साधारणपणे गणित, इंजिनीयरींग, किंवा विज्ञानातील लहान किंवा मोठा संकल्पनात्मक प्रश्न ज्याचे उत्तर पद्धतशीरपणे किंवा तार्किक विश्लेषण करून शोधता येऊ शकते; आपण ह्या पुस्तकात खूप उदाहरणे पाहणारच आहोत[१]. प्रॉब्लेम-सॉल्व्हिंग म्हणजे खरा प्रॉब्लेम काय आहे हे उकलण्याची, कल्पनाशीलपणे तो कसा सोडवावा ह्याचा विचार करण्याची, आणि त्या प्रॉब्लेमचे उत्तर स्पष्टपणे आणि अचूकपणे व्यक्त करण्याची क्षमता होय. आणि प्रोग्रामिंग शिकण्याची प्रक्रिया ही प्रॉब्लेम-सॉल्व्हिंगच्या कौशल्याचा सराव करण्याची उत्तम संधी आहे. म्हणूनच ह्या प्रकरणाचे नाव 'प्रोग्रामचा मार्ग' हे आहे.

तुम्ही प्रोग्रामिंग हे महत्त्वाचे कौशल्य तर शिकालच, पण तुम्ही प्रोग्राम खरे तर कोणतेतरी उद्दिष्ट पार पाडण्यासाठी वापराल. आपण जसजसे पुढे जाऊ, तसतसे ते उद्दिष्ट स्पष्ट होईल.

१.१ प्रोग्राम म्हणजे काय?

एखादे गणन कसे करावे ते सांगणाऱ्या क्रमाने दिलेल्या सूचना म्हणजेच **प्रोग्राम** (**program**). ते गणन गणिती असू शकते, उदा., समीकरणे सोडवणे, पण वेगळ्या प्रकारचेसुद्धा असू शकते, उदा., एखाद्या डॉक्युमेंटमध्ये एखादा मजकूर शोधून त्याला दुसऱ्या मजकुराने बदलणे, किंवा काहीतरी ग्राफिकल, उदा., फोटोवरच्या आणि व्हिडिओवरच्या प्रक्रिया.

काही साधारण सूचना प्रत्येक प्रोग्रामिंग लँग्वेजमध्ये आढळून येतात (पण सूचनांचा तपशील प्रोग्रामिंग लँग्वेज वर अवलंबून आहे).

इनपुट (input): कीबोर्ड (keyboard), फाइल (file), नेटवर्क (network), किंवा दुसऱ्या कोणत्यातरी डिव्हाइस (device) मधून डेटा (data) वाचा.

[१] आपण कधीच 'प्रॉब्लेम' ह्या शब्दाचा वापर इतर प्रकारे करणार नाही. उदा., प्रोग्राममधील एरर/बग (error/bug), संकल्पनात्मक संभ्रम, इत्यादींना आपण ह्या पुस्तकात कधीच प्रॉब्लेम असे संबोधणार नाही आहोत. आपल्यासाठी **प्रॉब्लेम** म्हणजे वर सांगितल्याप्रमाणे गणित, इंजिनीयरींग, किंवा विज्ञानाशी संबंधित संकल्पनात्मक प्रश्न.

आउटपुट (output): स्क्रीनवर (screen) डेटा दाखवा, फाइलमध्ये सेव्ह (save) करा, किंवा नेटवर्कवर पाठवा, इत्यादी.

गणित (math): मूलभूत गणिती क्रिया करा, उदा., बेरीज आणि गुणाकार.

कंडिश्नल एक्सेक्युशन (conditional execution): काही अटी तपासून त्याप्रमाणे कोड (code) चा एखादा भाग कृतीत उतरवा (म्हणजेच, एक्सेक्युट करा). (अनुवादकाची टिप्पणी: कोड म्हणजे प्रोग्रामच्या लिखितस्वरुपातील सूचना.)

पुनरावृत्ती (repetition): एखादी कृती (सहसा थोड्याशा फरकाने) नियमितपणे करा.

तुमचा विश्वास नाही बसणार, पण खरे तर इतक्याच प्रकारच्या सूचना आहेत. तुम्ही आतापर्यंत वापरलेला प्रत्येक प्रोग्राम कितीही किचकट असला तरी अशाच सूचनांचा बनलेला आहे. ह्याचा अर्थ असा की प्रोग्रामिंग म्हणजे मोठे आणि गुंतागुंतीचे कार्य लहान आणि लहान भागात तोपर्यंत विभागणे जोपर्यंत ते भाग ह्या सूचनांनी सोडवता येत नाहीत.

१.२ पायथॉन चालवणे

पायथॉन सुरू करण्यापुढे एक आव्हान म्हणजे तुम्हाला पायथॉन आणि इतर सॉफ्टवेअर्स तुमच्या काँप्युटरवर इन्स्टॉल करावी लागतात. जर तुमचा तुमच्या ऑपरेटिंग-सिस्टमशी (operating system, उदा., विंडोझ, उबुंटू) व्यवस्थित परिचय असेल, विशेषतः जर तुम्हाला कमांड-लाइन (command line) सहजगत्या वापरता येत असेल, तर तुम्हाला पायथॉन इन्स्टॉल करायला काहीच त्रास होणार नाही. पण नवशिक्यांना हे आणि प्रोग्रामिंग दोन्ही एकदम शिकणे त्रासदायक ठरू शकते.

ही असुविधा टाळण्यासाठी तुम्ही पायथॉन ब्राउझर (browser) वर, म्हणजेच उदा., मोझिला फायरफॉक्स (Mozilla Firefox), गूगल क्रोम (Google Chrome), ह्यावर वापरायला सुरुवात करा. नंतर जेव्हा तुमची पायथॉनशी व्यवस्थित ओळख होईल तेव्हा काँप्युटरवर कसे इन्स्टॉल करायचे हे आपण बघू.

इंटरनेट (internet) वर पुष्कळ वेबसाइट्स (websites) आहेत ज्यांवर पायथॉन प्रोग्राम चालवता येतो. तुम्हाला कोणती माहीत असेल तर खुशाल वापरा. PythonAnywhere हा एक चांगला पर्याय आहे. पुढील लिंकवर ते कसे वापरायचे ह्याच्या तपशीलवार सूचना (इंग्रजीमध्ये) दिलेल्या आहेत: http://tinyurl.com/thinkpython2e.

अनुवादकाची टिप्पणी: अजून एक चांगला पर्याय म्हणजे https://replit.com/.

पायथॉनचे दोन प्रकार आहेत, पायथॉन २ आणि पायथॉन ३. त्यांच्यात बऱ्यापैकी साम्य आहे, म्हणजे तुम्ही एक शिकलात तर तुम्हाला दुसरे वापरणे सोपे पडेल. खरे तर सुरुवातीला तुम्हाला त्यातले फरक जाणवणारही नाहीत. हे पुस्तक पायथॉन ३ वर आहे, पण मी पायथॉन २ वर काही माहिती दिली आहे.

पायथॉनचा **इंटरप्रिटर** (**interpreter**) हा एक प्रोग्राम असतो आणि जो पायथॉन कोड वाचून चालवतो (run/execute करतो). तुमच्या एनव्हायर्नमेंट (environment) नुसार तुम्ही एका आयकॉनवर क्लिक करून किंवा `python` कमांड कमांड-लाइनवर देऊन इंटरप्रिटर सुरू करू शकता. तो जेव्हा सुरू होतो, तेव्हा तुम्हाला असे काहीतरी दिसेल:

```
Python 3.4.0 (default, Jun 19 2015, 14:20:21)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

पहिल्या तीन ओळींवर इंटरप्रिटरविषयी आणि तो ज्या ऑपरेटिंग-सिस्टम वर चालतो आहे तिच्याविषयी माहिती आहे, तर ही माहिती तुमच्यासाठी वेगळी असू शकते. पण तुम्ही तपासून बघा की आवृत्ती क्रमांक जो ह्याठिकाणी `3.4.0` आहे त्याची सुरुवात 3 ने होतेय, म्हणजेच तुम्ही पायथॉन ३ चालवत आहात. जर तो क्रमांक २ पासून सुरू झाला असता तर त्याचा अर्थ तुम्ही (बरोबर ओळखले) पायथॉन २ चालवत आहात हा असला असता.

शेवटची ओळ ' >>> ' हा **प्रॉम्प्ट** (**prompt**) दाखवते म्हणजेच इंटरप्रिटर तुमच्या कोड-साठी तयार आहे. जर तुम्ही कोड-ची एक ओळ लिहून एंटर (enter) मारले तर इंटरप्रिटर खालील उत्तर दाखवेल:

```
>>> 1 + 1
2
```

आता तुम्ही सुरुवात करण्यास तयार आहात. इथून पुढे मी असे गृहीत धरून चालेल की तुम्हाला पायथॉन इंटरप्रिटर सुरू करायला आणि कोड चालवायला जमतेय.

१.३ पहिला प्रोग्राम

प्रोग्रामिंग जगतामध्ये कोणत्याही नवीन प्रोग्रामिंग-लँग्वेजमधल्या तुमच्या पहिल्या प्रोग्रामला 'Hello, World!' असे म्हणतात कारण तो प्रोग्राम फक्त 'Hello, World!' हे शब्द दाखवतो. पायथॉनमध्ये तो असा दिसतो:

```
>>> print('Hello, World!')
```

हे **प्रिंट स्टेटमेंट** (**print statement**) चे उदाहरण आहे, पण त्याने कोणत्याही कागदावर काही प्रिंट होत नाही. उत्तर स्क्रीनवर दिसते. ह्या उदाहरणात उत्तर आहे:

```
Hello, World!
```

प्रोग्राममधली अवतरण चिन्हे उत्तराच्या मजकुराची सुरुवात आणि शेवट खुणावतात, ती उत्तरात दिसत नाहीत.

कंस दाखवतात की `print` हे एक फंक्शन (function) आहे. आपण फंक्शनबद्दल सविस्तर चर्चा प्रकरण ३ मध्ये करणार आहोत.

पायथॉन २ मध्ये प्रिंट स्टेटमेंट थोडे वेगळे आहे, ते फंक्शन नाहीये म्हणून तिथे कंस वापरले जात नाहीत.

```
>>> print 'Hello, World!'
```

हा फरक का हे लवकरच कळेल, पण आतापुरते इतके ठीक आहे.

१.४ अंकगणितीय (Arithmetic) ऑपरेटर

'Hello, World' नंतरची पायरी म्हणजे अंकगणित. बेरीज आणि गुणाकार ह्यांसारखे गणन करण्यासाठी पायथॉनमध्ये **ऑपरेटर** (**operator**) चिन्हे असतात.

+, –, आणि * ऑपरेटर्स खालील उदाहरणाप्रमाणे बेरीज, वजाबाकी, आणि गुणाकार करतात:

```
>>> 40 + 2
42
>>> 43 - 1
42
>>> 6 * 7
42
```

/ ऑपरेटर भागाकार करतो:

```
>>> 84 / 2
42.0
```

तुम्हाला थोडीशी शंका आली असेल की उत्तर 42 च्या ऐवजी `42.0` का आहे. पुढच्या विभागात त्याचे कारण कळेल.

शेवटी ** ऑपरेटर घातांकन करतो, उदा., ६ वर्ग अधिक ६ ($6^2 + 6$):

```
>>> 6**2 + 6
42
```

बाकी काही लँग्वेजमध्ये, ^ चिन्ह घातांकांसाठी वापरले जाते पण पायथॉनमध्ये त्याला XOR (एक्सॉर) बिटवाइझ ऑपरेटर (bitwise operator) म्हणतात. तुम्हाला त्यांची माहिती नसेल तर ह्या उत्तराने तुम्ही चकित व्हाल:

```
>>> 6 ^ 2
4
```

ह्या पुस्तकामध्ये बिटवाइझ ऑपरेटर विषयी माहिती दिलेली नाही पण तुम्ही त्याविषयी पुढील लिंकवर वाचू शकता http://wiki.python.org/moin/BitwiseOperators.

१.५ व्हॅल्यू आणि टाइप (Values and types)

व्हॅल्यू (**value**) ही प्रोग्राममधली एक मूलभूत गोष्ट आहे, उदा., एखादे अक्षर किंवा एक संख्या. आतापर्यंत आपण बघितलेल्या काही व्हॅल्यूझ म्हणजे `2`, `42.0`, आणि `'Hello, World!'`.

ह्या व्हॅल्यूझ वेगवेगळ्या **टाइप** (**type**) च्या असतात `2` ही **इंटिजर** (**integer**) आहे, `42.0` ही **फ्लोटिंग-पॉइंट संख्या** (**floating-point number**) आहे, आणि `'Hello, World!'` ही **स्ट्रिंग** (**string**) आहे कारण त्यात अक्षरे स्ट्रिंग (दोरी) ने बांधल्यासारखी एकत्र आहेत.

जर तुम्हाला नक्की माहीत नसेल की एखाद्या व्हॅल्यूचा टाइप काय आहे, तर तुम्हाला ते इंटरप्रिटर सांगू शकतो:

```
>>> type(2)
<class 'int'>
>>> type(42.0)
<class 'float'>
>>> type('Hello, World!')
<class 'str'>
```

ह्या उत्तरांमध्ये 'class' हा शब्द वर्ग ह्या अर्थाने वापरला आहे, म्हणजे आपण व्हॅल्यूंचे क्लासमध्ये वर्गीकरण करतो. टाइप हा व्हॅल्यूंचा वर्ग आहे.

तुम्ही ओळखलेच असेल की इंटिजर चा टाइप `int` आहे, स्ट्रिंगचा टाइप `str` आहे, आणि फ्लोटिंग-पॉइंट संख्येचा टाइप `float` आहे.

पण `'2'` आणि `'42.0'` सारख्या व्हॅल्यूझ-चे काय? संख्यांसारख्या दिसतात पण त्या स्ट्रिंगसारख्या अवतरण चिन्हांमध्ये आहेत.

```
>>> type('2')
<class 'str'>
>>> type('42.0')
<class 'str'>
```

त्या स्ट्रिंगच आहेत.

तुम्ही पूर्णांक संख्या लिहिताना जर स्वल्पविराम वापरला, उदा., 1,000,000, तर पायथॉनमध्ये ती वैध इंटिजर नाही, पण ती व्हॅल्यू वैध आहे:

```
>>> 1,000,000
(1, 0, 0)
```

हे तर आपल्याला अपेक्षितच नव्हते! पायथॉन इंटरप्रिटर `1,000,000` ला स्वल्पविराम वापरून दिलेली यादी (sequence, सीक्वेन्स, मालिका) समजतो. आपण अशा याद्यांविषयी पुढे शिकणारच आहोत.

१.६ तर्कशुद्ध आणि नैसर्गिक भाषा (Formal and natural languages)

लोक ज्या भाषा बोलतात त्यांना **नैसर्गिक भाषा** म्हणतात, उदा., मराठी, इंग्रजी, जर्मन. त्यांची निर्मिती ही लोकांनी नाही केली तर त्यांची नैसर्गिक उत्क्रांती झाली, पण यथावकाश विद्वान लोकांनी त्यांचा खूप अभ्यास केलेला आहे.

ज्यांची निर्मिती काही विशिष्ठ अभ्यास, विज्ञान, किंवा व्यवहारासाठी केली गेली त्यांना **तर्कशुद्ध भाषा** (**formal languages**) म्हणतात. उदा., गणितज्ञ जी परिभाषा वापरतात ती एक तर्कशुद्ध भाषा असून ती संख्या आणि चिन्हे ह्यांतले सखोल संबंध दर्शवण्यासाठी अतिशय प्रभावी आहे. रसायनशास्त्रज्ञ हे रासायनिक रचना आणि रेणू दर्शवण्यासाठी (त्यांची वेगळी अशी) तर्कशुद्ध भाषा वापरतात. आणि सर्वांत महत्त्वाचे म्हणजे:

> **प्रोग्रामिंग लँग्वेजेस (programming languages) ह्या तर्कशुद्ध भाषा असून त्यांची रचना ही गणना कशी करावी हे व्यक्त करण्यासाठी केली गेली आहे.**

तर्कशुद्ध भाषांच्या **मांडणीचे नियम** (**syntax rule**) काटेकोर असतात जे त्या भाषांमधील विधानांचे रचना स्पष्ट करतात. उदा., गणितात $3 + 3 = 6$ ही रचना बरोबर आहे पण $3+ = 3\$6$ ही नाही. रसायनशास्त्रामध्ये H_2O हे एक रचनात्मकरित्या बरोबर सूत्र आहे पण ${}_2Zz$ हे नाही.

मांडणीचे नियम हे दोन प्रकारचे असतात. पहिल्या प्रकारचे नियम **टोकन** आणि रचनेशी (**token** and structure) संबंधित आहेत. भाषेचे मूलभूत घटक म्हणजे टोकन, उदा., शब्द, संख्या, आणि रासायनिक घटक (chemical elements). तर $3+ = 3\$6$ शी संबंधित एक मुख्य मुद्दा असा आहे की $ एक वैध टोकन नाहीये. त्याचप्रमाणे ${}_2Zz$ हे वैध नाही कारण Zz नावाचा कोणताच रासायनिक घटक नाही.

दुसऱ्या प्रकारचे मांडणीचे नियम टोकन्स एकत्रित कशी केली जातात ह्याच्याशी संबंधित आहेत. तर $3 + /3$ हे समीकरण अवैध आहे कारण जरी $+$ आणि $/$ ही वैध टोकन्स असली तरी आपण ती एकानंतर एक लिहू शकत नाही. त्याचप्रमाणे, रासायनिक सूत्रामध्ये सबस्क्रिप्ट (subscript[२]) हे रासायनिक घटकाच्या नंतर येते, आधी नाही.

हे एक @ व्यवस्थिपणे मांडलेले मरा$ी वाक्य आहे, पण त्यात अवैध टो*कन्स आहेत. हे वाक्य मराठी व्यवस्थितपणे मांडलेले आहे रचना अवैध, त्याची पण आहे. (अनुवादकाची टिप्पणी: आपल्या मराठीत शब्दांची कितीही फेरफार केली तरी अर्थ लागतो, त्यामुळे अशी चुकीची वाक्यरचना बनवण्यास मला थोडे झगडावे लागले हे खरे.)

जेव्हा तुम्ही मराठी वाक्य किंवा तर्कशुद्ध भाषेतील विधान वाचता, तेव्हा तुम्हाला रचना समजून घ्यावी लागते (अलबत एका नैसर्गिक भाषेत तुमच्या नकळतच हे तुमच्या मेंदूत होते). ह्या क्रियेला **पार्सिंग** (**parsing**) म्हणतात.

तर्कशुद्ध आणि नैसर्गिक भाषांचे अनेक गुणधर्म जरी सारखे असले, उदा., टोकन, रचना, आणि मांडणी-नियम, तरी त्यांत काही फरक आहेत:

अर्थाबाबत संदिग्धता: नैसर्गिक भाषांमध्ये विपुल प्रमाणात द्वयर्थीपणा आढळून येतो. लोक तो दूर करण्यासाठी संदर्भाकडे लक्ष देतात. नैसर्गिक भाषांची निर्मिती करताना त्यांना अशा संदिग्धतेपासून मुक्त ठेवण्याची काळजी घेतली जाते, म्हणजेच, प्रत्येक विधानाचा कोणत्याही संदर्भात एक आणि एकच अर्थ लावला जाऊ शकतो.

अनावश्यक भर: संदिग्धता घालवण्यासाठी आणि गैरसमज टाळण्यासाठी नैसर्गिक भाषा भरपूर अनावश्यक भर घालतात. त्यामुळे त्यांत शब्दांचा फापटपसारा आढळून येतो (उदा., कायदेशीर मजकूर). त्याउलट तर्कशुद्ध भाषा ह्या संक्षिप्त असतात.

तंतोतंतपणा: नैसर्गिक भाषांमध्ये वाक्प्रचार आणि अलंकार ह्यांचा खूप वापर होतो. समजा मी म्हणालो 'ट्युबलाइट पेटली' तर तुमच्या लक्षात येईल की खरे तर कोणतीच ट्युबलाइट नाहीये आणि काहीच पेटलेले नाहीये (ह्या वाक्प्रचाराचा अर्थ असा आहे की कोणाला तरी खूप वेळाने काहीतरी लक्षात आले जे आधीच यायला पाहिजे होते). तर्कशुद्ध भाषांचा अर्थ नेहमी शब्दशःच असतो.

आपण लहानपणीपासून नैसर्गिक भाषेत बोलत आलो असल्यामुळे कधीकधी आपल्याला तर्कशुद्ध भाषा अंगवळणी पडायला अवघड जाऊ शकते. तर्कशुद्ध भाषा आणि नैसर्गिक भाषा ह्यांमधला फरक हा काव्य आणि गद्य ह्यांच्यातल्या फरकासारखा किंवा कदाचित त्याहूनही जास्ती आहे:

काव्य: शब्द हे त्यांच्या अर्थासाठी आणि ध्वनीसाठी पण वापरले जातात, आणि संपूर्ण कविता एकत्रितरित्या एक कल्पना किंवा भावनिक प्रतिसाद निर्माण करते. संदिग्धता ही सामान्यपणे फक्त आढळूनच येत नाही तर ती सहेतुक वापरली जाते.

गद्य: शब्दांचा अक्षरशः अर्थ जास्ती महत्त्वाचा असतो, आणि अर्थाला रचनेचे जास्ती योगदान असते. गद्याचे विश्लेषण हे कवितेच्या विश्लेषणापेक्षा सोपे असले तरी गद्यातसुद्धा संदिग्धता असू शकते.

प्रोग्राम: कॉंप्युटर प्रोग्रामचा अर्थ हा निःसंदिग्ध आणि शब्दशः असतो, आणि टोकन्स आणि रचनेच्या विश्लेषणाने पूर्णपणे समजला जाऊ शकतो.

अनुवादकाची टिप्पणी: इंग्रजीमध्ये, काव्य, गद्य, आणि प्रोग्राम्स (poetry, prose, and programs) ह्यांचा छान अनुप्रास होता त्याचे lost in translation झाले.

[२] म्हणजे असे चिन्ह जे लहान असते आणि मुख्य ओळीच्या खाली असते, उदा., H_2O मध्ये 2 हे चिन्ह सबस्क्रिप्ट आहे.

तर्कशुद्ध भाषा ह्या नैसर्गिक भाषांपेक्षा जास्ती संक्षिप्त आणि जड असतात त्यामुळे त्यांतील मजकूर वाचायला आणि समजून घ्यायला वेळ लागतो. आणि त्यांची रचना महत्त्वाची असल्यामुळे वरपासून खालपर्यंत आणि डावीकडून उजवीकडे असे वाचणे नेहमी उचित ठरत नाही. त्याऐवजी तुमच्या डोक्यात प्रोग्राम पार्स (parse) करण्यास शिका, टोकन्स कोणती आहेत ते ओळखा, आणि रचना समजून घ्या. ह्या बाबींचा तपशील तर महत्त्वाचा आहेच. स्पेलिंग आणि विरामचिन्हांच्या छोट्या चुका ज्या नैसर्गिक भाषांमध्ये खपवून घेतल्या जातात त्यांमुळे तर्कशुद्ध भाषांमध्ये अर्थात किंवा समजण्यात जमीन अस्मानाचा फरक होऊ शकतो.

१.७ डीबगिंग (Debugging)

प्रोग्रामर्स चुका करतात. काही विचित्र कारणांमुळे प्रोग्राम मधल्या चुकांना **बग** (**bug**) असे म्हणतात आणि बग शोधून काढण्याच्या प्रक्रियेला **डीबगिंग** (**debugging**) असे म्हणतात.

प्रोग्रामिंग आणि विशेषतः डीबगिंग कधीकधी भावना जागृत करतात. जर तुम्ही एखाद्या अवघड बगशी झगडत असाल तर तुम्हाला राग येऊ शकतो, तुम्ही निराश होऊ शकता, किंवा तुम्हाला स्वतःची लाज वाटू शकते.

लोक कॉंप्युटरशी असे वागतात की तो एक व्यक्तीच आहे. जेव्हा तो नीट काम करतो तेव्हा तो आपल्याला आपला सोबती वाटतो, आणि जेव्हा तो हट्टी किंवा उद्धटासारखा वागतो तेव्हा आपण त्याच्याशी तसेच वागतो जसे आपण हट्टी आणि उद्धटासारखे वागणाऱ्या लोकांशी वागतो (Reeves and Nass, *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*).

असल्या प्रतिसादाची शक्यता गृहीत धरून तयारी केल्याने तुम्हाला त्याच्याशी लढायला मदत होईल. एक मार्ग म्हणजे असा विचार करणे की कॉंप्युटर आपल्यासाठी काम करणारा कर्मचारी असून त्याच्याकडे काही चांगले गुण आहेत उदा., गती आणि अचूकता, आणि काही कमतरता आहेत, जसे सहानुभूतीचा आणि जागरूकतेचा अभाव.

तुमचे काम एक चांगला मॅनेजर असणे हे आहे: चांगल्या गुणांचा फायदा घ्या आणि कमतरतांमुळे अडचणी होणार नाहीत ह्याची काळजी घ्या. आणि तुमच्या भावनांचा आधार घेऊन तुमच्या काम करण्याच्या क्षमतेवर परिणाम होऊ न देता प्रॉब्लेममध्ये गुंग व्हा.

डीबग करण्यास शिकणे हे निराशाजनक ठरू शकते, पण ते एक मोलाचे कौशल्य आहे ज्याचा उपयोग प्रोग्रामिंगच्याही पलीकडे तुम्हाला होईल. प्रत्येक प्रकरणाच्या शेवटी डीबगिंगवर एक विभाग आहे, ह्या विभागासारखाच, ज्यात काही टिपा आहेत. तुम्हाला त्यांचा उपयोग होईल अशी आशा आहे.

१.८ शब्दार्थ

प्रॉब्लेम-सॉल्व्हिंग (problem solving): खरा प्रॉब्लेम काय आहे हे उकलण्याची, तो सोडवण्याची, आणि उत्तर व्यक्त करण्याची क्रिया.

हाय-लेवल लँग्वेज (high-level language): पायथॉनसारखी प्रोग्रामिंग लँग्वेज जी लोकांना कॉंप्युटरशी संवाद साधण्यासाठी सोपी पडावी म्हणून बनवली गेली आहे.

लो-लेवल लँग्वेज (low-level language): कॉंप्युटरला चालवण्यास (एक्सेक्युट करण्यास) सोपी असणारी प्रोग्रामिंग लँग्वेज जिला 'असेंब्ली लँग्वेज' ('assembly language') किंवा 'मशीन लँग्वेज' ('machine language') असेही म्हणतात.

पोर्टेबिलिटी (portability): प्रोग्रामचा वेगवेगळ्या प्रकारच्या कॉंप्युटर्सवर चालण्याचा गुणधर्म.

इंटरप्रिटर (interpreter): असा प्रोग्राम जो दुसरा प्रोग्राम वाचून त्याला चालवतो.

प्रॉम्प्ट (prompt): आपले इनपुट घ्यायची तयारी दर्शवणारी इंटरप्रिटरने दाखवलेली अक्षरे (उदा., >>>).

प्रोग्राम (program): सूचनांची यादी जी एखादे गणन कसे करावे ते सांगते.

प्रिंट स्टेटमेंट (print statement): अशी सूचना जी पायथॉन इंटरप्रिटरला एखादी व्हॅल्यू दाखवायला सांगते.

ऑपरेटर (operator): एक विशेष चिन्ह जे साधे गणन दर्शवते, उदा., बेरीज, गुणाकार, किंवा स्ट्रिंग जोडणी.

व्हॅल्यू (value): डेटाचे (data) मूलभूत एकक ज्यावर प्रोग्राम क्रिया करतो. उदा., संख्या किंवा स्ट्रिंग.

टाइप (type): व्हॅल्यूझ-चे ज्यांच्यात विभाजन केले जाते ते वर्ग. आपण पुढील प्रकारचे टाइप बघितलेले आहेत: इंटिजर (type `int`), फ्लोटिंग-पॉइंट संख्या (type `float`), आणि स्ट्रिंग (type `str`).

इंटिजर (integer): पूर्णांक संख्या (..., -३, -२, -१, ०, १, २, ३, ...) दर्शवणारा टाइप.

फ्लोटिंग-पॉइंट (floating-point): ज्यांना अपूर्णांक भागसुद्धा असतो अशा संख्या दर्शवणारा टाइप.

स्ट्रिंग (string): अक्षरांची यादी दर्शवणारा टाइप.

नैसर्गिक भाषा (natural language): अशी भाषा जी लोक बोलतात आणि जिची नैसर्गिकपणे उत्क्रांती झाली आहे.

तर्कशुद्ध भाषा (formal language): अशी भाषा जिची लोकांनी विशिष्ठ कारणास्तव निर्मिती केली आहे, उदा., गणितातील कल्पना किंवा कॉंप्युटर प्रोग्राम व्यक्त करण्यास. सर्व प्रोग्रामिंग लँग्वेजेस ह्या तर्कशुद्ध भाषा आहेत.

टोकन (token): प्रोग्रामच्या रचनेतील एक मूलभूत एकक, जसे नैसर्गिक भाषेतील एकक म्हणजे शब्द.

मांडणी-नियम (syntax rules): प्रोग्रामच्या रचनेचे नियम.

पार्स (parse): प्रोग्रामचे मांडणी-नियमांनुसार विश्लेषण करणे.

बग (bug): प्रोग्राममधली चूक किंवा बिघाड.

डीबगिंग (debugging): बग हुडकून दुरूस्त करण्याची प्रक्रिया.

१.९ प्रश्नसंच (Exercises)

प्रश्न १.१. *वाचता वाचता उदाहरणांचा सराव करता येण्यासाठी हे पुस्तक कॉंप्युटरसमोर बसून वाचणेच उत्तम.*

काही नवीन गोष्ट शिकत असताना मुद्दाम चुका करून पहा. उदा., 'Hello, world!' प्रोग्राममध्ये दोनपैकी एखादे अवतरण चिन्ह नाही दिले तर काय होईल? दोन्ही नाही दिलीत तर काय होईल? जर `print` चे स्पेलिंग चुकीचे लिहिले तर?

वाचलेले लक्षात राहण्यासाठी ह्या प्रकारचे प्रयोग अतिशय उपयोगी ठरतात. प्रोग्रामिंग करताना सुद्धा त्यांचा फायदा होतो कारण तुमचा एरर मेसेजेसशी (error messages) परिचय होतो आणि त्यांचा अर्थ कळतो. नंतर आणि चुकून चुका करण्यापेक्षा आता आणि मुद्दाम चुका केलेले चांगले, नाही का.

१. *एका प्रिंट विधानामधून जर एक कंस काढला किंवा दोन्ही कंस काढले तर काय होते?*

२. *एक स्ट्रिंग प्रिंट करताना जर एक अवतरण चिन्ह काढले किंवा दोन्ही अवतरण चिन्हे काढली तर काय होते?*

३. *तुम्ही वजा (ऋण) चिन्ह वापरून ऋण संख्या दर्शवू शकता, उदा., -2. समजा अधिक (धन) चिन्ह एखाद्या संख्येच्या आधी लावले तर काय होते? आणि 2++2?*

४. *गणितात, सुरुवातीचे शून्य चालतात, उदा., 09. हे पायथॉनमध्ये चालेल का? आणि 011?*

५. *दोन व्हॅल्यूझ-च्यामध्ये समजा ऑपरेटर नसेल तर काय होते?*

प्रश्न १.२. *पायथॉन इंटरप्रिटर चालू करून त्याचा कॅल्क्युलेटर (calculator) म्हणून वापर करा.*

१. *४२ मिनिटे आणि ४२ सेकंद म्हणजे एकूण किती सेकंद?*

२. *१० किलोमीटर म्हणजे किती मैल? (टीप: एक मैल म्हणजे १.६१ मैल.)*

३. *जर तुम्हाला १० किलोमिटरची शर्यत ४२ मिनिटे आणि ४२ सेकंदांमध्ये पूर्ण करायची असेल तर तुमची सरासरी गती (प्रत्येक मैलाला लागणारा वेळ मिनीट आणि सेकंदांमध्ये) काय असावी लागेल? तुमची सरासरी गती किती मैल प्रति तास असावी लागेल?*

प्रकरण २

व्हेरिएबल, एक्स्प्रेशन, आणि स्टेटमेंट (Variable, expression, and statement)

एखाद्या प्रोग्रामिंग लँग्वेजचे अतिशय महत्त्वाचे वैशिष्ट्य म्हणजे **व्हेरिएबल** हाताळण्याची क्षमता. व्हेरिएबल म्हणजे एक नाव जे एक विशिष्ठ व्हॅल्यू दर्शवते.

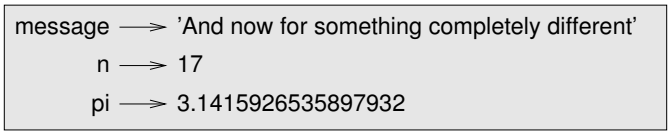
२.१ असाइनमेंट स्टेटमेंट (assignment statement, नियुक्ती विधान)

असाइनमेंट स्टेटमेंट एक नवीन व्हेरिएबल बनवते आणि त्याला एक व्हॅल्यू देते.

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897932
```

ह्या उदाहरणात तीन असाइनमेंट्स आहेत. पहिली एका स्ट्रिंगची नियुक्ती `message` नावाच्या नवीन व्हेरिएबलवर करते, दुसरी `n` ला `17` देते, आणि तिसरी π ची (अंदाजे) किंमत `pi` ह्या व्हेरिएबलला असाइन करते.

सामान्यपणे कागदावर व्हेरिएबल दाखवताना नाव आणि नावापासून व्हेरिएबलच्या व्हॅल्यूकडे जाणारा बाण दाखवतात. अशा आकृतीला **स्टेट डायग्राम** (state diagram, सद्यःस्थिती-आकृती) असे म्हणतात, कारण ती प्रत्येक व्हेरिएबलची सध्याची स्थिती म्हणजेच त्यातील व्हॅल्यू दाखवते. तुम्ही असा विचार करा की व्हेरिएबल म्हणजे एक खुर्ची आहे जिच्यावर त्या व्हेरिएबलचे नाव लिहिलेले आहे, व्हॅल्यू म्हणजे त्या खुर्चीवर बसलेली व्यक्ती, आणि स्टेट डायग्राम म्हणजे सगळ्या खुर्च्यांची सद्यःस्थिती दाखवणारी आकृती. आकृती २.१ वरच्या उदाहरणाची स्टेट डायग्राम दाखवते.



आकृती २.१: State diagram.

२.२ व्हेरिएबलचे नाव

प्रोग्रामर्स सामान्यपणे व्हेरिएबलसाठी अर्थपूर्ण नाव निवडतात जेणेकरून ते व्हेरिएबल कसे वापरले जाणार आहे हे स्पष्ट होईल.

व्हेरिएबलचे नाव कितीही मोठे असू शकते. त्यात अक्षरे आणि अंक असू शकतात, पण त्यांची सुरुवात ही अंकाने होऊ शकत नाही. कॅपिटल अक्षरे (उदा., A, B, C, ...) वापरणे जरी वैध असले तरी फक्त स्मॉलच (उदा., a, b, c, ...) वापरण्याची पद्धत आहे.

अंडरस्कोर (underscore), म्हणजेच _ हे नावात कुठेही लावले जाऊ शकते. अनेक शब्द असलेल्या नावात ते वापरले जाते, उदा., `your_name` किंवा `airspeed_of_unladen_swallow`.

व्हेरिएबलला चुकीचे नाव दिल्यास सिंटॅक्स एरर (syntax error) येतो.

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

`76trombones` अवैध आहे कारण ते अंकाने सुरू होते. `more@` अवैध आहे कारण त्यात अवैध अक्षर `@` आहे. पण `class` का चूक आहे?

खरे तर `class` हा पायथॉनचा एक **कीवर्ड** (**keyword**) आहे. इंटरप्रिटर प्रोग्रामची रचना जाणून घेण्यासाठी अनेक कीवर्ड्स वापरतो आणि त्यामुळे त्यांचा वापर व्हेरिएबलचे नाव म्हणून नाही केला जाऊ शकत.

पायथॉन ३ मध्ये हे कीवर्ड्स आहेत:

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

काळजी करू नका, तुम्हाला ही यादी पाठ करायची काहीच गरज नाही. तुम्ही प्रोग्राम सामान्यपणे ज्या सॉफ्टवेअरमध्ये लिहिणार आहात त्यात कीवर्ड वेगळ्या रंगात दाखवला जातो. म्हणजेच जर तुमच्या एखाद्या व्हेरिएबलचे नाव कोणता कीवर्ड असेल तर तुम्हाला लक्षात येईलच.

२.३ एक्स्प्रेशन आणि स्टेटमेंट (Expression and statement)

एक्स्प्रेशन हे काही व्हॅल्यूझ, व्हेरिएबल्स, आणि ऑपरेटर्स ह्यांच्या एकत्रीकरणाने बनते. केवळ एक व्हॅल्यू एक एक्स्प्रेशन मानली जाते आणि केवळ एक व्हेरिएबलसुद्धा, म्हणजेच खालील सगळी एक्स्प्रेशन्स वैध आहेत.

```
>>> 42
42
>>> n
17
>>> n + 25
42
```

जेव्हा तुम्ही एखादे एक्स्प्रेशन प्रॉम्प्टवर लिहिता, तेव्हा इंटरप्रिटर त्या एक्स्प्रेशनला **इव्हॅल्यूएट** (**evaluate**) करतो, म्हणजेच तो त्या एक्स्प्रेशनची व्हॅल्यू शोधून काढतो. ह्या उदाहरणात `n` ची व्हॅल्यू 17 आहे आणि `n + 25` ची व्हॅल्यू 42 आहे.

एक **स्टेटमेंट** (**statement**) हे कोड-चे असे एकक आहे ज्याचा स्वतंत्रपणे परिणाम होतो, जसे एखादे व्हेरिएबल बनवणे किंवा एखादी व्हॅल्यू दाखवणे.

```
>>> n = 17
>>> print(n)
```

वर पहिली ओळ एक असाइनमेंट स्टेटमेंट असून त्याने `n` ला व्हॅल्यू दिली जाते. दुसरी ओळ ही एक प्रिंट स्टेटमेंट असून त्याने `n` ची व्हॅल्यू प्रिंट केली जाते.

जेव्हा तुम्ही एखादे स्टेटमेंट लिहून एंटर मारता, तेव्हा इंटरप्रिटर त्याला **एक्सेक्युट** (**execute**) करतो, म्हणजेच त्याला चालवतो, दुसऱ्या शब्दांत, तो स्टेटमेंट जे सांगेल ते करतो. साधारणपणे स्टेटमेंटला व्हॅल्यू नसतात.

२.४ स्क्रिप्ट मोड (Script mode)

आतापर्यंत आपण पायथॉन **इंटरॅक्टिव्ह मोड** (**interactive mode**) मध्ये चालवले, म्हणजे तुमचा संवाद थेट इंटरप्रिटरशी होत होता. इंटरॅक्टिव्ह मोडमध्ये शिकणे/प्रयोग-करणे सोयीस्कर असले तरी प्रोग्राममध्ये जास्ती ओळी असल्यावर अडचणी येऊ शकतात.

दुसरा पर्याय असा आहे की कोड एका फाइलमध्ये सेव्ह (save) केला जाऊ शकतो. अशा फाइल (file) ला **स्क्रिप्ट** (**script**) म्हणतात. नंतर मग इंटरप्रिटर **स्क्रिप्ट मोड** (**script mode**) मध्ये चालवून ती स्क्रिप्ट एक्सेक्युट करू शकतो. पायथॉन-स्क्रिप्टच्या नावांचा शेवट `.py` ने करण्याची पद्धत आहे.

जर तुम्हाला काँप्युटरवर स्क्रिप्ट बनवून चालवता येत असेल तर उत्तम नाहीतर PythonAnywhere वापरा. स्क्रिप्ट मोड चालवण्याच्या सूचना पुढील लिंकवर बघा: http://tinyurl.com/thinkpython2e.

पायथॉन दोन्ही मोड्स पुरवत असल्यामुळे तुम्ही कोड-चा काही भाग स्क्रिप्टमध्ये टाकण्याआधी त्याची चाचणी इंटरप्रिटरमध्ये करून बघू शकता. पण ह्या दोन्ही मोड्समध्ये काही फरक आहेत ज्यांनी तुम्ही संभ्रमात पडू शकता.

उदा., जर तुम्ही पायथॉनचा कॅल्क्युलेटर म्हणून वापर करत असाल, तर तुम्ही कदाचित लिहाल:

```
>>> miles = 26.2
>>> miles * 1.61
42.182
```

पहिली ओळ `miles` ला एक व्हॅल्यू असाइन करते पण त्याचा काही परिणाम आपल्याला दिसून येत नाही. दुसरी ओळ ही एक एक्स्प्रेशन आहे, तर इंटरप्रिटर त्याला इव्हॅल्यूएट करतो आणि उत्तर दाखवतो. तर आपल्याला ह्यावरून कळून येते की मॅरेथॉनची शर्यत अंदाजे ४२ किलोमीटरची असते.

पण तुम्ही हाच कोड एका स्क्रिप्टमध्ये टाकून ती स्क्रिप्ट चालवली तर तुम्हाला काहीच आउटपुट (output) मिळणार नाही. स्क्रिप्ट-मोडमध्ये एक्स्प्रेशनचा दिसणारा स्वतंत्र परिणाम होत नाही. पायथॉन एक्स्प्रेशन इव्हॅल्यूएट करतो पण उत्तर दाखवत नाही. ते उत्तर दाखवण्यासाठी तुम्हाला खालीलप्रमाणे प्रिंट (`print`) स्टेटमेंट द्यावे लागते:

```
miles = 26.2
print(miles * 1.61)
```

पायथॉनच्या ह्या वागण्याने सुरुवातीला तुम्ही संभ्रमात पडू शकता. तुम्हाला ही गोष्ट नीट समजली आहे ह्याची खात्री करण्यासाठी खालील स्टेटमेंट्स पायथॉन इंटरप्रिटरमध्ये टाका आणि पहा ती काय करतात:

```
5
x = 5
x + 1
```

आता हीच स्टेटमेंट्स एका स्क्रिप्टमध्ये टाकून ती चालवा. आउटपुट काय आहे? आता स्क्रिप्टमध्ये प्रत्येक एक्स्प्रेशनचे प्रिंट स्टेटमेंट बनवून तिला परत चालवा.

२.५ ऑपरेटर्सचा अनुक्रम (Order of operations)

जेव्हा एक्स्प्रेशनमध्ये एकाहून अधिक ऑपरेटर्स असतात तेव्हा त्यांनी दर्शवलेली ऑपरेशन्स कोणत्या क्रमाने करायची हे त्यातल्या ऑपरेटर्सवरून ठरते. त्या क्रमाला **ऑर्डर ऑफ ऑपरेशन्स** (**order of operations**) म्हणतात. गणितातल्या ऑपरेटरसाठी पायथॉन गणितातलीच पद्धत वापरतो. **PEMDAS** हा शॉर्टकट हे नियम लक्षात ठेवायला सोपा आहे:

- **P**arentheses (कंस) ह्यांना सर्वांत जास्ती प्राधान्य असून त्यांचा वापर कोणतेही एक्स्प्रेशन पाहिजे त्या क्रमाने इव्हॅल्यूएट करण्यास होतो. कंसांतील एक्स्प्रेशन्स सर्वांत आधी इव्हॅल्यूएट केली जातात म्हणून `2 * (3-1)` ह्याचे उत्तर 4 आहे, आणि `(1+1)**(5-2)` ह्याचे उत्तर 8 आहे. एखादे एक्स्प्रेशन वाचण्यास सोपे जावे म्हणूनसुद्धा तुम्ही कंस वापरू शकता, जसे `(minute * 100) / 60`, आणि ह्याने उत्तर बदलत नाही.

- **E**xponentiation (घातांक) ला पुढचे प्राधान्य आहे, म्हणजे `1 + 2**3` ह्याचे उत्तर 9 आहे, 27 नाही, आणि `2 * 3**2` ह्याचे उत्तर 18 आहे, 36 नाही.

- **M**ultiplication (गुणाकार) आणि **D**ivision (भागाकार) ह्यांना **A**ddition (बेरीज) आणि **S**ubtraction (वजाबाकी) च्यावर प्राधान्य आहे. म्हणजे `2*3-1` ह्याचे उत्तर 5 आहे, 4 नाही, आणि `6+4/2` ह्याचे उत्तर 8 आहे, 5 नाही.

- सारखेच प्राधान्य असलेले ऑपरेटर्स (घातांक सोडून) डावीकडून उजवीकडे इव्हॅल्यूएट केले जातात. म्हणजेच `degrees / 2 * pi` ह्या एक्स्प्रेशनमध्ये भागाकार आधी होतो आणि नंतर उत्तराला `pi` ने गुणले जाते. जर 2π ने भागायचे असेल तर तुम्ही कंस वापरू शकता किंवा `degrees / 2 / pi` असे लिहू शकता.

ऑपरेटर्सचा प्राधान्यक्रम लक्षात ठेवायला जास्ती महत्त्व देऊ नका. जर एक्स्प्रेशनकडे बघून पटकन ते लक्षात नाही आले, तर ते स्पष्ट करण्यासाठी कंसांचा वापर करणे रास्त ठरते.

२.६ स्ट्रिंगवरील प्रक्रिया (String operations)

साधारणपणे, आपण स्ट्रिंगवर गणितातील क्रिया नाही करू शकत, जरी ती स्ट्रिंग संख्यांसारखी दिसत असली तरी; म्हणजे खालील ऑपरेशन्स अवैध आहेत:

```
'chinese'-'food'    'eggs'/'easy'    'third'*'a charm'
```

पण ह्याठिकाणी दोन अपवाद आहेत, ते म्हणजे + and *.

पहिला + ऑपरेटर **स्ट्रिंग कन्कॅटनेशन** (**string concatenation**, स्ट्रिंग जोडणी) करतो, म्हणजेच तो अनेक स्ट्रिंग्सला एका पुढे एक जोडतो. उदा.,

```
>>> first = 'throat'
>>> second = 'warbler'
>>> first + second
throatwarbler
```

आणि, * ऑपरेटर पुनरावृत्ती करतो. उदा., `'Spam'*3` चे उत्तर `'SpamSpamSpam'` आहे. जर एक व्हॅल्यू स्ट्रिंग असेल तर दुसरी इंटिजर असली पाहिजे.

हे दोन ऑपरेटर्स, म्हणजे + आणि * ह्याठिकाणी काहीसे बेरजेच्या आणि गुणाकाराच्या संलग्नच चालतात. जसे `4*3` म्हणजेच `4+4+4`, तसेच `'Spam'*3` म्हणजेच `'Spam'+'Spam'+'Spam'`. दुसरीकडे, स्ट्रिंग कन्कॅटनेशन आणि पुनरावृत्ती हे संख्यांची बेरीज आणि गुणाकार ह्यांपेक्षा खूप वेगळे आहेत. तुम्ही संख्यांची बेरीज व स्ट्रिंग कन्कॅटनेशन मधील एखादा फरक सांगू शकता?

२.७ कॉमेंट (Comment)

प्रोग्राम जसजसा मोठा अणि गुंतागुंतीचा होत जातो तसतसे त्याला वाचून समजून घेणे अवघड होत जाते. तर्कशुद्ध भाषा ह्या संक्षिप्त असल्यामुळे भरपूरदा कोड-च्या एखाद्या भागाकडे बघून तो भाग नेमके काय (किंवा का) करतो आहे हे समजत नाही.

ह्या कारणास्तव तुमच्या प्रोग्राममध्ये तो काय करतोय हे सांगणाऱ्या नैसर्गिक भाषेतील टिप्पण्या मोलाच्या ठरतात. अशा टिप्पणीला **कॉमेंट** (**comment**) म्हणतात, आणि तिची सुरुवात # ह्या चिन्हाने होते.

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

वरील कॉमेंट स्वतंत्र ओळीवर आहे. तुम्ही ओळीच्या शेवटीसुद्धा एक कॉमेंट लिहू शकता.

```
percentage = (minute * 100) / 60     # percentage of an hour
```

कोणत्याही ओळीवर # च्या नंतर लिहिलेले सर्व ते सोडून दिले जाते: त्याचा प्रोग्रामच्या एक्सेक्युशनवर काहीच परिणाम होत नाही.

प्रोग्रामची उघडपणे *न* दिसणारी वैशिष्ट्ये नमूद करणाऱ्या कॉमेंट्सचा सर्वांत जास्त उपयोग होतो. प्रोग्राम *काय* करतोय हे वाचकाला समजेल ही वाजवी अपेक्षा आहे, म्हणून तो प्रोग्राम ते *का* करतोय हे सांगणे जास्त महत्त्वाचे.

उदा., खालील कॉमेंट ही दिलेला कोड बघता निरुपयोगी आहे:

```
v = 5     # assign 5 to v
```

पण ही पुढील कॉमेंट आपल्याला उपयोगी माहिती देते:

```
v = 5     # velocity in meters/second.
```

व्हेरिएबल्सची नावे चांगली ठेवली तर कॉमेंट्सची गरज कमी भासते, पण लांब नावांनी मोठी एक्स्प्रेशन्स समजायला कठीण होतात, म्हणजेच ह्याठिकाणी तोल साधावा लागतो.

२.८ डीबगिंग (Debugging)

प्रोग्राममध्ये तीन प्रकारचे एरर्स येऊ शकतात: सिंटॅक्स एरर्स, रनटाइम (runtime) एरर्स, आणि सिमँटिक (semantic) एरर्स. त्यांचा शोध घेण्यासाठी त्यांतला फरक समजून घेणे महत्त्वाचे आहे.

सिंटॅक्स एरर: सिंटॅक्स हा शब्द प्रोग्रामच्या रचनेशी आणि नियमांशी निगडीत आहे. उदा., कंस जोडीनेच आले पाहिजेत, म्हणजे `(1 + 2)` वैध आहे, पण `8)` हा **सिंटॅक्स एरर** (**syntax error**) आहे.

तुमच्या प्रोग्राममध्ये कुठेही सिंटॅक्स एरर असला तर पायथॉन एक एरर मेसेज दाखवतो आणि आपले काम संपवतो, म्हणजेच तुम्हाला प्रोग्राम चालवता येणार नाही. तुमच्या प्रोग्रामिंगच्या कारकीर्दी सुरुवातीच्या काही आठवड्यात तुम्हाला खूप सिंटॅक्स एरर्स मिळतील. जसजसा तुमचा अनुभव वाढेल, तसतसे तुम्ही कमी एरर्स कराल आणि त्यांची लवकर शिकार कराल.

रनटाइम एरर: ह्या दुसऱ्या प्रकारच्या एररचे रनटाइम एरर हे नाव पडण्याचे कारण म्हणजे तो एरर प्रोग्राम रन व्हायला सुरुवात होईपर्यंत दिसून येत नाही. ह्या एरर्सना **एक्सेप्शन** (**exception**) असेही म्हणतात कारण सहसा ते काही तरी exceptional (आणि वाईट) झालेले आहे असे दर्शवतात.

सुरुवातीच्या काही प्रकरणांमधील सोप्या प्रोग्राम्समध्ये तुम्हाला क्वचितच रनटाइम एरर्स आढळून येतील.

सिमँटिक एरर: ह्या तिसऱ्या प्रकारच्या एररचे नाव सिमँटिक (semantic) असे आहे कारण त्याचा संबंध प्रोग्रामच्या अर्थाशी आहे. जर प्रोग्राममध्ये सिमँटिक एरर असेल तर तो एरर मेसेज *न* दाखवता चालेल, पण तुम्हाला *जे* अपेक्षित आहे तो *ते* करणार नाही. तो काहीतरी भलतेच करेल. पण विशेष म्हणजे तो तुम्ही सांगितलेले तंतोतंतपणे करेल. (कोणाला सांगकाम्या बाळू माहीत आहे? कॉंप्युटर सांगकाम्या बाळूच आहे.)

एखाद्या सिमँटिक एररचा पेच लक्षात येण्यासाठी प्रोग्रामचे आउटपुट बघून तो काय करतो आहे ह्याचा माग घेऊन समजून घ्यावे लागते.

२.९ शब्दार्थ

व्हेरिएबल (variable): एक व्हॅल्यू दर्शवणारे नाव.

असाइनमेंट (assignment): व्हेरिएबलला व्हॅल्यू असाइन करणारे स्टेटमेंट.

स्टेट डायग्राम (state diagram): काही व्हेरिएबल्स आणि त्यांच्यात ठेवलेल्या व्हॅल्यूझ दर्शवणारी आकृती.

कीवर्ड (keyword): पायथॉनने प्रोग्राम पार्स करता यावा म्हणून राखून ठेवलेले शब्द. हे शब्द तुम्ही व्हेरिएबलचे नाव म्हणून वापरू शकत नाही. उदा., `if`, `def`, आणि `while`.

ऑपरँड (operand): ऑपरेटर जिच्यावर क्रिया करतो ती व्हॅल्यू.

एक्स्प्रेशन (expression): व्हॅल्यूझ, व्हेरिएबल्स, आणि ऑपरेटर्स ह्यांचे केलेले एकत्रीकरण ज्याचे एक आणि एकच उत्तर असते.

इव्हॅल्यूएट (evaluate): एक्स्प्रेशनमधील ऑपरेशन्स पार पाडून उत्तर मिळवणे.

स्टेटमेंट (statement): एक आज्ञा किंवा कृती दर्शवणारा कोड-चा भाग. आतापर्यंत आपण फक्त दोनच प्रकारची स्टेटमेंट्स पाहिली आहेत: असाइनमेंट आणि प्रिंट स्टेटमेंट.

एक्सेक्युट (execute): एखादे स्टेटमेंट चालवून (रन करून) ते सांगेल तसे करणे.

इंटरॅक्टिव्ह मोड (interactive mode): प्रॉम्प्टपाशी कोड लिहून पायथॉन इंटरप्रिटर वापरण्याची पद्धत.

स्क्रिप्ट मोड (script mode): एका स्क्रिप्ट (फाइल) मधून कोड वाचून चालवण्यासाठी पायथॉन इंटरप्रिटर वापरण्याची पद्धत.

स्क्रिप्ट (script): एका फाइलमध्ये ठेवलेला प्रोग्राम.

ऑपरेटर्सचा अनुक्रम (order of operations): अनेक ऑपरेटर्स असलेले एक्स्प्रेशन इव्हॅल्यूएट करताना कोणता क्रम वापरावा हे सांगणारे नियम.

कन्कॅटनेट (concatenate): दोन ऑपरँड एकानंतर एक जोडणे.

कॉमेंट (comment): प्रोग्राममधली ती माहिती जी कोड वाचणाऱ्यांसाठी असून जिचा प्रोग्रामच्या चालण्यावर काहीच परिणाम होत नाही.

सिंटॅक्स एरर (syntax) error: प्रोग्राममधला तो एरर ज्यामुळे प्रोग्राम पार्स करणे अशक्य होते.

एक्सेप्शन (exception): प्रोग्राम रन करताना आलेला एरर.

सिमँटिक्स (semantics): प्रोग्रामचा अर्थ.

सिमँटिक एरर (semantic error): असा एरर ज्यामुळे प्रोग्राम प्रोग्रामरच्या अपेक्षेहून वेगळेच काही तरी करतो.

२.१० प्रश्नसंच (Exercises)

प्रश्न २.१. *मागच्या प्रकरणातील सल्ल्याची पुनरावृत्ती: जेव्हा जेव्हा तुम्ही नवीन गोष्ट शिकता तेव्हा तेव्हा ती लगेच इंटरॅक्टिव्ह मोडमध्ये तपासून बघा आणि मुद्दाम एरर करून काय होते ते बघा.*

- *आपण पाहिले आहे की* `n = 42` *वैध आहे, तर* `42 = n` *वैध आहे? (अनुवादकाची टिप्पणी: ४२ ही पाश्चिमात्य पॉप कल्चरमध्ये विशेष स्थान असलेली एक संख्या आहे. पुढील लिंक बघा: https://en.wikipedia.org/wiki/42_(number)#The_Hitchhiker's_Guide_to_the_Galaxy.*
- *आणि* `x = y = 1`*?*

- *काही लँग्वेजेसमध्ये प्रत्येक स्टेटमेंटचा शेवट हा अर्धविरामाने, म्हणजेच ; ने करतात. पायथॉनमध्ये स्टेटमेंटच्या शेवटी असा अर्धविराम दिल्याने काय होते?*
- *आणि जर स्टेटमेंटच्या शेवटी पूर्णविराम दिला तर काय होते?*
- *गणितातील परिभाषेत आपण x आणि y ह्यांचा गुणाकार xy असा दर्शवतो. असे आपण पायथॉनमध्ये करून पाहिले तर काय होते?*

प्रश्न २.२. *पायथॉन इंटरप्रिटरचा कॅल्क्युलेटर म्हणून वापर करण्याचा सराव:*

१. *त्रिज्या r असलेल्या गोलाचे घनफळ पुढील सूत्राने दिले जाते: $\frac{4}{3}\pi r^3$. त्रिज्या 5 असलेल्या गोलाचे घनफळ किती?*

२. *एका पुस्तकाची छापील किंमत ₹२४९५ आहे, पण पुस्तक विक्रेत्यांना सगळ्या पुस्तकांवर ४० टक्के सूट मिळते. त्यांनी एक पुस्तक पोस्टाने मागवले तर त्याचे ₹३०० पडतात आणि पुढील प्रत्येक पुस्तकाचे प्रत्येकी ₹७५ पडतात. तर ६० पुस्तकांची एकूण किंमत किती?*

३. *मी सकाळी ६:५२ ला घरातून निघालो आणि ८मि:१५से प्रति मैलच्या सहज गतीने १ मैल धावलो, नंतर ३ मैल टेंपो गतीने म्हणजे ७मि:१२से प्रति मैलच्या गतीने धावलो, आणि त्यानंतर परत १ मैल आधीच्या सहज गतीने धावलो तर मी नाश्त्यासाठी घरी किती वाजता पोहोचेन?*

प्रकरण ३

फंक्शन (Function)

प्रोग्रामिंगच्या संदर्भात एक **फंक्शन** म्हणजे एका शीर्षकाखाली (नावाखाली) क्रमाने दिलेली काही स्टेटमेंट्स जी ठराविक गणन (computation, कॉंप्युटेशन) करतात. फंक्शन लिहिताना आपण त्याचे नाव आणि खाली क्रमाने त्यातील स्टेटमेंट्स लिहितो. नंतर आपण ते फंक्शन त्याच्या नावाने 'कॉल' करू शकतो.

३.१ फंक्शन कॉल (Function call)

आपण आधीच एका **फंक्शन कॉल**चे उदाहरण पाहिले आहे:

```
>>> type(42)
<class 'int'>
```

ह्याठिकाणी फंक्शनचे नाव `type` आहे. कंसांतल्या एक्स्प्रेशनला फंक्शनचे **अर्ग्युमेंट** (**argument**) म्हणतात. वरील फंक्शन (`type`) चे उत्तर म्हणजे अर्ग्युमेंटचा टाइप.

ह्या प्रक्रियेचे असे वर्णन करू शकतो: a function 'takes' an argument and 'returns' a result. मराठीत आपण ह्याला असे म्हणूया: एक फंक्शन अर्ग्युमेंट 'घेते' आणि रिझल्ट (उत्तर) 'रिटर्न' करते. रिझल्टला **रिटर्न व्हॅल्यू** (**return value**) म्हणतात.

पायथॉनमध्ये अशी काही फंक्शन्स आहेत जी एका व्हॅल्यूचे दुसऱ्या व्हॅल्यूत रुपांतर करतात. उदा., `int` फंक्शन कोणतीही व्हॅल्यू घेते आणि त्याचे इंटिजरमध्ये रुपांतर करते; पण फक्त शक्य असेल तर, नाहीतर ते फंक्शन एरर देते.

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int(): Hello
```

`int` हे फंक्शन फ्लोटिंग-पॉइंट (floating-point) व्हॅल्यूझ चे इंटिजरमध्ये रुपांतर करू शकते, पण त्याची बरोबर राउंडिंग *न* करता त्यातला अपूर्णांक काढून टाकते.

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

`float` हे फंक्शन इंटिजर आणि स्ट्रिंगचे फ्लोटिंग-पॉइंट मध्ये रुपांतर करते:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

आणि `str` हे फंक्शन अर्ग्युमेंटचे स्ट्रिंगमध्ये रुपांतर करते.

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

३.२ गणितीय फंक्शन (Math function)

पायथॉनमध्ये एक गणिताचे मोड्युल (module) आहे जे आपणास परिचित गणितीय फंक्शन्स उपलब्ध करून देते. **मोड्युल** म्हणजे एक फाइल ज्यात निगडीत फंक्शन्सचा साठा असतो.

पण एखाद्या मोड्युलमधील फंक्शन्स वापरण्याआधी आपल्याला ते मोड्युल आपल्या कोडच्या सान्निध्यात आणावे लागते आणि त्यासाठी आपण **इंपोर्ट स्टेटमेंट** (**import statement**) वापरतो.

```
>>> import math
```

हे स्टेटमेंट math नावाचे **मोड्युल ऑब्जेक्ट** (**module object**) तयार करते. मोड्युल ऑब्जेक्ट पाहिल्यावर त्याविषयी काही माहिती मिळते:

```
>>> math
<module 'math' (built-in)>
```

मोड्युल ऑब्जेक्टमध्ये संबंधित मोड्युल फाइल मधली फंक्शन्स आणि व्हेरिएबल्स असतात. त्यातले फंक्शन वापरण्यासाठी आपल्याला आधी मोड्युलचे नाव, नंतर एक बिंदू (dot, पूर्णविराम), आणि नंतर फंक्शनचे नाव असे त्या फंक्शनला संबोधावे लागते. ह्या पद्धतीला **डॉट नोटेशन** (**dot notation**) म्हणतात.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

सिग्नल-टू-नॉइझ डेसिबलमध्ये व्यक्त करण्याचे सूत्र खाली दिले आहे:

$$10 \log_{10} \left(\frac{P_{\text{signal}}}{P_{\text{noise}}} \right) .$$

अधिक माहितीसाठी पुढील लिंक बघा: https://en.wikipedia.org/wiki/Signal-to-noise_ratio#Decibels.

ह्याठिकाणी आपण पाया १० असलेला लॉग वापरतो; पहिल्या उदाहरणात वरचे सूत्र वापरून आपण सिग्नल-टू-नॉइझ डेसिबलमध्ये शोधून काढले आहे. ह्याची नोंद घ्या की आपण असे गृहीत धरून चाललो आहोत की `signal_power` आणि `noise_power` मध्ये P_{signal} आणि P_{noise} च्या बरोबर किंमती आहेत. (अनुवादकाची टिप्पणी: मी शाळेत असताना आम्ही नवनीतचे लॉग टेबल वापरायचो. आता पायथॉनमध्ये सहज काम होते.) हे `math` मोड्युल `log` नावाचे फंक्शन पुरवते जे नैसर्गिक लॉग शोधते, म्हणजेच e पाया असलेला लॉग. (पण e म्हणजे काय असा तुम्हाला प्रश्न पडला आहे का? तर पुढील लिंक बघा. https://en.wikipedia.org/wiki/E_(mathematical_constant).)

दुसऱ्या उदाहरणात त्रिकोणमितीतील sin हे फंक्शन वापरले आहे. इथे व्हेरिएबलच्या `radians` ह्या नावावरून तुम्हाला कल्पना आली असेल की math मोड्युल मधील कोणतेही त्रिकोणमितीय फंक्शन (उदा., sin, cos, tan) हे त्याचे अर्ग्युमेंट रेडियनमध्ये आहे असे गृहीत धरते. डिग्रीचे रेडियनमध्ये रुपांतर करण्यासाठी π ने गुणावे आणि १८० ने भागावे:

```
>>> degrees = 45
>>> radians = degrees / 180.0 * math.pi
>>> math.sin(radians)
0.707106781187
```

`math.pi` हे एक्स्प्रेशन आपण आधी डॉट नोटेशनबद्दल सांगितल्याप्रमाणे `math` मोड्युलमधील `pi` हे व्हेरिएबल दर्शवते. त्याच्या व्हॅल्यूचा टाइप हा फ्लोटिंग-पॉइंट असून त्याची व्हॅल्यू (किंमत) अंदाजे π आहे जी १५ अंकांपर्यंत अचूक असते.

तुमचे त्रिकोणमिती पक्के असेल तर तुम्ही वरचे उत्तर तपासण्यासाठी त्याची तुलना २ चे वर्गमूळ, भागिले २ शी करू शकता:

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

३.३ काँपझिशन (Composition)

आतापर्यंत आपण व्हेरिएबल, एक्स्प्रेशन, आणि स्टेटमेंट हे प्रोग्रामचे घटक स्वतंत्रपणे पाहिले, पण त्यांचा एकमेकांशी कसा संबंध येतो ह्यावर आपण काहीच चर्चा केली नाही.

प्रोग्रामिंग लँग्वेजेसचे एक छान वैशिष्ट्य म्हणजे आपण त्यांत छोटे छोटे भाग **कंपोझ** (**compose**) करू शकतो (इथे, कंपोझ करणे म्हणजे गुंफणे). उदा., एका फंक्शनचे अर्ग्युमेंट हे कोणत्याही प्रकारचे एक्स्प्रेशन असू शकते आणि त्या एक्स्प्रेशनमध्ये गणितीय ऑपरेटरचा वापर सुद्धा होऊ शकतो:

```
x = math.sin(degrees / 360.0 * 2 * math.pi)
```

आणि तुम्हाला विलक्षण वाटेल, पण अर्ग्युमेंट हे एक फंक्शन कॉलसुद्धा असू शकते:

```
x = math.exp(math.log(x+1))
```

साधारणपणे, जिथे जिथे तुम्ही व्हॅल्यू वापरू शकता तिथे तिथे तुम्ही कोणतेही एक्स्प्रेशनसुद्धा वापरू शकता; पण ह्याला एक अपवाद आहे, तो म्हणजे असाइनमेंट स्टेटमेंटच्या डाव्या बाजूला फक्त व्हेरिएबलचे नावच असले पाहिजे. दुसरे कोणतेही एक्स्प्रेशन डाव्या बाजूला असेल तर तो सिंटॅक्स एरर ठरतो (खरे तर ह्याला पण काही अपवाद आहेत, पण ते आपण नंतर बघू).

```
>>> minutes = hours * 60                # right
>>> hours * 60 = minutes                # wrong!
SyntaxError: can't assign to operator
```

३.४ नवीन फंक्शनची व्याख्या देणे

आतापर्यंत आपण पायथॉनबरोबर आलेली फंक्शन्सच वापरली, पण त्यात नवीन फंक्शनची भर घालणे शक्य आहे. नवीन फंक्शनचे नाव आणि त्याखाली ते फंक्शन कॉल केल्यावर कोणती स्टेटमेंट्स चालवली जावी त्यांची क्रमाने दिलेली यादी म्हणजेच **फंक्शन डेफनिशन** (**function definition**) होय. उदा.,

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")
```

ही एक फंक्शन डेफनिशन आहे हे `def` कीवर्ड दर्शवतो. ह्या फंक्शनचे नाव `print_lyrics` आहे (lyrics म्हणजे गाण्यांचे शब्द). फंक्शनच्या नावांना व्हेरिएबलच्या नावांना लागू होणारेच नियम लागू होतात. अक्षरे, अंक, आणि अंडरस्कोर वैध आहे, पण पहिले अक्षर हे अंक असू शकत नाही. कोणताही कीवर्ड फंक्शनचे नाव म्हणून वापरू शकत नाही. एखाद्या फंक्शनला आणि व्हेरिएबलला सारखेच नाव देणे टाळले पाहिजे, जरी तसा नियम नसला तरी.

नावानंतरचे रिकामे कंस हे दर्शवतात की हे फंक्शन कोणतेच अर्ग्युमेंट घेत नाही.

फंक्शन डेफनिशनच्या पहिल्या ओळीला **हेडर** (**header**) म्हणतात तर उर्वरीत भागाला **बॉडी** (**body**) म्हणतात. हेडरच्या शेवटी अपूर्णविराम (म्हणजे colon, : चिन्ह) द्यावा लागतो. आणि (महत्त्वाचे) बॉडीतल्या प्रत्येक ओळीच्या आधी सारखीच जागा सोडली पाहिजे. ही जागा ४ स्पेसेस ठेवण्याची पद्धत आहे. बॉडीमध्ये कितीही स्टेटमेंट्स असू शकतात.

वर प्रिंट स्टेटमेंटमधल्या स्ट्रिंग्स दुहेरी अवतरण चिन्हांत आहेत. एकेरी आणि दुहेरी अवतरण चिन्हे दोन्ही चालतात. साधारणपणे एकेरी अवतरण चिन्हे वापरण्याची पद्धत आहे पण ह्याठिकाणी (जे इंग्रजीमधील apostrophe आहे असे) एक एकेरी अवतरण चिन्ह स्ट्रिंगमध्येच असल्यामुळे त्या स्ट्रिंगभोवती दुहेरी अवतरण चिन्हे वापरली आहेत.

सगळी अवतरण चिन्हे (एकेरी असो वा दुहेरी) ही 'सरळ आणि उभी' असावीत जी साधारणपणे कीबोर्डमध्ये एंटरच्या बाजूला असतात (ती अशी दिसतात: ' आणि "). कृपया 'वळणदार अवतरण चिन्हे' जी तुम्हाला ह्या वाक्यात दिसताहेत ती वापरू नका, ती पायथॉनमध्ये अवैध आहेत (ती अशी दिसतात: ‘ ’ , आणि “ ”).

अजून एक महत्त्वाचे: जर तुम्ही इंटरॅक्टिव्ह मोडमध्ये फंक्शन डेफनिशन टाइप करत असाल तर तुमची फंक्शन डेफनिशन अजून चालू आहे हे खुणावण्यासाठी इंटरप्रिटर काही बिंदू दाखवतो:

```
>>> def print_lyrics():
...     print("I'm a lumberjack, and I'm okay.")
...     print("I sleep all night and I work all day.")
...
```

फंक्शन डेफनिशन संपवण्यासाठी तुम्हाला एक रिकामी ओळ सोडावी लागेल.

एका फंक्शनची डेफनिशन दिल्यावर पायथॉन एक **फंक्शन ऑब्जेक्ट** (**function object**) तयार करतो ज्याचा टाइप `function` हा असतो:

```
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> type(print_lyrics)
<class 'function'>
```

नवीन बनवलेले फंक्शन कॉल करण्याचा सिंटॅक्स हा पायथॉनचे फंक्शन कॉल करताना वापरल्या जाणाऱ्या सिंटॅक्ससारखाच आहे:

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

एखादे फंक्शन बनवल्यावर ते दुसऱ्या फंक्शनमध्ये पण वापरू शकतो. उदा., आधीच्या चरणाची पुनरावृत्ती करण्यासाठी आपण `repeat_lyrics` नावाचे फंक्शन लिहू शकतो:

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

आणि आता `repeat_lyrics` कॉल करा.

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

पण खरे तर गाणे तसे नाहीये.

३.५ डेफनिशन्स आणि वापर

मागच्या विभागातील कोड-च्या तुकड्यांना आपण एकत्र आणले तर पूर्ण प्रोग्राम असा दिसेल:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")

def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()
```

ह्या प्रोग्राममध्ये दोन फंक्शन डेफनिशन्स आहेत: `print_lyrics` आणि `repeat_lyrics`. फंक्शन डेफनिशन्सना पायथॉन एका स्टेटमेंटसारखेच चालवतो आणि त्याची परिणती फंक्शन ऑब्जेक्ट्सच्या निर्मितीत होते. पण त्या फंक्शनमधील स्टेटमेंट्स ते फंक्शन कॉल केल्याशिवाय चालवली जात नाहीत आणि त्याचमुळे फक्त फंक्शन डेफनिशनने आउटपुट दिसत नाही.

तुम्ही ओळखलेच असेल की, एखादे फंक्शन वापरण्याआधी ते बनवणे गरजेचे आहे. म्हणजेच फंक्शन डेफनिशन फंक्शन कॉल करायच्या आधी चालवली गेली पाहिजे.

एक प्रयोग म्हणून, ह्या प्रोग्रामची शेवटची ओळ सुरुवातीला हलवा म्हणजे फंक्शन कॉल डेफनिशनच्या अगोदर येईल. मग तो प्रोग्राम चालवा आणि बघा तुम्हाला काय एरर मेसेज दिसतो.

आता तो फंक्शन कॉल परत खाली आणा आणि `print_lyrics` ची डेफनिशन `repeat_lyrics` च्या डेफनिशनच्या खाली आणा. हा प्रोग्राम चालवल्यावर काय होते?

३.६ फ्लो-ऑफ-एक्सेक्युशन (Flow of execution)

एखाद्या फंक्शनची डेफनिशन त्याच्या पहिल्या वापराच्या आधी दिली आहे ह्याची खात्री करण्यासाठी तुम्हाला प्रोग्राममधील स्टेटमेंट्स कोणत्या क्रमाने चालणार आहेत हे जाणून घेतले पाहिजे; आणि स्टेटमेंट्सच्या ह्याच क्रमाला **फ्लो-ऑफ-एक्सेक्युशन** (**flow of execution**) म्हणतात.

एक्सेक्युशन नेहमी प्रोग्राममधल्या पहिल्या स्टेटमेंटपासून सुरू होते. स्टेटमेंट्स एकेक करून वरपासून खालपर्यंत चालवली जातात.

हा प्रवाह फंक्शन डेफनिशन्सने बदलत नाही पण हे लक्षात असू द्या की फंक्शनमधील स्टेटमेंट्स ते फंक्शन कॉल केल्याशिवाय चालवली जात नाहीत.

एक फंक्शन कॉल ह्या प्रवाहाच्या मार्गात फाटा फोडतो. पुढे जाण्याऐवजी प्रवाह त्या फंक्शनच्या बॉडीला जातो, तिथली स्टेटमेंट्स चालवतो आणि जिथून फाटा फुटला होता तिथे परततो.

हे जर तुम्हाला सोपे वाटत असले तर हे लक्षात असू द्या की एका फंक्शनमध्ये असताना दुसरे फंक्शन कॉल केले जाऊ शकते. एक फंक्शन चालवताना मधेच प्रोग्रामला दुसऱ्या फंक्शनची स्टेटमेंट्स चालवावी लागू शकतात. नंतर ते नवीन फंक्शन चालवता चालवता प्रोग्रामला अजून वेगळे फंक्शन चालवावे लागू शकते! म्हणजेच, एका फाट्यावरून दुसरा फाटा, आणि त्यावरून तिसरा, तो संपला की परत दुसऱ्यावर, जो संपल्यावर मूळ पहिल्यावर असे होऊ शकते.

सुदैवाने हे सगळे आपल्यासाठी पायथॉन करतो. पायथॉन सध्या प्रवाह कुठे आहे आणि सध्याचे फंक्शन संपल्यावर त्या प्रवाहाला परत (सध्याच्या फंक्शनला) ज्याने कॉल केले आहे त्या फंक्शनमध्ये कुठे न्यायचे आहे ह्याची सगळी माहिती व्यवस्थित ठेवतो. प्रोग्रामच्या शेवटी सगळी स्टेटमेंट्स चालवून झाल्यावर प्रवाह थांबतो.

निष्कर्ष असा की कोणताही प्रोग्राम वाचून समजून घेताना वरपासून खालपर्यंत वाचणे हे नेहमीच योग्य नसते. कधीकधी एक्सेक्युशनच्या प्रवाहाबरोबर जावे.

३.७ परॅमीटर आणि अर्ग्युमेंट (Parameter and argument)

आपण पाहिलेल्या काही फंक्शन्सना अर्ग्युमेंट्स द्यावी लागतात. उदा., `math.sin` कॉल करताना एक संख्या अर्ग्युमेंट म्हणून द्यावी लागते. काही फंक्शन्स एकाहून जास्ती अर्ग्युमेंट्स घेतात: `math.pow` दोन घेते, एक पाया (base) आणि एक घात (exponent).

फंक्शनच्यामध्ये, अर्ग्युमेंट ज्या व्हेरिएबलमध्ये पाठवले जाते त्या व्हेरिएबलला **परॅमीटर** (**parameter**) म्हणतात. खालील फंक्शन एक अर्ग्युमेंट घेते:

```
def print_twice(bruce):
    print(bruce)
    print(bruce)
```

वरील फंक्शनला पाठवलेले अर्ग्युमेंट `bruce` नावाच्या व्हेरिएबलमध्ये ठेवले जाते. कॉल केल्यावर हे फंक्शन दोन वेळा परॅमीटरची व्हॅल्यू प्रिंट करते.

प्रिंट करता येऊ शकणाऱ्या व्हॅल्यूला अर्ग्युमेंट म्हणून पाठवले, तरच हे फंक्शन चालेल (नाहीतर नाही चालणार).

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(42)
42
42
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

कॉंपझिशनचे नियम आपण बनवलेल्या (म्हणजे, प्रोग्रामर-परिभाषित) फंक्शनला पण लागू होतात, म्हणून `print_twice` कॉल करताना कोणतेही एक्स्प्रेशन अर्ग्युमेंट म्हणून वापरू शकतो:

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

फंक्शन कॉल सुरू होण्याअगोदरच अर्ग्युमेंट इव्हॅल्यूएट केले जाते, म्हणून वरच्या उदाहरणांमध्ये `'Spam '*4` आणि `math.cos(math.pi)` ह्यांचे इव्हॅल्यूएशन एकदाच होते.

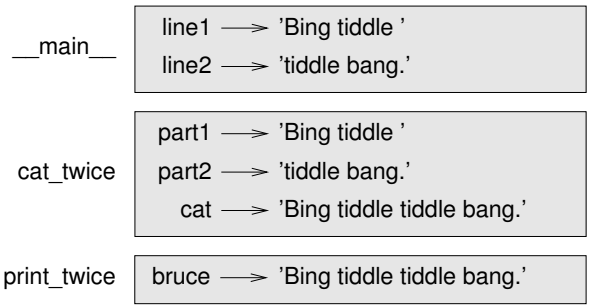
व्हेरिएबलसुद्धा अर्ग्युमेंट म्हणून वापरता येते. तर फंक्शन कॉलच्या वेळी त्या व्हेरिएबलमध्ये जी व्हॅल्यू आहे ती अर्ग्युमेंट म्हणून वापरली जाते:

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

जे व्हेरिएबल आपण अर्ग्युमेंटसारखे वापरतो (`michael`) त्याच्या नावाचा आणि परॅमीटरच्या नावाचा (`bruce`) काहीच संबंध नसतो. त्या व्हॅल्यूचे घरी काय नाव होते आपल्याला काही देणेघेणे नाही, `print_twice`मध्ये आम्ही सगळ्यांना `bruce` असेच संबोधतो.

३.८ व्हेरिएबलची आणि परॅमीटरची व्याप्ती स्थानिक असते

फंक्शनमध्ये बनवलेले व्हेरिएबल **स्थानिक** (**local**) असते, म्हणजेच त्याची व्याप्ती फक्त त्या फंक्शनपुरतीच मर्यादित असते. उदा.:



आकृती ३.१: स्टॅक डायग्राम.

```
def cat_twice(part1, part2):
    cat = part1 + part2
    print_twice(cat)
```

हे फंक्शन दोन अर्ग्युमेंट्स घेते, त्यांना जोडते, आणि उत्तर दोन वेळा प्रिंट करते. खालील उदाहरणात हे फंक्शन वापरले आहे:

```
>>> line1 = 'Bing tiddle '
>>> line2 = 'tiddle bang.'
>>> cat_twice(line1, line2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

जेव्हा `cat_twice` संपते (terminate होते) तेव्हा `cat` व्हेरिएबल नष्ट होते. आपण जर ते प्रिंट करण्याचा प्रयत्न केला तर आपल्यावर एक एक्सेप्शन फेकले जाते.

```
>>> print(cat)
NameError: name 'cat' is not defined
```

त्याचप्रमाणे परॅमीटरसुद्धा स्थानिक असतो. उदा., `print_twice` च्या बाहेर `bruce` नावाची कोणतीच गोष्ट नाही.

३.९ स्टॅक डायग्राम (Stack diagram)

कोणती व्हेरिएबल्स कोणत्या फंक्शन्समध्ये वापरली जात आहेत ह्याची नोंद ठेवण्यासाठी **स्टॅक डायग्राम** (**stack diagram**) उपयोगी पडते. स्टेट डायग्राम सारखीच स्टॅक डायग्राम प्रत्येक व्हेरिएबलची व्हॅल्यू दाखवते पण ती प्रत्येक व्हेरिएबल कोणत्या फंक्शनचे आहे हे पण दाखवते.

प्रत्येक फंक्शन हे एका **फ्रेम**ने (**frame**) दर्शवले जाते. फ्रेम म्हणजे विशेष काही नसून एक आयताकृती बॉक्स (ठोकळा) असतो, त्याच्या बाजूला फंक्शनचे नाव लिहिलेले असते आणि त्या फंक्शनच्या परॅमीटर्स आणि व्हेरिएबल्सची नावे बॉक्समध्ये लिहिलेली असतात. वरच्या उदाहरणाची स्टॅक डायग्राम आकृती ३.१ मध्ये दाखवली आहे.

आकृतीत फ्रेम्स एकाखाली एक रचण्यामागे एक कारण आहे. ती मांडणी हे दाखवते की कोणत्या फंक्शनने कोणते फंक्शन कॉल केले, आणि त्यापुढच्या फंक्शनने कोणते फंक्शन कॉल केले, आणि हे असेच पुढे. ह्या उदाहरणात `cat_twice` ने `print_twice` कॉल केले, आणि `__main__` ने `cat_twice` कॉल केले; सर्वांत वरच्या फ्रेमचे विशेष नाव `__main__` असते. तुम्ही कोणत्याही फंक्शनच्याबाहेर जर व्हेरिएबल बनवले तर ते `__main__` मध्ये राहते.

प्रत्येक परॅमीटर संबंधित अर्ग्युमेंट दर्शवतो. म्हणजे, `part1` आणि `line1` मध्ये **एकच** व्हॅल्यू आहे, `part2` आणि `line2` मध्ये एकच व्हॅल्यू आहे, आणि `bruce` आणि `cat`मध्ये एकच व्हॅल्यू आहे.

एखाद्या फंक्शन कॉलमध्ये एरर आल्यास पायथॉन त्या फंक्शनचे नाव दाखवतो, ते फंक्शन ज्याने कॉल केले त्याचे नाव दाखवतो, आणि ते फंक्शन ज्याने कॉल केले; आणि असे तो थेट `__main__` पर्यंत दाखवतो.

उदा., जर तुम्ही `print_twice`मध्ये `cat` वापरायचा प्रयत्न केला तर तुम्हाला `NameError` मिळेल:

```
Traceback (innermost last):
  File "test.py", line 13, in __main__
    cat_twice(line1, line2)
  File "test.py", line 5, in cat_twice
    print_twice(cat)
  File "test.py", line 9, in print_twice
    print(cat)
NameError: name 'cat' is not defined
```

फंक्शन्सच्या ह्या यादीला **ट्रेसबॅक** (**traceback**) म्हणतात. ह्या यादीमध्ये एखादा एरर कोणत्या फाइलमध्ये आला, कोणत्या ओळींवर आला, आणि त्या वेळी कोणती फंक्शन्स चालू होती ही माहिती असते.

ट्रेसबॅकमध्ये फंक्शन्सचा क्रम हा स्टॅक डायग्राममधल्या फंक्शन्सच्या क्रमासारखाच असतो: सध्या चालू असलेले फंक्शन सर्वांत खाली असते.

३.१० फलदायी फंक्शन आणि व्हॉय्ड (void) फंक्शन

आपण वापरलेली काही फंक्शन्स काहीतरी उत्तर पाठवतात (उदा., गणितीय फंक्शन्स), आपण त्यांना फलदायी फंक्शन्स म्हणूया. पण `print_twice` सारखी इतर फंक्शन्स काहीतरी कृती करतात पण कोणतेही उत्तर परत पाठवत नाहीत. त्यांना **व्हॉय्ड फंक्शन** (**void function**) म्हणतात.

जेव्हा तुम्ही एक फलदायी फंक्शन कॉल करता, तेव्हा बहुतांश वेळा तुम्हाला त्या फंक्शनचे उत्तर पुढे लागत असते. उदा., ते उत्तर एका व्हेरिएबलला असाइन करू शकता किंवा एका एक्स्प्रेशनमध्ये वापरू शकता:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

इंटरॅक्टिव्ह मोडमध्ये एखादे फंक्शन कॉल केल्यावर पायथॉन उत्तर दाखवतो:

```
>>> math.sqrt(5)
2.2360679774997898
```

पण स्क्रिप्टमध्ये एखादे फलदायी फंक्शन स्वतंत्रपणे कॉल केले तर त्याचे उत्तर कायमचे हरवून जाते!

```
math.sqrt(5)
```

ही स्क्रिप्ट ५ चे वर्गमूळ शोधते पण ती त्या उत्तराला कुठे ठेवत नाही आणि दाखवतही नाही म्हणून ती स्क्रिप्ट निरुपयोगी आहे.

व्हॉय्ड फंक्शन काहीतरी दाखवते किंवा दुसरे काही कार्य करते, पण त्याला रिटर्न व्हॅल्यू नसते. तुम्ही व्हॉय्ड फंक्शनचे उत्तर व्हेरिएबलला असाइन करायचा प्रयत्न केलात तर तुम्हाला `None` नावाची विशेष व्हॅल्यू मिळेल.

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print(result)
None
```

`None` व्हॅल्यू आणि `'None'` स्ट्रिंग ह्या दोन वेगळ्या गोष्टी आहेत. `None` एक विशेष व्हॅल्यू असून तिचा स्वत:चा असा टाइप आहे:

```
>>> type(None)
<class 'NoneType'>
```

आपण आतापर्यंत लिहिलेली सर्व फंक्शन्स व्हॉय्ड होती. काही प्रकरणांनंतर आपण फलदायी फंक्शन्स लिहायला सुरुवात करणार आहोत.

३.११ फंक्शन का?

फंक्शन वापरून प्रोग्रामचे छोटे-छोटे तुकडे करण्यामागचा हेतू काय हा प्रश्न तुम्हाला पडला असू शकतो. त्याची खूप कारणे आहेत:

- नवीन फंक्शन बनवण्याने स्टेटमेंट्सच्या एका गटाला नाव देता येते आणि त्यामुळे तुमचा प्रोग्राम समजायला सोपा होतो.
- कोड-ची पुनरावृत्ती टाळण्यासाठी फंक्शनचा उत्तम वापर होतो. नंतर तुम्हाला कोडमध्ये काही सुधार करावे लागले तर ते एकाच ठिकाणी करावे लागतात, शंभर ठिकाणी नाही.
- तुम्ही प्रोग्रामचे हे सुटे भाग एकेक करून डीबग करू शकता; त्यानंतर सर्व कोड असेंबल (एकत्र) करणे सोपे जाते.
- छान लिहिलेले फंक्शन कोड-च्या पुनर्वापरासाठी खूप फायदेशीर ठरते. एकदा लिहून डीबग केलेले फंक्शन परत वापरू शकतो.

३.१२ डीबगिंग

डीबगिंग हे महत्त्वाचे कौशल्य आहे. जरी डीबगिंग कधीकधी निराशाजनक आणि क्रोधदायी ठरू शकणारे असले तरी ते प्रोग्रामिंगचा अत्यंत महत्त्वाचा, सुरस, आणि अविरत भाग आहे.

डीबगिंग काही अंशी एका गुप्तहेराच्या कामासारखे आहे. तुम्हाला काही खुणा आणि पुरावे दिसतात, ज्या वापरून तुम्हाला कोणत्या घटना आणि प्रक्रियांनी ह्या गोष्टी झाल्या हे शोधून काढायचे असते.

डीबगिंग हे प्रायोगिक विज्ञानासारखे पण आहे. जेव्हा तुम्हाला काय चुकतेय आहे ह्याची थोडी कल्पना येते, तेव्हा प्रोग्राम सुधारून तुम्ही परत चालवून पाहता. जर तुमचे हायपॉथसिस (गृहीतक) बरोबर असले तर तुम्ही ह्या सुधारणेच्या परिणामाचे बरोबर भाकित करू शकता, आणि अशाप्रकारे तुम्ही बरोबर चालणाऱ्या प्रोग्रामकडे एक पायरी चढता. आणि जर तुमचे हायपॉथसिस चुकीचे असले तर तुम्हाला दुसरे शोधावे लागते. जसे शेरलॉक होम्स म्हणाला "When you have eliminated the impossible, whatever remains, however improbable, must be the truth." (आ. कोनन डॉय्ल, द *साइन ऑफ फोर*, *The Sign of Four*)

काही लोकांचे असे म्हणणे आहे की प्रोग्रामिंग आणि डीबगिंग ही एकच गोष्ट आहे. म्हणजे, प्रोग्रामिंग ही प्रोग्राम जोपर्यंत आपल्याला जे पाहिजे आहे ते करत नाही तोपर्यंत हळूहळू डीबग करण्याची क्रिया आहे. मुद्दा असा आहे की तुम्ही चालणारा (पण छोटा) प्रोग्राम घेऊन त्यात छोट्या सुधारणा घडवाव्यात; आणि त्या सुधारणा एकेक करून डीबग कराव्यात.

उदा., लिनक्स ही ऑपरेटिंग सिस्टम कोट्यवधी ओळींच्या कोड-ने बनलेली आहे, पण तिची सुरुवात एका छोट्याशा प्रोग्रामने झाली होती जो लायनस टोरवाल्ड्स ह्यांनी (Linus Torvalds) इंटेलच्या 80386 चिप (chip) वर फक्त प्रयोग म्हणून लिहिला होता. लॅरी ग्रीनफिल्ड म्हणाले: ''लायनसचा एक आधीचा प्रकल्प असा प्रोग्राम होता जो अदलून-बदलून AAAA आणि BBBB दाखवायचा. पुढे ह्याचीच लिनक्समध्ये उत्क्रांती झाली. (द *लिनक्स युझर्स गाइड* बीटा व्हर्जन १).

३.१३ शब्दार्थ

फंक्शन (function): एका शीर्षकाखाली (नावाखाली) क्रमाने दिलेली काही स्टेटमेंट्स जे उपयुक्त गणन करतात. फंक्शन काही अर्ग्युमेंट्स घेऊ शकते किंवा अर्ग्युमेंटशिवायही असू शकते. आणि ते काही उत्तर शोधून ते परत पाठवू शकते किंवा काहीही उत्तर *न* पाठवणारे पण असू शकते.

फंक्शन डेफनिशन (function definition): असे स्टेटमेंट जे नवीन फंक्शन तयार करते, ज्यात त्याचे नाव, पॅरॅमीटर्स, आणि त्या फंक्शनमधली सर्व स्टेटमेंट्स असतात.

फंक्शन ऑब्जेक्ट (function object): अशी व्हॅल्यू जी एका फंक्शन डेफनिशननंतर तयार होते. त्या फंक्शनचे नाव हे हा फंक्शन ऑब्जेक्ट दर्शवणारे व्हेरिएबल बनते.

हेडर (header): फंक्शन डेफनिशनची पहिली ओळ.

बॉडी (body): फंक्शन डेफनिशनमधली स्टेटमेंट्स.

परॅमीटर (parameter): फंक्शनला अर्ग्युमेंट म्हणून पाठवलेली व्हॅल्यू दर्शवण्यासाठी वापरलेले व्हेरिएबल.

फंक्शन कॉल (function call): असे स्टेटमेंट जे एखाद्या फंक्शनला रन करते. त्यात फंक्शनचे नाव असते आणि त्यानंतर कंसांत अर्ग्युमेंट्सची यादी असते.

अर्ग्युमेंट (argument): फंक्शन कॉल करताना त्याला पाठवलेली व्हॅल्यू. ही व्हॅल्यू संलग्न परॅमीटरला असाइन केली जाते.

स्थानिक व्हेरिएबल (local variable): फंक्शनमध्ये बनवलेले व्हेरिएबल. स्थानिक व्हेरिएबल फक्त त्याच्याच फंक्शनमध्ये वापरले जाऊ शकते.

रिटर्न व्हॅल्यू (return value): फंक्शनने शोधलेले उत्तर. जर फंक्शन कॉल एक्स्प्रेशन म्हणून वापरला असेल तर त्या फंक्शन कॉलची रिटर्न व्हॅल्यू ही त्या एक्स्प्रेशनची व्हॅल्यू बनते.

फलदायी फंक्शन (fruitful function): एक व्हॅल्यू रिटर्न करणारे फंक्शन.

व्हॉय्ड फंक्शन (void function): नेहमी `None` रिटर्न करणारे फंक्शन.

`None`**:** व्हॉय्ड फंक्शन्स रिटर्न करतात ती विशेष व्हॅल्यू.

मोड्युल (module): एकमेकांशी निगडीत फंक्शन्स आणि इतर डेफनिशन्स असणारी फाइल.

इंपोर्ट स्टेटमेंट (import statement): मोड्युल फाइल वाचून एक मोड्युल ऑब्जेक्ट बनवणारे स्टेटमेंट.

मोड्युल ऑब्जेक्ट (module object): अशी व्हॅल्यू जी एका `import` स्टेटमेंटने बनवली जाते जिच्यामार्फत मोड्युलमधल्या व्हॅल्यूझ वापरता येतात.

डॉट नोटेशन (dot notation): अन्य मोड्युलमधील फंक्शन कॉल करण्यासाठी वापरायचा सिंटॅक्स; ह्यात आधी मोड्युलचे नाव, मग एक बिंदू (पूर्णविराम), आणि मग त्या फंक्शनचे नाव लिहिले जाते.

कॉंपझिशन (composition): एखादे एक्स्प्रेशन एका मोठ्या एक्स्प्रेशनचा भाग म्हणून वापरणे, किंवा एखादे स्टेटमेंट एका मोठ्या स्टेटमेंटचा भाग म्हणून वापरणे.

फ्लो-ऑफ-एक्सेक्युशन (flow of execution, एक्सेक्युशनचा प्रवाह): ज्या क्रमाने स्टेटमेंट्स एक्सेक्युट (रन) केली जातात तो क्रम.

स्टॅक डायग्राम (stack diagram): फंक्शन्स, त्यांची व्हेरिएबल्स, आणि त्यांच्या व्हॅल्यूझ ह्यांच्या एकावर एक ठेवलेल्या मांडणीची आकृती.

फ्रेम (frame): स्टॅक डायग्राममधला एक बॉक्स जो एक फंक्शन कॉल दर्शवतो. त्यात त्या फंक्शनची स्थानिक व्हेरिएबल्स आणि परॅमीटर्स असतात.

ट्रेसबॅक (traceback): एखादे एक्सेप्शन आल्यावर पायथॉन तेव्हा जी फंक्शन्स चालवली जात होती त्यांची यादी दाखवतो, त्या यादीलाच ट्रेसबॅक म्हणतात.

३.१४ प्रश्नसंच (Exercises)

प्रश्न ३.१. `right_justify` *नावाचे फंक्शन लिहा जे एक* `s` *नावाची स्ट्रिंग परॅमीटर म्हणून घेते आणि ती स्ट्रिंग पुरेशा स्पेसेसनंतर प्रिंट करते, अशाप्रकारे की त्या स्ट्रिंगचे शेवटचे अक्षर हे डिस्प्लेच्या ७० व्या रकान्यात (कॉलममध्ये) असेल.*

```
>>> right_justify('monty')
                                                                  monty
```

टीप: स्ट्रिंग कन्कॅटनेशन आणि पुनरावृत्ती (repetition, रेपटीशन) वापरा. पायथॉन `len` *नावाचे फंक्शन पुरवते जे स्ट्रिंगची लांबी, म्हणजेच स्ट्रिंगमध्ये किती अक्षरे आहेत हे रिटर्न करते: उदा.,* `len('monty')` *ची व्हॅल्यू ५ आहे.*

प्रश्न ३.२. *फंक्शन ऑब्जेक्ट एक व्हॅल्यू असते जी तुम्ही एका व्हेरिएबलला असाइन करू शकता किंवा अर्ग्युमेंट म्हणून पाठवू शकता. उदा.,* `do_twice` *फंक्शन एक फंक्शन ऑब्जेक्ट अर्ग्युमेंट म्हणून घेते आणि त्यातील फंक्शन दोनदा कॉल करते:*

```
def do_twice(f):
    f()
    f()
```

खालील उदाहरणात `print_spam` *नावाचे फंक्शन दोनदा कॉल करण्यासाठी* `do_twice` *चा वापर केला आहे.*

```
def print_spam():
    print('spam')


do_twice(print_spam)
```

१. *हे उदाहरण एका स्क्रिप्टमध्ये लिहून तपासून बघा.*

२. `do_twice` *ला अशा प्रकारे सुधारा की ते*

   - *दोन अर्ग्युमेंट्स घेईल, एक फंक्शन ऑब्जेक्ट आणि एक व्हॅल्यू, आणि*
   - *अर्ग्युमेंट म्हणून घेतलेले फंक्शन अर्ग्युमेंट म्हणून घेतलेली व्हॅल्यू वापरून दोनदा कॉल करेल.*

३. *ह्या प्रकरणात आधी दिलेली* `print_twice` *ची डेफनिशन तुमच्या स्क्रिप्टमध्ये कॉपी करा.*

४. *तुमचे* `do_twice` *चे सुधारित स्वरूप वापरून* `'spam'` *हे अर्ग्युमेंट वापरून* `print_twice` *दोनदा कॉल करा.*

५. `do_four` *नावाचे नवीन फंक्शन द्या जे एक फंक्शन ऑब्जेक्ट आणि एक व्हॅल्यू घेते आणि त्या फंक्शन ऑब्जेक्टमधील फंक्शन ती व्हॅल्यू अर्ग्युमेंट म्हणून पाठवून चार वेळा कॉल करते. तुमच्या ह्या नवीन फंक्शनमध्ये फक्त दोन स्टेटमेंट्स असली पाहिजेत.*

उत्तर: http://thinkpython2.com/code/do_four.py.

प्रश्न ३.३. *नोंद: हा प्रश्न सोडवण्यासाठी आपण आतापर्यंत पाहिलेली स्टेटमेंट्स आणि माहिती ह्यांचाच वापर करावा.*

१. *खालील आकृती काढणारे फंक्शन लिहा:*

```
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
```

टीप: एकाहून अधिक व्हॅल्यू एकाच ओळीवर दाखवण्यासाठी तुम्ही स्वल्पविराम वापरून बनवलेली यादी वापरू शकता:

```
print('+', '-')
```

सामान्यपणे `print` *झाले की पुढच्या ओळीवर जाते, पण तुम्ही ते बदलू शकता. एका स्पेसने शेवट करण्यासाठी खालील पद्धत वापरा.*

```
print('+', end=' ')
print('-')
```

वरच्या स्टेटमेंट्सचे आउटपुट `'+ -'` *हे एकाच ओळीवर दिसेल. पण ह्यानंतरच्या प्रिंट स्टेटमेंटचे आउटपुट पुढच्या ओळीवर सुरू होईल.*

२. *वर २ बाय २ (2x2) ची ग्रिड (grid) दाखवली आहे. तशी पण ४ बाय ४ (4x4) ची ग्रिड दाखवणारे फंक्शन लिहा.*

उत्तर: http://thinkpython2.com/code/grid.py. आभार: हा प्रश्न Oualline, Practical C Programming, Third Edition*, O'Reilly Media, 1997 मधल्या एका प्रश्नावर आधारित आहे.*

प्रकरण ४

इंटरफेस डिझाइनची केस स्टडी
Case study: interface design

केस स्टडी म्हणजे एका विशिष्ठ पण पुरेशा साधारण प्रॉब्लेमवर केलेला अभ्यास होय. ह्या प्रकरणात आपण एकत्र काम करणारी अनेक फंक्शन्स कशी डिझाइन करायची हे दाखवणारी केस स्टडी बघणार आहोत.

ह्यात आपली `turtle` (टर्टल) मोड्युलशी ओळख होणार आहे, जे वापरून आपण टर्टल ग्राफिक्सद्वारे चित्रे काढू शकतो. `turtle` मोड्युल प्रत्येक पायथॉन इन्स्टॉलेशन (installation) मध्ये असते, पण तुम्ही पायथॉन ऑनलाइन वापरत असाल (उदा., PythonAnywhere किंवा repl.it) तर तुम्हाला टर्टल मोड्युल वापरता येणार नाही (निदान हे लिहिण्याच्या वेळी ते शक्य नव्हते). अनुवादकाची टिप्पणी: ह्या वेळी चटकन केलेल्या गूगल सर्चवरून मला पुढील वेबसाइट सापडली ज्यावर तुम्हाला टर्टल वापरणे शक्य आहे: https://trinket.io/turtle.

जर तुम्ही काँप्युटरवर पायथॉन इन्स्टॉल केले असेल किंवा https://trinket.io/turtle वापरू शकत असाल तर उत्तम, नाही तर आता पायथॉन इन्स्टॉल करा. पुढील लिंकवर सूचना आहेत http://tinyurl.com/thinkpython2e.

ह्या प्रकरणातील कोड-ची उदाहरणे पुढील लिंकवर आहेत http://thinkpython2.com/code/polygon.py.

४.१ टर्टल मोड्युल

तुमच्याकडे टर्टल मोड्युल आहे का हे तपासून बघण्यासाठी पायथॉन इंटरप्रिटर उघडा आणि लिहा:

```
>>> import turtle
>>> bob = turtle.Turtle()
```

हा कोड चालवल्यावर एक नवीन विंडो (window) उघडेल आणि त्यात एक छोटा बाण दिसेल जो टर्टल दर्शवतो. ती विंडो बंद करा.

`mypolygon.py` नावाची फाइल बनवा आणि खालील कोड लिहा:

```
import turtle
bob = turtle.Turtle()
print(bob)
turtle.mainloop()
```

`turtle` मोड्युल ('t' स्मॉल) एक `Turtle` ('T' कॅपिटल) नावाचे फंक्शन पुरवते जे एक Turtle ऑब्जेक्ट बनवते, ज्याला आपण `bob` नावाच्या व्हेरिएबलला असाइन केले आहे. जर `bob`ला प्रिंट केले तर असे दिसते:

```
<turtle.Turtle object at 0xb7bfbf4c>
```

दुसऱ्या शब्दांत: bob हे व्हेरिएबल Turtle टाइपचे ऑब्जेक्ट दर्शवते, आणि तो टाइप turtle मोड्युलमध्ये डिफाइन (define) केला आहे.

mainloop फंक्शन विंडो-ला सांगते की युझर (user) काही करेपर्यंत थांब, पण ह्याठिकाणी युझरकडे विंडो बंद करण्याशिवाय करण्याजोगे इतर काही नाही.

एकदा का Turtle बनवला की तुम्ही **मेथड** (**method**) वापरून त्याला विंडो-मध्ये फिरवू शकता. मेथड ही फंक्शनसारखीच असते पण तिचा सिंटॅक्स थोडा वेगळा असतो. उदा., टर्टलला पुढे करण्यासाठी:

```
bob.fd(100)
```

fd ही मेथड bob ह्या ऑब्जेक्टशी निगडीत आहे. एक मेथड कॉल करणे हे एक विनंती करण्यासारखे आहे: तुम्ही bob ला पुढे (forward) होण्यास सांगत आहात. fd चे अर्ग्युमेंट किती पिक्सेल्स (pixels) पुढे जायचे आहे ते दर्शवते. म्हणजे खरे अंतर तुमच्या डिस्प्ले (display) वर अवलंबून आहे.

Turtle वर कॉल करतायेण्याजोग्या इतर मेथड्स आहेत: मागे (back) जायला bk, डावीकडे (left) जायला lt, आणि उजवीकडे (right) जायला rt. नोंद: lt आणि rt किती अंश (डिग्रीझ) वळायचे आहे हे अर्ग्युमेंट म्हणून घेतात.

प्रत्येक Turtle एक पेन धरून आहे, जो एकतर खाली किंवा वर आहे; जर पेन खाली असला तर Turtle जाताना एक माग (वाट) सोडतो. pu आणि pd ह्या मेथड्स अनुक्रमे 'pen up' आणि 'pen down' करायला सांगतात.

काटकोन काढण्यासाठी, प्रोग्राममध्ये (bob बनवल्यानंतर आणि mainloop कॉल करायच्या आधी) खालील ओळी टाका:

```
bob.fd(100)
bob.lt(90)
bob.fd(100)
```

हा प्रोग्राम चालवल्यावर तुम्हाला bob पूर्वेकडे आणि नंतर उत्तरेकडे जाताना दिसेल आणि त्याने सोडलेले दोन रेषाखंड दिसतील.

आता हा प्रोग्राम सुधारून एक चौरस (square) काढा. हे जमल्याशिवाय पुढे जाऊ नका!

४.२ साधी पुनरावृत्ती (simple repetition)

बहुतेक तुम्ही असे लिहिले असेल:

```
bob.fd(100)
bob.lt(90)

bob.fd(100)
bob.lt(90)

bob.fd(100)
bob.lt(90)

bob.fd(100)
```

आपण हीच गोष्ट जास्त संक्षेपाने for स्टेटमेंट वापरून करू शकतो. खालील उदाहरणातील कोड mypolygon.py मध्ये टाकून ती स्क्रिप्ट परत चालवा:

```
for i in range(4):
    print('Hello!')
```

तुम्हाला असे काहीसे दिसेल:

```
Hello!
Hello!
Hello!
Hello!
```

हा `for` स्टेटमेंटचा सर्वांत सोपा वापर आहे; आपण पुढे सविस्तरपणे पाहूच. पण हे चौरस काढणाऱ्या प्रोग्रामला सुधारण्यासाठी पुरेसे आहे. ते केल्याशिवाय पुढे जाऊ नका.

खाली एक `for` स्टेटमेंट आहे जे एक चौरस काढते:

```
for i in range(4):
    bob.fd(100)
    bob.lt(90)
```

`for` स्टेटमेंटचा सिंटॅक्स हा फंक्शन डेफनिशनसारखाच आहे. त्याला एक हेडर असते ज्याचा शेवट अपूर्णविरामाने केला जातो आणि प्रत्येक ओळीच्या आधी सारखेच स्पेसेस सोडून लिहिलेली (indented) बॉडी. बॉडीमध्ये कितीही स्टेटमेंट्स असू शकतात.

`for` स्टेटमेंटला एक **लूप** (**loop**, शब्दशः अर्थ वळसा) असेही म्हणतात कारण फ्लो-ऑफ-एक्सेक्युशन बॉडीतून जाऊन परत वळसा घेऊन बॉडीच्या वर येतो. ह्याठिकाणी तो बॉडी चार वेळा चालवतो.

हा कोड आधीच्या चौरस-काढणाऱ्या कोड-पेक्षा थोडा वेगळा आहे कारण चौरसाची शेवटची बाजू रेखाटून झाल्यावर टर्टल अजून एकदा वळतो. ते जास्तीचे वळण घ्यायला जास्तीचा वेळ लागतो, पण एखाद्या गोष्टीची पुनरावृत्ती लूप-द्वारे केल्याने कोड वाचण्यास सोपा होतो. ह्या पद्धतीचा अजून फायदा असा की त्यानंतर टर्टल परत सुरुवातीच्या स्थानी आणि सुरुवातीच्या दिशेला तोंड करून उभा राहतो.

४.३ प्रश्नसंच (Exercises)

खाली `turtle` मोड्युलवर काही प्रश्न आहेत. ते फक्त मजेशीरच नाहीत तर त्यांच्याद्वारे शिकण्यासारखे काही मुद्देही आहेत. ते सोडवण्याचा प्रयत्न करताना मुद्दा काय असेल ह्याविषयी विचार जरूर करा.

ह्यानंतरच्या विभागांमध्ये उत्तरे असल्यामुळे प्रश्न सोडवल्याशिवाय (किंवा निदान नीट प्रयत्न केल्याशिवाय) पुढे बघू नका.

१. `square` नावाचे फंक्शन लिहा जे `t` नावाचा परॅमीटर घेते; `t` एक टर्टल आहे. हे फंक्शन टर्टल वापरून एक चौरस काढेल.

   एक फंक्शन कॉल लिहा जो `square` ला `bob` अर्ग्युमेंट म्हणून पाठवतो. हे झाल्यानंतर प्रोग्राम परत चालवा.

२. `square` मध्ये `length` नावाचा अजून एक परॅमीटर घाला. सर्व बाजूंची लांबी `length` एवढी असेल असा चौरस काढणारे `square`चे सुधारित रूप द्या. नंतर वरच्या मुद्द्यातला फंक्शन कॉलसुद्धा ह्या बदलास अनुसरून बदला. प्रोग्राम परत चालवा. वेगवेगळ्या लांब्या `length` मध्ये वापरून हा प्रोग्राम तपासा.

३. `square` ची एक कॉपी बनवा आणि तिचे नाव `polygon` ठेवा. त्यात `n` नावाचा अजून एक परॅमीटर टाका. आता `polygon`ची बॉडी बदला; अशाप्रकारे की ह्या बदलानंतर `polygon` फंक्शन n-बाजू असलेली नियमित बहुभुजाकृती (n-sided regular polygon) रेखाटेल. टीप: n-बाजू असलेल्या नियमित बहुभुजाकृतीचा प्रत्येक बाह्यकोन (exterior angle) $\frac{360}{n}$ अंश असतो. उदा., समभुज त्रिकोणातील आतले सर्व कोन 60 अंश असतात पण बाहेरचे सर्व कोन 120 अंश असतात (त्रिकोण म्हणजेच 3-बाजू असलेली नियमित बहुभुजाकृती).

४. `circle` नावाचे फंक्शन लिहा जे एक टर्टल `t` आणि त्रिज्या (radius) `r` परॅमीटर्स म्हणून घेते आणि एक अदमासे(अंदाजे)-वर्तुळ काढते. हे करण्यासाठी `polygon` वापरले जाईल. तुम्हाला `polygon` च्या अर्ग्युमेंट्ससाठी बाजूंची योग्य लांबी आणि किती बाजू वापरायच्या ते दर्शवणारी संख्या शोधून काढावी लागेल. वेगवेगळ्या त्रिज्या `r` मध्ये वापरून हे फंक्शन तपासा.

   टीप: वर्तुळाचा परिघ (circumference) शोधून काढा आणि `length * n = circumference` ह्या सूत्राचा वापर करा.

५. `circle` ची `arc` (वर्तुळकंस) नावाची एक थोडी जास्त व्यापक (more general) सुधारित आवृत्ती लिहा जी `angle` नावाचा अजून एक परॅमीटर घेते; `angle` आपल्याला हे सांगतो की वर्तुळाचा किती भाग काढायचा आहे. `angle` चे एकक अंश (degrees) हे आहे, म्हणजे जेव्हा `angle=360`, तेव्हा `arc` ने एक पूर्ण वर्तुळ काढले पाहिजे. आणि ह्याच कारणामुळे `arc`ला जास्त व्यापक म्हटले आहे. (`arc` सर्व वर्तुळे काढू शकते पण `circle` सर्व वर्तुळकंस नाही काढू शकत.)

४.४ एन्कॅप्सुलेशन (Encapsulation, विभागीकरण)

पहिल्या प्रश्नात तुम्ही चौरस-काढणाऱ्या कोड-चे फंक्शन डेफनिशनमध्ये रुपांतर केले आणि नंतर ते फंक्शन टर्टलला परॅमीटर म्हणून पाठवून कॉल केले. खाली त्या प्रश्नाचे उत्तर दिले आहे:

```
def square(t):
    for i in range(4):
        t.fd(100)
        t.lt(90)

square(bob)
```

सर्वांत आतल्या स्टेटमेंट्सच्या म्हणजेच `fd`च्या आणि `lt`च्या आधी तुम्हाला जास्ती जागा सोडलेली दिसेल, म्हणजेच ती दोन वेळा इंडेंट (indent) केली आहेत. त्याचे कारण म्हणजे हे दाखवणे की ती स्टेटमेंट्स ही `for` लूपच्या आत आहेत, जो फंक्शन डेफनिशनच्या आत आहे. त्याच्या पुढची ओळ डाव्या बाजूला चिकटून आहे, आणि हे `for` लूप आणि फंक्शन डेफनिशनह्या दोन्हींचा शेवट दर्शवते.

फंक्शनच्या आतमध्ये, `t` हे व्हेरिएबल `bob` दर्शवतो तेच टर्टल दर्शवते, म्हणजे `t.lt(90)`चा तोच परिणाम होतो जो `bob.lt(90)`चा होतो. तर तुम्हाला प्रश्न पडेल की, मग त्या परॅमीटरला `bob` हेच नाव दिलेले का चांगले नाही? मुद्दा असा आहे की `t` हे कोणतेही टर्टल असू शकते, फक्त `bob`च नाही, म्हणजे तुम्ही दुसरे टर्टल बनवून ते `square` ला अर्ग्युमेंट म्हणून पाठवू शकता:

```
alice = turtle.Turtle()
square(alice)
```

कोड-चा एक भाग एका फंक्शनमध्ये जमा करण्याला **एन्कॅप्सुलेशन** (**encapsulation, विभागीकरण**) म्हणतात. एन्कॅप्सुलेशनचा एक उपयोग असा की कोड-च्या त्या भागाला नाव देता येते, आणि त्या नावाचा फंक्शनविषयी संक्षिप्त माहिती देण्यासाठीही वापर होतो. दुसरा उपयोग असा की तुम्ही कोड-चा पुनर्वापर करू शकता. एक फंक्शन दोन वेळा कॉल करणे हे त्याची बॉडी दोन वेळा पेस्ट (paste) करण्यापेक्षा नक्कीच चांगले आहे.

४.५ व्यापकता (Generalization, जनरलायझेशन)

पुढची पायरी, `square`मध्ये `length` हा परॅमीटर टाकण्याची आहे. असे:

```
def square(t, length):
    for i in range(4):
        t.fd(length)
        t.lt(90)

square(bob, 100)
```

फंक्शनमध्ये नवीन परॅमीटर घालण्याला **जनरलायझेशन** (**generalization**) म्हणतात कारण त्याने फंक्शनची व्यापकता वाढते: आधी चौरस नेहमी १०० बाजू असलेला होता, आता त्याची बाजू कितीही लांब असू शकते.

पुढची पायरीसुद्धा जनरलायझेशनच आहे. `polygon` फंक्शन चौरस काढण्याऐवजी नियमित बहुभुजाकृती[१] (म्हणजेच regular polygon) काढते ज्याला चारच नाही तर कितीही बाजू असू शकतात. ते असे:

[१] मराठी विश्वकोशवर बहुभुजाकृतीविषयी माहिती:
https://vishwakosh.marathi.gov.in/28076/ आणि https://marathivishwakosh.org/52698/.

```
def polygon(t, n, length):
    angle = 360 / n
    for i in range(n):
        t.fd(length)
        t.lt(angle)

polygon(bob, 7, 70)
```

वरचे उदाहरण सात बाजू असलेली आणि प्रत्येक बाजूची लांबी ७० असलेली नियमित बहुभुजाकृती काढते.

पायथॉन २ मध्ये तुम्हाला `angle = 360/n` ऐवजी `angle = 360.0/n` वापरावे लागेल नाहीतर इंटिजर भागाकारामुळे `angle`ची किंमत थोडीशी चुकीची येऊ शकते. ह्याठिकाणी `angle = 360.0/n` मध्ये `360.0` फ्लोटिंग-पॉइंट असल्यामुळे `360.0/n` सुद्धा फ्लोटिंग-पॉइंट आहे. (अनुवादकाची टिप्पणी: आजकाल पायथॉन २ कोणीच वापरत नाही.)

जेव्हा फंक्शनमध्ये एकाहून धिक अर्ग्युमेंट्स असतात तेव्हा त्यांबद्दल किंवा त्यांच्या क्रमाबद्दल थोडा गोंधळ उडू शकतो. तेव्हा तुम्ही ते फंक्शन कॉल करताना परॅमीटर्सचे नाव घेऊन कॉल करू शकता:

```
polygon(bob, n=7, length=70)
```

ह्यांना एक विशेष नाव आहे **कीवर्ड अर्ग्युमेंट** (**keyword argument**) कारण ते परॅमीटरचे नाव 'कीवर्ड' ('keyword') म्हणून वापरते (ह्याचा `while` आणि `def` सारख्या पायथॉन कीवर्ड्सशी ह्यासंदर्भात काहीच संबंध नाही).

ह्या सिंटॅक्समुळे प्रोग्राम वाचायला सोपा होतो. ह्यामुळे अर्ग्युमेंट आणि परॅमीटर ह्यांचा संबंधही स्पष्ट होतो: फंक्शन कॉल केल्यावर अर्ग्युमेंट संलग्न परॅमीटरला असाइन होते.

४.६ इंटरफेस डिझाइन (Interface design)

पुढची पायरी `circle` फंक्शन लिहिणे ही आहे, जे त्रिज्या `r` परॅमीटर म्हणून घेते. खाली दिलेले उत्तर `polygon` फंक्शन वापरून ५० बाजू असलेली नियमित बहुभुजाकृती काढते:

```
import math

def circle(t, r):
    circumference = 2 * math.pi * r
    n = 50
    length = circumference / n
    polygon(t, n, length)
```

पहिली ओळ `r` त्रिज्या असलेल्या वर्तुळाचा परिघ $2\pi r$ हे सूत्र वापरून शोधते. आपण `math.pi` वापरत असल्यामुळे आपल्याला `math` इंपोर्ट करावे लागते. इंपोर्ट स्टेटमेंट स्क्रिप्टच्या सुरुवातीला ठेवण्याची पद्धत आहे.

आपले अदमासे(अंदाजे)-वर्तुळ `n` रेषाखंडांनी बनलेले आहे आणि प्रत्येक रेषाखंडाची लांबी `length` एकक आहे. अशाप्रकारे `polygon` फंक्शन ५० बाजूंची नियमित बहुभुजाकृती काढते जी अंदाजे `r` त्रिज्या असलेल्या वर्तुळासारखी दिसते.

ह्या उत्तराची कमतरता अशी आहे की `n` निश्चित (५०) असल्यामुळे खूप मोठ्या वर्तुळांचे रेषाखंड खूप लांब असतील आणि लहान वर्तुळांसाठी खूप लहान रेषाखंड काढण्यात आपला वेळ जाईल. एक मार्ग असा आहे की फंक्शनमध्ये `n` परॅमीटर टाकून त्याला अजून जनरलाइझ करणे. ह्याने युझर (`circle` फंक्शन कॉल करणारी) ला नियंत्रण मिळेल पण इंटरफेस थोडा मलिन होइल.

फंक्शनचा **इंटरफेस** (**interface**) म्हणजे फंक्शन कसे वापरायचे ह्याचा सारांश: परॅमीटर्स काय आहेत? फंक्शन काय करते? रिटर्न व्हॅल्यू काय आहे? अनावश्यक तपशील टाळून कॉल करणाऱ्याला जे पाहिजे आहे ते सोयीस्करपणे करू देणे हे एका 'व्यवस्थित' इंटरफेसचे लक्षण आहे.

ह्या उदाहरणात `r` हे इंटरफेसचा घटक आहे कारण ते कोणते वर्तुळ काढायचे आहे ते दर्शवते. `n` हा कमी योग्य घटक आहे कारण तो ते वर्तुळ *कसे* काढायचे ह्याचा तपशील आहे.

इंटरफेस घाण करण्यापेक्षा `circumference` वरून `n`ची किंमत ठरवणे जास्ती योग्य आहे:

```
def circle(t, r):
    circumference = 2 * math.pi * r
    n = int(circumference / 3) + 3
    length = circumference / n
    polygon(t, n, length)
```

आता रेषाखंडांची संख्या ही `circumference/3` च्या आसपास आहे, म्हणजेच प्रत्येक रेषाखंडाची लांबी अंदाजे ३ पिक्सेल्स आहे, जी इतकी लहान आहे की वर्तुळ चांगले दिसेल, आणि इतकी मोठी आहे की आपला प्रोग्राम कार्यक्षम राहील, आणि कोणत्याही मापाच्या वर्तुळासाठी ठीक असेल.

`n` मध्ये ३ मिळवण्याचे कारण हे की बहुभुजाकृतीमध्ये कमीतकमी ३ बाजू असाव्यात.

४.७ रिफॅक्टरिंग (Refactoring, पुनर्रचना)

`circle` फंक्शन लिहिताना आपण `polygon` फंक्शनचा पुनर्वापर करू शकलो कारण नियमित बहुभुजाकृती ही वर्तुळाचा चांगला अदमास (अंदाज) आहे. पण `arc` फंक्शन असे सहजासहजी जमणारे नाहीये; arc (वर्तुळकंस) काढण्यासाठी `polygon` किंवा `circle` चा उपयोग करू शकत नाही.

एक पर्याय असा आहे की `polygon`च्या एका कॉपीने सुरुवात करून तिचे `arc`मध्ये रुपांतर करणे. त्याचे उत्तर असे दिसेल:

```
def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = angle / n

    for i in range(n):
        t.fd(step_length)
        t.lt(step_angle)
```

ह्या फंक्शनचा खालचा अर्धा भाग `polygon` फंक्शन सारखाच दिसतोय; पण आपण `polygon` फंक्शनचा इंटरफेस बदलल्याशिवाय पुनर्वापर करू शकत नाही. त्यासाठी आपण `polygon` फंक्शनमध्ये कोन (angle) हा तिसरा परॅमीटर घालून जनरलाइझ करू शकतो, पण मग त्याचे `polygon` हे एक योग्य नाव राहणार नाही! त्याऐवजी आपण त्या जास्त व्यापक फंक्शनला `polyline` म्हणूया:

```
def polyline(t, n, length, angle):
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

आता आपण वरच्या बदलास अनुसरून `polygon` आणि `arc` ही फंक्शन्स बदलूया म्हणजे ती `polyline` वापरतील:

```
def polygon(t, n, length):
    angle = 360.0 / n
    polyline(t, n, length, angle)

def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
```

```
    step_length = arc_length / n
    step_angle = float(angle) / n
    polyline(t, n, step_length, step_angle)
```

शेवटी आपण `circle` फंक्शन बदलून त्यात `arc` फंक्शन वापरूया:

```
def circle(t, r):
    arc(t, r, 360)
```

इंटरफेस सुधारण्यासाठी आणि कोड-चा पुनर्वापर सोयीस्कर करण्यासाठी प्रोग्रामची पुनर्रचना करण्याच्या ह्या प्रक्रियेला **रिफॅक्टरिंग** (**refactoring, पुनर्रचना**) म्हणतात. आपण हे निरीक्षण केले की `arc` आणि `polygon` मध्ये सारखा कोड होता, तर आपण त्या सारख्या भागास `polyline` मध्ये घेऊन कोड-ची 'पुनर्रचना' केली (we 'factored it out' into `polyline`). आपण आधीच योजना केली असती तर `polyline` आधी लिहून रिफॅक्टरिंग टाळले असते, पण बहुतेक वेळा प्रोजेक्टच्या सुरुवातीला सर्व इंटरफेसेस डिझाइन करण्याइतकी माहिती नसते. एकदा का तुम्ही कोडिंग सुरू केले की तुम्हाला प्रॉब्लेम नीट समजतो. अनेकदा रिफॅक्टरिंग ही काहीतरी नवीन शिकल्याची निशाणी ठरते.

४.८ डेव्हेलपमेंट प्लान (A development plan, एक विकास योजना)

डेव्हेलपमेंट प्लान (**development plan**) ही प्रोग्राम लिहायची एक प्रक्रिया आहे. ह्या केस-स्टडीमध्ये आपण 'एन्कॅप्सुलेशन आणि जनरलायझेशन' ('encapsulation and generalization', म्हणजेच 'विभागीकरण आणि विस्तारवाढ') ची प्रक्रिया वापरली. ह्या प्रक्रियेच्या खालील पायऱ्या आहेत:

१. एका लहान आणि फंक्शन नसलेल्या प्रोग्रामने सुरुवात करा.

२. एकदा का प्रोग्राम चालायला लागला की त्यातला सुसंगत भाग शोधून काढा, त्याचे विभागीकरण करा, म्हणजेच त्या भागाला एका फंक्शनमध्ये एन्कॅप्सुलेट करा आणि नाव द्या.

३. योग्य परॅमीटर्स टाकून त्या फंक्शनची व्याप्ती वाढवा (त्याला जनरलाइझ करा).

४. पायऱ्या क्र. १–३ तोपर्यंत वारंवार करा जोपर्यंत तुमच्याकडे अनेक व्यवस्थित चालणारी फंक्शन्स होत नाहीत. चालणारा कोड कॉपी-पेस्ट (copy-paste) करून परत लिहिण्याचे आणि डीबग करण्याचे टाळा.

५. रिफॅक्टरिंगने प्रोग्राम सुधारण्याची संधी शोधा. उदा., एकसारखा कोड अनेक ठिकाणी दिसल्यास त्याची योग्य तितक्या साधारण (general) फंक्शनमध्ये पुनर्रचना करा (रिफॅक्टर करा).

ह्या प्रक्रियेच्या काही कमतरता आहेत—आपण पर्याय नंतर पाहणार आहोत—पण जर तुम्हाला विविध फंक्शन्समध्ये प्रोग्राम कसा विभागायचा ह्याची पूर्वकल्पना नसेल तर ही प्रक्रिया फायदेशीर ठरू शकते. ह्या मार्गाने तुम्ही पुढे जाता जाता डिझाइन करता.

४.९ डॉकस्ट्रिंग (docstring)

फंक्शनच्या सुरुवातीला इंटरफेस समजावून सांगणाऱ्या विशिष्ठ स्ट्रिंग-ला **डॉकस्ट्रिंग** (**docstring**) म्हणतात ('doc' हे 'documentation' संक्षिप्त स्वरूप आहे). उदा.:

```
def polyline(t, n, length, angle):
    """Draws n line segments with the given length and
    angle (in degrees) between them.  t is a turtle.
    """
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

सर्व डॉकस्ट्रिंग्स तिहेरी-अवतरणचिन्हांत (triple-quoted) देण्याची पद्धत आहे; ह्यांना मल्टीलाइन (multiline) स्ट्रिंग्स म्हणतात कारण अनेक ओळींवर पसरलेली स्ट्रिंग तिहेरी-अवतरणचिन्हे देऊन लिहिता येऊ शकते.

डॉकस्ट्रिंग संक्षिप्त जरी असली तरी त्यात कोणालाही फंक्शन वापरण्यासाठी लागणारी महत्त्वाची माहिती दिली जाते. तपशीलात *न* शिरता ती स्पष्टपणे फंक्शन काय करते हे सांगते. प्रत्येक परॅमीटरचा फंक्शनवर काय परिणाम होतो आणि प्रत्येक परॅमीटरचा टाइप काय आहे हेही ती सांगते (ते जर आधीपासून स्पष्ट नसेल तर).

अशा प्रकारच्या नोंदी (documentation, डॉक्युमेंटेशन) लिहिणे हे इंटरफेस-डिझाइनचा महत्त्वाचा भाग आहे. चांगले डिझाइन असलेला इंटरफेस समजावून सांगायला सोपा असला पाहिजे; जर तुमचेच फंक्शन समजावून सांगताना तुमचा गोंधळ होत असेल तर त्याचा निष्कर्ष म्हणजे इंटरफेस सुधारला पाहिजे.

४.१० डीबगिंग

इंटरफेस हा फंक्शन आणि ते फंक्शन कॉल करणारे ह्यांच्यामधला करार (contract) असतो असे म्हणता येईल. कॉल करणारे विशिष्ठ परॅमीटर्स पुरवण्याचे वचन देतात आणि सांगितलेले काम करण्याचे वचन फंक्शन देते.

उदा., `polyline`ला चार अर्ग्युमेंट्स लागतात: `t` अर्ग्युमेंट हे टर्टल असले पाहिजे; `n` अर्ग्युमेंट हे इंटिजर असले पाहिजे; `length` अर्ग्युमेंट हे धन संख्या असले पाहिजे; आणि `angle` अर्ग्युमेंट हे एक संख्या असले पाहिजे, जिचे एकक अंश (degree) असे समजले जाईल.

ह्या अटींना **प्रीकंडिशन्स** (**preconditions**) म्हणतात कारण फंक्शन एक्सेक्युट होण्याच्या आधीच त्यांची पूर्तता झाली पाहिजे. ह्याउलट, फंक्शनच्या शेवटी ज्यांची पूर्तता होते त्या अटींना **पोस्टकंडिशन्स** (**postconditions**) म्हणतात. ह्यांत योजलेला परिणाम (उदा., रेषाखंड काढणे) आणि इतर परिणाम (उदा., Turtle हलवणे किंवा इतर काही बदल करणे) ह्यांचा समावेश होतो.

प्रीकंडिशन्सची पूर्तता करणे ही कॉल करणाऱ्यांची जबाबदारी आहे. जर त्यांनी व्यवस्थितपणे नोंद (डॉक्युमेंट) केलेल्या प्रीकंडिशनचे उल्लंघन केले आणि फंक्शन नीट चालले नाही तर तो बग (bug) कॉलरचा (शर्टच्या collarचा नव्हे, फंक्शनच्या callerचा) आहे, फंक्शनचा नाही.

ह्याउलट जर प्रीकंडिशन्सचे समाधान होत असेल पण पोस्टकंडिशन्सचे नाही, तर तो बग फंक्शनचा आहे. जर तुमच्या प्री-आणि-पोस्टकंडिशन्स सुस्पष्ट असतील तर त्याचा डीबगिंगला नक्कीच उपयोग होतो.

४.११ शब्दार्थ

मेथड (method): ऑब्जेक्टशी संलग्न असलेले फंक्शन; हे कॉल करण्यासाठी डॉट नोटेशन वापरतात.

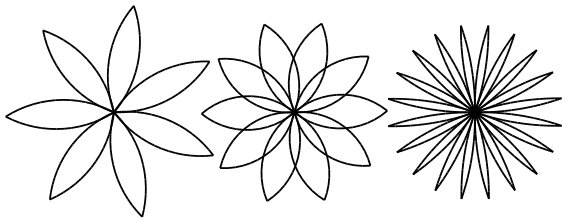
लूप (loop, वळसा): प्रोग्रामचा तो भाग जो वारंवार चालवला जातो (वळसा घालणे).

एन्कॅप्सुलेशन (encapsulation, विभागीकरण): स्टेटमेंट्सची एक यादी घेऊन तिचे फंक्शन-डेफनिशनमध्ये रुपांतर करणे.

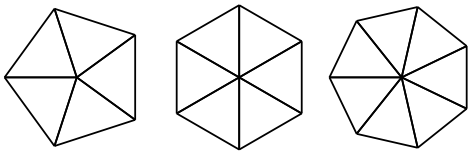
जनरलायझेशन (generalization, व्यापकपणा वाढवणे): अनावश्यकरित्या सीमित किंवा मर्यादित (specialized/limited) असलेला भाग काढून त्याला जास्त व्यापक भागाने बदलणे. उदा., वापरलेल्या स्थिर (constant) संख्येला व्हेरिएबल किंवा परॅमीटरने बदलणे, जसे आपण ५० ह्या स्थिर संख्येऐवजी `n` हा परॅमीटर वापरला होता.

कीवर्ड अर्ग्युमेंट (keyword argument): असे अर्ग्युमेंट ज्यात परॅमीटरचे नाव 'कीवर्ड' ('keyword') म्हणून वापरलेले असते.

इंटरफेस (interface): फंक्शन कसे वापरावे ह्याचे वर्णन; ह्यात अर्ग्युमेंट्स आणि रिटर्न व्हॅल्यू ह्यांची नावे आणि त्यांबद्दल लागणारी माहिती असते.



आकृती ४.१: टर्टलफुले.



आकृती ४.२: टर्टल पाय्झ (pies).

पुनर्रचना (refactoring): फंक्शन इंटरफेसेस सुधारण्यास आणि कोड-चा दर्जा उंचावण्यास कोड-मध्ये बदल करण्याची प्रक्रिया.

डेव्हेलपमेंट प्लान (development plan, विकास योजना): प्रोग्राम लिहिण्याची एक प्रक्रिया.

डॉकस्ट्रिंग (docstring): फंक्शन-डेफनिशनच्या सुरुवातीला दिलेली स्ट्रिंग जी फंक्शनच्या इंटरफेसबद्दलच्या माहितीची नोंद ठेवते.

प्रीकंडिशन (precondition): फंक्शन सुरू होण्याच्या आधी कॉल करणाऱ्यांनी पूर्ण करायची अट.

पोस्टकंडिशन (postcondition): फंक्शन पूर्ण होण्याच्या आधी फंक्शनने पूर्ण करायची अट.

४.१२ प्रश्नसंच (Exercises)

प्रश्न ४.१. *ह्या प्रकरणातील कोड पुढील लिंकवरून डाऊनलोड करा http://thinkpython2.com/code/polygon.py.*

१. `circle(bob, radius)` *एक्सेक्युट होत असतानाची स्थिती दाखवणारी स्टॅक डायग्राम काढा.*

२. *विभाग ४.७ मधील* `arc` *फंक्शन हे अचूक नाहीये कारण वर्तुळाचा त्या पद्धतीने केलेला रेखीय अंदाज (linear approximation) हा नेहमी त्या खऱ्या वर्तुळाच्या बाहेर असतो. त्यामुळे टर्टल जिथे पोहोचायला पाहिजे त्याच्या काही पिक्सेल्स दूर पोहोचतो. ह्या त्रुटीचा परिणाम कमी करण्याचा एक मार्ग दिलेला कोड (ज्याची लिंक वर दिली आहे तो) दाखवतो. तो कोड वाचून समजतो का ते बघा. तुम्ही जर आकृती काढून पाहिली तर तुम्हाला बहुतेक तो कोड कसा चालतो हे समजेल.*

प्रश्न ४.२. *आकृती ४.१ मध्ये दाखवल्याप्रमाणे फुले काढण्यासाठी व्यापकतेची योग्य पातळी असलेली फंक्शन्स (appropriately general set of functions) लिहा.*

उत्तर: http://thinkpython2.com/code/flower.py.

आणि पुढील लिंकवरील कोड-ही लागेल: http://thinkpython2.com/code/polygon.py.

प्रश्न ४.३. *आकृती ४.२ मध्ये दाखवलेले आकार (shapes) काढण्यासाठी व्यापकतेची योग्य पातळी असलेली फंक्शन्स (appropriately general set of functions) लिहा. उत्तर: http://thinkpython2.com/code/pie.py.*

प्रश्न ४.४. *इंग्रजी वर्णमालेतील अक्षरे काही मूलभूत आकारांनी (basic shapes नी) बनवता येऊ शकतात: उभ्या, आडव्या रेषा आणि काही वक्ररेषा (curves, arcs). कमीतकमी मूलभूत आकार वापरून काढता येतील अशी अक्षरे डिझाइन करा; आणि नंतर, ती काढण्यासाठी फंक्शन्स लिहा.*

ह्यात तुम्ही प्रत्येक अक्षरासाठी एक फंक्शन लिहा, उदा., `draw_a`, `draw_b`, *इत्यादी. तुमची ही फंक्शन्स* `letters.py` *मध्ये टाका. तुम्ही एक 'टर्टल टाइपराइटर' ('turtle typewriter') पुढील लिंकवरून डाऊनलोड करून तुमचा कोड तपासू शकता: http://thinkpython2.com/code/typewriter.py.*

उत्तरासाठी http://thinkpython2.com/code/letters.py बघा, ज्याला http://thinkpython2.com/code/polygon.py ह्यातला कोड पण लागतो.

प्रश्न ४.५. *स्पायरल्स (spirals) बद्दल http://en.wikipedia.org/wiki/Spiral इथे माहिती वाचा; नंतर आर्किमिडियन स्पायरल (Archimedian spiral) किंवा दुसरे कोणतेही स्पायरल काढणारा प्रोग्राम लिहा.*

उत्तर: http://thinkpython2.com/code/spiral.py.

प्रकरण ५

कंडिशनल आणि रिकर्शन (Conditional and recursion)

ह्या प्रकरणाचा मुख्य विषय `if` स्टेटमेंट आहे, जे प्रोग्रामच्या स्थितीला अनुसरून वेगळा कोड चालवते. पण त्याआधी दोन नवीन ऑपरेटर्सची ओळख करून घेऊ: फ्लोअर डिव्हिझन (floor division) आणि मॉड्युलस (modulus).

५.१ फ्लोअर डिव्हिझन आणि मॉड्युलस

फ्लोअर डिव्हिझन (**floor division**) ऑपरेटर, म्हणजेच `//` हा दोन संख्यांचा भागाकार करतो आणि उत्तराला इंटिजरवर राउंड डाऊन (round down) करतो. उदा., एखादा चित्रपट १०५ मिनिटांचा आहे; जर हा वेळ तासांत पाहिजे असेल आणि नेहमीचा भागाकार वापरला तर:

```
>>> minutes = 105
>>> minutes / 60
1.75
```

पण आपण दशांश चिन्ह वापरून तासातली वेळ सहसा नाही दर्शवत. फ्लोअर डिव्हिझन ऑपरेटर खाली राउंड करून तास इंटिजरमध्ये रिटर्न करतो:

```
>>> minutes = 105
>>> hours = minutes // 60
>>> hours
1
```

बाकी काढण्यासाठी तुम्ही एकूण मिनिटांमधून एक तास (म्हणजे ६० मिनिटे) वजा करू शकता:

```
>>> remainder = minutes - hours * 60
>>> remainder
45
```

दुसरा पर्याय **मॉड्युलस ऑपरेटर** (**modulus operator**), म्हणजेच `%` हा आहे जो दोन संख्यांचा भागाकार करून बाकी रिटर्न करतो.

```
>>> remainder = minutes % 60
>>> remainder
45
```

मॉड्युलस ऑपरेटर हा तुम्हाला वाटत असेल त्याहून अधिक उपयुक्त आहे. उदा., एका संख्येला दुसऱ्या संख्येने भाग जातो का हे तुम्ही तपासू शकता—जर `x % y` शून्य असेल तर `x` ला `y` ने पूर्ण भाग जातो.

अजून: संख्येच्या उजवीकडचे अंक शोधून काढू शकता. उदा., `x % 10` हे एक्स्प्रेशन `x`चा (पाया १० असलेल्या, म्हणजेच दशमान संख्यापद्धतीत) सर्वांत उजवीकडचा अंक देते. त्याचप्रमाणे `x % 100` उजवीकडचे दोन अंक देते, इ.

पायथॉन २ मध्ये थोडे वेगळे आहे. तिथे डिव्हिझन ऑपरेटर `/` हा दोन्ही संख्या इंटिजर्स असतील तर फ्लोअर डिव्हिझन करतो, आणि एक किंवा दोन्ही संख्या (ऑपरँड्स) `float` असतील तर फ्लोटिंग-पॉइंट डिव्हिझन करतो.

५.२ बूलियन एक्स्प्रेशन (Boolean expression)

बूलियन एक्स्प्रेशन (**boolean expression**) म्हणजे असे एक्स्प्रेशन जे एक तर सत्य (true) किंवा असत्य (false) असते. खालील उदाहरणे == ऑपरेटर वापरतात; हा ऑपरेटर दोन ऑपरँड्सची तुलना करतो आणि जर ते एकसारखेच असतील तर `True` उत्तर देतो नाही तर `False`:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

`True` आणि `False` ह्या विशेष व्हॅल्यूझ आहेत ज्यांचा टाइप `bool` हा आहे; त्या स्ट्रिंग्स नाहीयेत:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

== हा **रिलेश्नल ऑपरेटर** (**relational operator**) आहे; बाकीचे रिलेश्नल ऑपरेटर्स खालीलप्रमाणे:

```
x != y               # x आणि y वेगळे आहेत.
x > y                # x हा y पेक्षा मोठा आहे.
x < y                # x हा y पेक्षा लहान आहे.
x >= y               # x हा y पेक्षा मोठा आहे किंवा त्यासारखाच आहे.
x <= y               # x हा y पेक्षा लहान आहे किंवा त्यासारखाच आहे.
```

जरी ही वरची ऑपरेशन्स तुम्हाला माहीत असली तरी ह्याची नोंद घ्या की पायथॉन चिन्हे ही गणितीय चिन्हांपेक्षा वेगळी आहेत. खूपच वारंवार घडणारी एक चूक म्हणजे दुहेरी बरोबर चिन्हांऐवजी एकेरी बरोबर चिन्ह वापरणे (== ऐवजी =). हे पक्के लक्षात ठेवा की = हा असाइनमेंट ऑपरेटर आहे आणि == हा एक रिलेश्नल ऑपरेटर आहे. अजून एक: =< आणि => हे अर्थहीन आहे.

५.३ लॉजिकल ऑपरेटर (Logical operator)

पायथॉनमध्ये तीन **लॉजिकल ऑपरेटर्स** आहेत: `and`, `or`, आणि `not` ('आणि', 'किंवा', आणि, 'नव्हे'). ह्या ऑपरेटर्सचे सिमँटिक्स (semantics) म्हणजे ऑपरेटर्सचा 'अर्थ' हा संबंधित इंग्रजी शब्दांच्या अर्थासारखाच आहे. उदा., `x > 0 and x < 10` हे तेव्हा आणि फक्त तेव्हाच सत्य असेल जेव्हा `x`ची व्हॅल्यू ० हून अधिक *आणि* १० हून कमी असेल.

`n%2 == 0 or n%3 == 0` हे तेव्हा (आणि फक्त तेव्हाच) सत्य असेल जेव्हा एक किंवा दोन्ही अटी (conditions) सत्य असतील, दुसऱ्या शब्दांत जर `n`ला २ *किंवा* ३ ने पूर्ण भाग जात असेल तरच.

आणि `not` ऑपरेटर बूलियन एक्स्प्रेशनची व्हॅल्यू उलट करतो, म्हणजे `not (x > y)` हे तेव्हा (आणि फक्त तेव्हाच) सत्य असेल जेव्हा `x > y` हे असत्य असेल, म्हणजे, जेव्हा `x` जी व्हॅल्यू दर्शवते ती `y` जी दर्शवते तिच्यापेक्षा लहान किंवा त्यासारखीच असेल तर.

खरे तर लॉजिकल ऑपरेटरचे ऑपरँड्स बूलियन एक्स्प्रेशन्सच असले पाहिजेत पण पायथॉन त्याबाबतीत विशेष कडक नाहीये. कोणत्याही शून्याहून वेगळ्या संख्येला `True` असे मानले जाते:

```
>>> 42 and True
True
```

ह्या गोष्टीचा तुम्ही फायदा करून घेऊ शकता, पण काही अनाकलनीय मुद्द्यांमुळे ते थोडे संभ्रमात टाकणारे ठरू शकते. तर (तुम्हाला पायथॉनची चांगली समज येईपर्यंत) ते टाळलेलेच बरे.

५.४ कंडिश्नल एक्सेक्युशन (Conditional execution)

उपयुक्त प्रोग्राम्स लिहिण्यासाठी अटी तपासून त्यांनुसार प्रोग्रामची वर्तणूक (behavior) बदलता येणे गरजेचे आहे. **कंडिश्नल स्टेटमेंट** (**conditional statement**) आपल्याला ही क्षमता देते. सर्वांत सोपा प्रकार म्हणजे `if` स्टेटमेंट:

```
if x > 0:
    print('x is positive')
```

`if` नंतरच्या बूलियन एक्स्प्रेशनला **कंडिशन** (**condition, अट**) म्हणतात. जर ती अट पूर्ण केली, म्हणजेच त्या एक्स्प्रेशनची व्हॅल्यू सत्य असली तरच खालचे जागा सोडून लिहिलेले (indented) स्टेटमेंट एक्सेक्युट होते, नाहीतर नाही.

`if` स्टेटमेंट साधारण फंक्शन डेफनिशनसारखेच दिसते: एक हेडर आणि नंतर जागा सोडून लिहिलेली (indented) बॉडी. अशा स्टेटमेंटला **कंपाउंड स्टेटमेंट** (**compound statement**) म्हणतात. बॉडीमध्ये कितीही स्टेटमेंट्स असू शकतात, पण कमीतकमी एक तरी हवे. कधीकधी तुम्हाला कोणतेही स्टेटमेंट नसलेल्या बॉडीची गरज पडू शकते (विशेषतः भविष्यात कोड लिहिण्यासाठी जागा राखून ठेवण्यासाठी). तेव्हा तुम्ही तिथे `pass` स्टेटमेंट वापरू शकता; हे स्टेटमेंट काहीच करत नाही.

```
if x < 0:
    pass          # TODO: need to handle negative values!
```

५.५ पर्यायी एक्सेक्युशन (Alternative execution, पर्यायी फाटा)

`if` स्टेटमेंटचे दुसरे रूप म्हणजे: 'अल्टरनेटिव्ह एक्सेक्युशन.' ह्यात दोन शक्यता असतात आणि कोणती शक्यता निवडली जाईल हे त्याठिकाणी दिलेली अट ठरवते. सिंटॅक्स असा दिसतो (else म्हणजे अन्यथा/नाहीतर):

```
if x % 2 == 0:
    print('x is even')
else:
    print('x is odd')
```

`x`ला २ ने भागल्यावर बाकी ० असेल तर `x` सम (even) आहे, नाहीतर अट असत्य ठरेल आणि खालचे स्टेटमेंट एक्सेक्युट होईल. आणि अट ही सत्य किंवा असत्य ह्यांपैकी एकच असू शकते म्हणून दोन्हींपैकी नेमके एकच स्टेटमेंट एक्सेक्युट होऊ शकते. पर्यायाला **ब्रांच** (**branch**, फाटा) म्हणतात कारण तो फ्लो-ऑफ-एक्सेक्युशनमधील फाटा दर्शवतो.

५.६ कंडिश्नल्सची साखळी (Chained conditionals)

कधीकधी दोनपेक्षा अधिक शक्यता हाताळाव्या लागतात, म्हणून दोनपेक्षा जास्ती फाट्यांची गरज पडू शकते. असे गणन व्यक्त करण्यास **कंडिश्नल्सची साखळी** (**chained conditionals**) वापरतात:

```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

`elif` हे 'else if' चे संक्षिप्त रूप आहे. इथेही एक आणि एकच ब्रांच एक्सेक्युट होईल. तुम्ही कितीही `elif` स्टेटमेंट्स लावू शकता. जरी `else` नसले तरी चालते, पण `else` असले तर ते शेवटी टाकावे लागते.

```
if choice == 'a':
    draw_a()
elif choice == 'b':
    draw_b()
elif choice == 'c':
    draw_c()
```

प्रत्येक अट ही क्रमाने तपासली जाते. जर पहिलीची पूर्तता होत नसेल तर पुढची तपासली जाते, पुढचीची पूर्तता होत नसेल तर त्यापुढची, इत्यादी. जर त्यांपैकी एक सत्य असेल तर संबंधित ब्रांचमधला कोड चालवला जातो आणि स्टेटमेंट संपते. आणि जरी एकाहून जास्ती अटींची पूर्तता होत असेल तरी सर्वांत पहिले समाधान होणाऱ्या अटीची ब्रांच चालवली जाईल.

५.७ (एकात एक) गुंफलेले कंडिश्नल्स (Nested conditionals)

एका कंडिश्नलला दुसऱ्या कंडिश्नलमध्ये टाकता येते. मागच्या विभागातले उदाहरण आपण असेही लिहू शकलो असतो:

```
if x == y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less than y')
    else:
        print('x is greater than y')
```

बाहेरच्या कंडिश्नलला दोन फाटे आहेत. पहिल्यात एक साधे स्टेटमेंट आहे. दुसऱ्यात अजून एक `if` स्टेटमेंट आहे ज्याचे स्वतःचे दोन फाटे आहेत ज्यात प्रत्येकी एक साधे स्टेटमेंट आहे, पण त्यात कंडिश्नल स्टेटमेंटही असू शकले असते.

जरी सोडलेल्या जागेने (indentation) प्रोग्रामची मांडणी स्पष्ट होत असली तरी (एकात एक) **गुंफलेले कंडिश्नल्स** (**nested conditionals**) वाचायला अवघड होऊ शकतात. जमल्यास ते टाळलेले बरे.

कधीकधी, लॉजिकल ऑपरेटरचा गुंफलेल्या कंडिश्नल स्टेटमेंट्सचा गुंता सोडवण्यास उपयोग होतो. उदा., खालील कोड एकच कंडिश्नल वापरून आपण चांगल्या प्रकारे लिहू शकतो:

```
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

वर, `print` स्टेटमेंट तेव्हाच एक्सेक्युट होते जेव्हा दोन्ही अटींची पूर्तता होते, तर हेच साध्य करण्यासाठी `and` ऑपरेटर खालीलप्रमाणे वापरू शकतो:

```
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

अशा विशिष्ट अटीसाठी पायथॉन अजून संक्षिप्त पर्याय पुरवतो:

```
if 0 < x < 10:
    print('x is a positive single-digit number.')
```

५.८ रिकर्शन (Recursion)

एका फंक्शनमधून दुसरे फंक्शन कॉल करणे वैध आहे; स्वत:ला कॉल करणे सुद्धा वैध आहे. असे करण्याची गरज काय हे तुम्हाला जरी स्पष्ट नसले, तरी प्रोग्रामचे असे करण्याची क्षमता ही जवळजवळ जादुई गोष्टच आहे. उदा., खालील फंक्शन बघा. अनुवादकाची टिप्पणी: धरतीवरून अवकाशयान (रॉकेट) सोडतानाच्या काउंटडाऊनचा (count-down) संदर्भ खाली आहे; अवकाशयानाच्या उड्डाणाला ब्लास्टॉफ (blastoff) असे म्हणतात, आणि उलटे मोजण्याला काउंटडाऊन म्हणतात, जसे १०, ९, ८, ... काउंटडाऊन शून्यला पोहोचले की रॉकेट उडते.

```
def countdown(n):
    if n <= 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n-1)
```

जर `n` शून्य किंवा ऋण (negative) असेल तर 'Blastoff!' हा शब्द प्रिंट होतो. अन्यथा `n`ची व्हॅल्यू प्रिंट होते आणि `countdown` नावाचे फंक्शन, म्हणजे स्वत:लाच, `n-1` अर्ग्युमेंट देऊन कॉल केले जाते.

हे फंक्शन आपण खाली दाखवल्याप्रमाणे कॉल केल्यास काय होते?

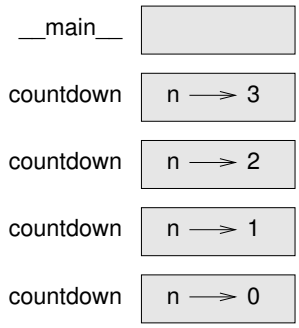
```
>>> countdown(3)
```

हे होते:

> `countdown`चे एक्सेक्युशन `n=3` ला सुरू होते, आणि `n` शून्याहून मोठा असल्याने ते 3 प्रिंट करते आणि स्वत:ला कॉल करते...
>
> > `countdown`चे एक्सेक्युशन `n=2` ला सुरू होते, आणि `n` शून्याहून मोठा असल्याने ते 2 प्रिंट करते आणि स्वत:ला कॉल करते...
> >
> > > `countdown`चे एक्सेक्युशन `n=1` ला सुरू होते, आणि `n` शून्याहून मोठा असल्याने ते 1 प्रिंट करते आणि स्वत:ला कॉल करते...
> > >
> > > > `countdown`चे एक्सेक्युशन `n=0` ला सुरू होते, आणि `n` *शून्याहून मोठा नसल्याने* ते 'Blastoff!' प्रिंट करते, संपते, आणि परतते (returns).
> > >
> > > `countdown`चा तो कॉल ज्याला `n=1` मिळाले होते तो संपतो, आणि परततो (returns).
> >
> > `countdown`चा तो कॉल ज्याला `n=2` मिळाले होते तो संपतो, आणि परततो (returns).
>
> `countdown`चा तो कॉल ज्याला `n=3` मिळाले होते तो संपतो, आणि परततो (returns).

ह्यानंतर तुम्ही परत `__main__` मध्ये येता. म्हणून आउटपुट असे दिसते:

```
3
2
1
Blastoff!
```



आकृती ५.१: स्टॅक डायग्राम.

स्वतःला कॉल करणाऱ्या फंक्शनला **रिकर्सिव्ह** (**recursive**) म्हणतात; त्याला एक्सेक्युट करण्याच्या प्रक्रियेला **रिकर्शन** (**recursion**) म्हणतात.

अजून एक उदाहरण म्हणजे दिलेली स्ट्रिंग `n` वेळा प्रिंट करणारे फंक्शन:

```
def print_n(s, n):
    if n <= 0:
        return
    print(s)
    print_n(s, n-1)
```

जर `n <= 0`, तर **रिटर्न स्टेटमेंट** (**return statement**) फंक्शनमधून बाहेर काढते. फ्लो-ऑफ-एक्सेक्युशन तडक कॉलर (caller) कडे परत येतो, त्यामुळे फंक्शनच्या बाकीच्या ओळी एक्सेक्युट होत नाहीत.

फंक्शनचा उरलेला भाग `countdown`सारखाच आहे: ते `s`ची व्हॅल्यू प्रिंट करते आणि स्वतःला `s`ची व्हॅल्यू अजून $n-1$ वेळा प्रिंट करायच्या दृष्टीने कॉल करते. म्हणजेच एकूण `1 + (n - 1)` ओळी प्रिंट होतात, ज्याची बेरीज `n` आहे.

असल्या साध्या उदाहरणांसाठी `for` लूप वापरलेले सोपे. पण आपण पुढे काही उदाहरणे पाहणार आहोत जी `for` लूप वापरून लिहिण्यास अवघड पण रिकर्शन वापरून लिहिण्यास सोपी आहेत, तर लवकर सुरुवात केलेली बरी.

५.९ रिकर्सिव्ह फंक्शनची स्टॅक डायग्राम

विभाग ३.९ मध्ये, प्रोग्रामची फंक्शन कॉलदरम्यानची स्थिती आपण स्टॅक डायग्रामने व्यक्त केली. तशाच आकृतीचा वापर आपण रिकर्सिव्ह फंक्शन समजून घ्यायला करू शकतो.

जेव्हाही फंक्शन कॉल होते, तेव्हा पायथॉन त्या फंक्शनच्या स्थानिक व्हेरिएबल्स आणि परॅमीटर्ससाठी एक फ्रेम बनवतो. पण ह्या स्टॅकमध्ये, रिकर्सिव्ह फंक्शनच्या एकाहून अधिक फ्रेम्स एकाच वेळेला असू शकतात.

आकृती ५.१ मध्ये `n = 3` घेऊन `countdown` कॉल केल्यावरची स्टॅक डायग्राम आहे.

आधी पाहिल्याप्रमाणे, स्टॅकची सर्वांत वरची फ्रेम `__main__` साठी आहे. ती रिकामी आहे कारण आपण `__main__` मध्ये एकही व्हेरिएबल बनवले नाही किंवा त्याला एकही अर्ग्युमेंट पाठवले नाही.

`countdown` साठीच्या चार फ्रेम्समध्ये `n`ची प्रत्येकी वेगळी व्हॅल्यू आहे. स्टॅकमध्ये सर्वांत खाली जिथे `n=0` आहे तिला **बेस-केस** (**base case**) म्हणतात. ती रिकर्सिव्ह कॉल करत नाही म्हणून त्यानंतर अजून फ्रेम्स नाहीत.

सरावासाठी `print_n` फंक्शनची `s = 'Hello'` आणि `n = 2` घेऊन कॉल केल्यानंतरची स्टॅक डायग्राम काढा. नंतर `do_n` नावाचे एक फंक्शन लिहा जे अर्ग्युमेंट्स म्हणून एक फंक्शन ऑब्जेक्ट आणि एक संख्या `n` घेते आणि दिलेले फंक्शन `n` वेळा कॉल करते.

५.१० इन्फिनेट रिकर्शन (Infinite recursion)

रिकर्शन जर बेस-केस पर्यंत पोहोचलेच नाही, तर ते अखंडितपणे रिकर्सिव्ह कॉल्स करत राहते, आणि प्रोग्राम कधीच थांबत नाही. ह्याला **इन्फिनेट रिकर्शन** (**infinite recursion**) म्हणतात, आणि ते साधारणपणे वाईट मानले जाते. खालील साधा प्रोग्राम इन्फिनेट रिकर्शन दाखवतो:

```
def recurse():
    recurse()
```

बहुतेक प्रोग्रामिंग एनव्हायर्नमेंट्स (programming environments) मध्ये, इन्फिनेट रिकर्शन असलेला प्रोग्राम काही खरेच अखंडितपणे चालू नाही राहत. रिकर्शनची कमाल खोली (maximum recursion depth) पोहोचल्यावर पायथॉन आपल्याला तसा एरर मेसेज देतो:

```
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
                  .
                  .
                  .
  File "<stdin>", line 2, in recurse
RuntimeError: Maximum recursion depth exceeded
```

आधी पाहिलेल्यांपेक्षा हा ट्रेसबॅक थोडा मोठा आहे. एरर आला तेव्हा स्टॅकवर `recurse` फंक्शनच्या १००० फ्रेम्स होत्या!

जर तुम्हाला चुकून इन्फिनेट रिकर्शन मिळाले, तर तुमचे फंक्शन नीट तपासून त्यात रिकर्सिव्ह कॉल न करणारी एकतरी बेस-केस आहे ह्याची खात्री करा. आणि जर बेस-केस असलीच, तर हे तपासा की आपल्या प्रोग्रामचा फ्लो-ऑफ-एक्सेक्युशन तिथपर्यंत खात्रीशीरपणे पोहोचू शकतो.

५.११ कीबोर्ड इनपुट (Keyboard input)

आपण आतापर्यंत पाहिलेले प्रोग्राम्स युझर (user) कडून कोणतेही इनपुट घेत नाहीत. ते प्रत्येक वेळेस सारखेच काम करतात.

पायथॉन, `input` नावाचे बिल्ट-इन (built-in) फंक्शन पुरवते जे प्रोग्राम थांबवून युझरच्या काहीतरी टाइप करण्याची वाट पाहते. युझरने Return किंवा Enter मारल्यावर प्रोग्राम परत सुरू होतो आणि युझरने जेही टाइप केले (लिहिले), त्याला `input` फंक्शन स्ट्रिंग म्हणून रिटर्न करते. (पायथॉन २ मध्ये ह्याच फंक्शनला `raw_input` म्हणतात.)

```
>>> text = input()
What are you waiting for?
>>> text
'What are you waiting for?'
```

इनपुट घेण्याआधी युझरकडून कोणती माहिती अपेक्षित आहे हे प्रिंट केलेले चांगले; ह्याला प्रॉम्प्ट म्हणतात. `input` फंक्शन अर्ग्युमेंट म्हणून एक प्रॉम्प्ट घेऊ शकते:

```
>>> name = input('What...is your name?\n')
What...is your name?
Arthur, King of the Britons!
>>> name
'Arthur, King of the Britons!'
```

प्रॉम्प्टच्या शेवटी असलेले `\n` हे **न्यूलाइन** (**newline**) दर्शवणारे आणि ओळ संपवणारे एक विशेष कॅरेक्टर (character) आहे. आणि म्हणूनच युझरचे इनपुट त्या प्रॉम्प्टच्या खाली येते. युझर एक इंटिजर टाइप करेल अशी तुमची अपेक्षा असेल तर तुम्ही मिळालेल्या व्हॅल्यूचे `int` मध्ये रुपांतर करू शकता:

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
42
>>> int(speed)
42
```

पण जर युझरने अंकांची स्ट्रिंग सोडून दुसरे काही टाइप केले तर तुम्हाला एरर मिळेल:

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10
```

अशाप्रकारचे एरर्स कसे हाताळायचे हे आपण नंतर पाहू.

५.१२ डीबगिंग (Debugging)

सिंटॅक्स किंवा रनटाइम एरर जेव्हा येतो, तेव्हा एरर मेसेजमध्ये भरपूर माहिती असते, पण ती पाहून तुम्ही भारावून जाऊ शकता. त्यातले फायदेशीर भाग साधारणपणे हे असतात:

- एरर कोणत्या प्रकारचा होता, आणि
- तो कुठे आला.

सिंटॅक्स एरर्स सहसा शोधायला सोपे असतात, पण सुरुवातीला काही थोडे अनपेक्षित असतात. व्हाइटस्पेस (whitespace) एरर्स थोडे संभ्रमात पाडणारे असतात कारण स्पेसेस आणि टॅब्स (tabs) अदृश्य असतात आणि आपल्याला त्यांच्याकडे दुर्लक्ष करायची सवय असते.

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

ह्या उदाहरणात एका स्पेसने केलेल्या इंडेंटमुळे दुसऱ्या ओळीवर चूक झाली आहे. पण एरर मेसेज `y`ला निर्देशित करतो, जे दिशाभूल करणारे आहे. साधारणपणे एरर मेसेज चूक कुठे सापडली ते दर्शवतो, पण खरा एरर आधी असू शकतो, किंवा कधीकधी वरच्याच ओळीवर.

हीच गोष्ट रनटाइम एरर्सना पण लागू होते. जर तुम्ही सिग्नल-टू-नॉइझ डेसिबलमध्ये शोधत असाल तर सूत्र (जे आपण विभाग ३.२ मध्ये पाहिले होते) असे आहे

$$\mathrm{SNR}_{\mathrm{db}} = 10 \log_{10}(P_{\mathrm{signal}} / P_{\mathrm{noise}}) .$$

पायथॉनमध्ये तुम्ही हे खालीलप्रमाणे शोधू शकता:

```
import math
signal_power = 9
noise_power = 10
ratio = signal_power // noise_power
decibels = 10 * math.log10(ratio)
print(decibels)
```

हा प्रोग्राम चालवल्यावर तुम्हाला एक एक्सेप्शन (exception) मिळते:

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
    decibels = 10 * math.log10(ratio)
ValueError: math domain error
```

एरर मेसेज पाचवी ओळ दर्शवतो, पण तिथे काहीच चूक नाहीये. खरा एरर शोधण्यासाठी `ratio` ची व्हॅल्यू प्रिंट करणे फायदेशीर ठरते, जी शून्य आहे. (शून्यचा लॉग घेणे गणितात अर्थहीन आहे.) ओळ क्रमांक ४ मध्ये चूक आहे, आणि ती म्हणजे फ्लोटिंग-पॉइंट डिव्हिझन ऐवजी फ्लोअर डिव्हिझन वापरणे.

एरर मेसेजेस तुम्ही अतिशय काळजीपूर्वक वाचले पाहिजेत, पण असे समजू नका की ते सांगतात ते सर्व सत्य असते. (गुप्तहेरगिरीचा संदर्भ आठवतोय का?)

५.१३ शब्दार्थ

फ्लोअर डिव्हिझन (floor division): `//` ने दर्शवला जाणारा ऑपरेटर जो पहिल्या संख्येला दुसरीने भागतो आणि उत्तराचे इंटिजरवर राउंडिंग डाऊन (round down) करतो; राउंडिंग ही नेहमी **खाली** होते. उदा., `5 // 2` चे उत्तर 2 आहे पण `-5 // 2` चे उत्तर -3 आहे.

मॉड्युलस ऑपरेटर (modulus operator): % ह्या चिन्हाने दर्शवला जाणारा ऑपरेटर जो दोन इंटिजर्स वर वापरता येतो आणि पहिल्याला दुसऱ्याने भागल्यावर येणारी बाकी रिटर्न करतो.

बूलियन एक्स्प्रेशन (boolean expression): असे एक्स्प्रेशन ज्याची व्हॅल्यू एकतर `True` किंवा `False` असते.

रिलेश्नल ऑपरेटर (relational operator): खालीलपैकी एक ऑपरेटर; हा त्याच्या ऑपरँड्सची तुलना करतो:

==, !=, >, <, >=, आणि <=.

लॉजिकल ऑपरेटर (logical operator): खालीलपैकी एक ऑपरेटर; हा बूलियन एक्स्प्रेशन्सचे एकत्रीकरण करतो:

`and`, `or`, आणि `not`.

कंडिश्नल स्टेटमेंट (conditional statement): फ्लो-ऑफ-एक्सेक्युशन मध्ये अट (condition) वापरून बदल घडवून आणणारे स्टेटमेंट.

कंडिशन (condition, अट): एका कंडिश्नल स्टेटमेंटमधील एक्स्प्रेशन जे कोणती ब्रांच घ्यायची ते ठरवते.

कंपाउंड स्टेटमेंट (compound statement): असे स्टेटमेंट ज्यात हेडर आणि बॉडी असते. हेडरचा शेवट अपूर्णविरामाने (म्हणजे ':' ने) होतो. स्टेटमेंटची बॉडी त्याच्या हेडरपेक्षा थोडी जास्त जागा सोडून लिहिलेली (म्हणजेच, indented) असते.

ब्रांच (branch, फाटा): एका कंडिश्नल स्टेटमेंटमधल्या अनेक पर्यायी स्टेटमेंट्सच्या गटांपैकी एक गट.

कंडिश्नलची साखळी (chained conditional, चैन्ड कंडिश्नल): असे कंडिश्नल स्टेटमेंट ज्यात अनेक पर्यायी ब्रांचेस आहेत.

नेस्टेड कंडिश्नल (nested conditional, गुंफलेले कंडिश्नल): असे कंडिश्नल स्टेटमेंट जे दुसऱ्या कंडिश्नल स्टेटमेंटच्या एका ब्रांचमध्ये आहे.

रिटर्न स्टेटमेंट (return statement): असे स्टेटमेंट ज्याने फंक्शन ताबडतोब संपते आणि कॉलर (caller) कडे परतते.

रिकर्शन (recursion): जे फंक्शन एक्सेक्युट होत आहे तेच कॉल करण्याची प्रक्रिया.

बेस-केस (base case): रिकर्सिव्ह फंक्शनमधली रिकर्सिव्ह कॉल *न* करणारी कंडिश्नल ब्रांच.

इन्फिनेट रिकर्शन (infinite recursion): असे रिकर्शन ज्यात बेस-केस नसते किंवा असले तरी तिथपर्यंत कधीच पोहोचत नाही. इन्फिनेट रिकर्शनचा शेवट रनटाइम एररने होतो.

५.१४ प्रश्नसंच (Exercises)

प्रश्न ५.१. `time` *नावाचे एक मोड्युल आहे त्यात असे एक फंक्शन आहे ज्याचे नावसुद्धा* `time`*च आहे. ते फंक्शन सध्याची ग्रीनविच प्रमाणवेळ (Greenwich Mean Time, GMT) द इपॉक ('the epoch') स्वरुपात देते; म्हणजेच ही एक विशिष्ठ संदर्भ वापरून व्यक्त केलेली वेळ आहे. युनिक्स (UNIX) सिस्टम्समध्ये १ जानेवारी १९७० ही इपॉकची वेळ आहे.*

```
>>> import time
>>> time.time()
1437746094.5735958
```

सध्याची वेळ ह्या स्वरुपात वाचून आता किती वाजले आहेत ते आणि इपॉक होऊन किती दिवस झाले ते शोधून काढणारी स्क्रिप्ट लिहा.

प्रश्न ५.२. *फर्मॅटचे शेवटचे प्रमेय (Fermat's Last Theorem) म्हणते की जर $n \geq 3$ ही एक पूर्णांक संख्या असेल तर खालील समीकरणाचे समाधान करणाऱ्या कोणत्याच धन पूर्णांक संख्या (positive integers) a, b, आणि c अस्तित्वात नाहीयेत*

$$a^n + b^n = c^n .$$

मराठी विश्वकोशात ह्याची नोंद पुढील लिंकवर आहे: https://marathivishwakosh.org/32990/.

१. `check_fermat` *नावाचे फंक्शन लिहा जे चार पॅरमीटर्स घेते—*`a`, `b`, `c` *आणि* `n`*—आणि फर्मॅटचे प्रमेय चूक आहे का ते तपासते. जर n ची व्हॅल्यू २ पेक्षा जास्त असली आणि $a^n + b^n = c^n$ असले, तर प्रोग्रामने 'Holy smokes, Fermat was wrong!' हे प्रिंट केले पाहिजे, नाही तर 'No, that doesn't work.'*

२. *युझरला* `a`, `b`, `c` *आणि* `n` *इनपुट करायला सांगून, त्यांचे इंटिजरमध्ये रुपांतर करून,* `check_fermat` *वापरून त्या संख्या फर्मॅटच्या प्रमेयाचे उल्लंघन करतात का हे तपासणारे फंक्शन लिहा.*

प्रश्न ५.३. *जर तुम्हाला तीन काठ्या दिल्या तर त्यांची मांडणी एका त्रिकोणात करणे शक्य होईलच असे नाही. उदा., एक काठी १२ इंची असली आणि इतर दोन प्रत्येकी एक इंच लांब असल्या तर तुम्हाला लहान काठ्यांचा एकमेकांना स्पर्श घडवता येणार नाही. कोणत्याही तीन लांब्यांसाठी त्यांची त्रिकोणात मांडणी करता येईल का नाही हे एका सोप्या चाचणीने तपासता येते:*

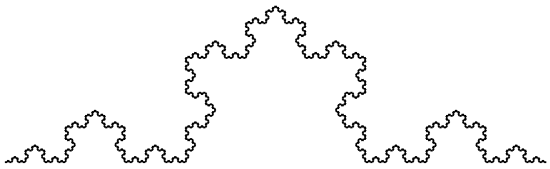
> *तिन्हींपैकी कोणतीही लांबी जर इतर दोन लांब्यांच्या बेरजेहून अधिक असेल, तर तुम्हाला त्रिकोण बनवता नाही येणार. अन्यथा येईल. (जर दोन लांब्यांची बेरीज तिसरीइतकी असेल तर त्यांचा 'degenerate' त्रिकोण बनतो.)*

१. *तीन इंटिजर्स अर्ग्युमेंट म्हणून घेणारे* `is_triangle` *नावाचे फंक्शन लिहा जे दिलेल्या तीन लांब्यांचा त्रिकोण बनवता येईल की नाही ह्यावरून 'Yes' किंवा 'No' आउटपुट करेल.*

२. *युझरला तीन काठ्यांच्या लांब्या इनपुट करायला सांगून, त्यांचे इंटिजरमध्ये रुपांतर करून,* `is_triangle` *वापरून त्यांचा त्रिकोण होतो का हे तपासणारे फंक्शन लिहा.*

प्रश्न ५.४. *खालील प्रोग्रामचे आउटपुट काय आहे? ज्यावेळी हा प्रोग्राम उत्तर प्रिंट करतो त्यावेळची स्टॅक डायग्राम काढा.*

```
def recurse(n, s):
    if n == 0:
        print(s)
    else:
        recurse(n-1, n+s)

recurse(3, 0)
```



आकृती ५.२: एक Koch कर्व्ह (curve, वक्ररेषा).

१. *जर तुम्ही हे फंक्शन* `recurse(-1, 0)` *असे कॉल केले तर काय होईल?*

२. *हे फंक्शन वापरण्यास लागणारी सर्व माहिती (आणि अजून काही नाही, जी लागते नेमकी तीच माहिती) देणारी डॉकस्ट्रिंग (docstring) लिहा.*

खालील प्रश्नांत प्रकरण ४ मध्ये बघितलेल्या `turtle` मोड्युलचा वापर केला आहे:

प्रश्न ५.५. *खालील फंक्शन वाचा आणि ते काय करते हे तुम्हाला समजतेय का ते बघा (प्रकरण ४ मधली उदाहरणे बघा). नंतर ते फंक्शन चालवून तुम्हाला बरोबर समजले का ह्याची खात्री करा.*

```
def draw(t, length, n):
    if n == 0:
        return
    angle = 50
    t.fd(length*n)
    t.lt(angle)
    draw(t, length, n-1)
    t.rt(2*angle)
    draw(t, length, n-1)
    t.lt(angle)
    t.bk(length*n)
```

प्रश्न ५.६. *Koch कर्व्ह हा एक फ्रॅक्टल (fractal) आहे जो आकृती ५.२ मध्ये दाखवल्याप्रमाणे दिसतो. असा x लांबीचा Koch कर्व्ह काढण्यासाठी तुम्हाला खालील कृती करावी लागते:*

१. *एक $x/3$ लांबीचा Koch कर्व्ह काढा.*

२. *डावीकडे ६० अंशात वळा.*

३. *एक $x/3$ लांबीचा Koch कर्व्ह काढा.*

४. *उजवीकडे १२० अंशात वळा.*

५. *एक $x/3$ लांबीचा Koch कर्व्ह काढा.*

६. *डावीकडे ६० अंशात वळा.*

७. *एक $x/3$ लांबीचा Koch कर्व्ह काढा.*

ह्या कृतीला अपवाद हा की जर x ची व्हॅल्यू ३ पेक्षा कमी असेल तर तुम्ही एक x लांबीचा रेषाखंड काढू शकता.

१. *एक टर्टल आणि एक लांबी असे दोन अर्ग्युमेंट्स घेणारे आणि दिलेले टर्टल वापरून दिलेल्या लांबीचा Koch कर्व्ह काढणारे* `koch` *नावाचे फंक्शन लिहा.*

२. *तीन Koch कर्व्ह काढून हिमकणाची (snowflake) आकृती काढणारे* `snowflake` *नावाचे फंक्शन लिहा.*

   उत्तर: http://thinkpython2.com/code/koch.py.

३. *Koch कर्व्हला अनेक मार्गांनी जनरलाइझ (generalize, व्यापकता वाढवणे) करता येते. पुढील लिंकवर अधिक माहिती मिळवून तुमच्या आवडीचे कर्व्ह काढणारी फंक्शन्स लिहा:* http://en.wikipedia.org/wiki/Koch_snowflake

प्रकरण ६

फलदायी फंक्शन (Fruitful function)

आपण वापरलेल्या फंक्शन्सपैकी बरीच फंक्शन्स एक व्हॅल्यू रिटर्न करतात, उदा., math फंक्शन्स. पण आपण लिहिलेली सर्व फंक्शन्स व्हॉय्ड आहेत: त्यांच्याने काहीतरी परिणाम होतो, जसे एखादी व्हॅल्यू प्रिंट करणे, टर्टलला हलवणे, इ., पण ती काही रिटर्न करत नाहीत. ह्या प्रकरणात आपण फलदायी फंक्शन्स लिहिण्यास शिकणार आहोत.

६.१ रिटर्न व्हॅल्यू (Return value)

फंक्शन कॉल केल्यावर एका रिटर्न व्हॅल्यूची निर्मिती होते, जी आपण सहसा एका व्हेरिएबलला असाइन करतो किंवा एखाद्या एक्स्प्रेशनमध्ये वापरतो.

```
e = math.exp(1.0)
height = radius * math.sin(radians)
```

आतापर्यंत आपण लिहिलेली सर्व फंक्शन्स व्हॉय्ड आहेत. ढोबळमानाने, त्यांची काहीच रिटर्न व्हॅल्यू नाही; पण खरे तर, त्यांची रिटर्न व्हॅल्यू `None` आहे.

एकदाचे आपण ह्या प्रकरणात आपण फलदायी फंक्शन्स लिहिणार आहोत. पहिले उदाहरण `area` फंक्शनचे आहे, जे वर्तुळाची त्रिज्या (radius) घेऊन त्याचे क्षेत्रफळ (area) शोधून रिटर्न करते:

```
def area(radius):
    a = math.pi * radius**2
    return a
```

`return` स्टेटमेंट आपण आधी पाहिले आहे, पण एका फलदायी फंक्शनमधील `return` स्टेटमेंटमध्ये एका एक्स्प्रेशनचा समावेश असतो. तशा रिटर्न स्टेटमेंटचा अर्थ असा होतो: 'ह्या फंक्शनमधून ताबडतोब निघा आणि पुढील एक्स्प्रेशन रिटर्न व्हॅल्यू म्हणून वापरा.' एक्स्प्रेशन कितीही गुंतागुंतीचे असू शकते, म्हणजेच, आपण वरील फंक्शन संक्षिप्तपणे असे लिहू शकलो असतो:

```
def area(radius):
    return math.pi * radius**2
```

परंतु, a सारख्या **टेंपररी व्हेरिएबल** (**temporary variable**) चा डीबगिंगसाठी फायदा होतो. कधीकधी एकाधिक रिटर्न स्टेटमेंट्सची गरज पडते, एका कंडिश्नलच्या प्रत्येक ब्रांचमध्ये एक, असे:

```
def absolute_value(x):
    if x < 0:
        return -x
    else:
        return x
```

ही `return` स्टेटमेंट्स अल्टरनेटिव्ह कंडिश्नल्स (alternative conditionals) मध्ये असल्याने त्यांपैकी नेमके एकच एक्सेक्युट होते.

अनुवादकाची टिप्पणी: Absolute value म्हणजे, केवलमूल्य, म्हणजे त्या संख्येचे शून्यापासूनचे संख्यारेषेवरील अंतर. पुढील विकिपीडिया लिंक बघा: https://en.wikipedia.org/wiki/Absolute_value.

कोणतेही `return` स्टेटमेंट रन (run) झाल्याझाल्या लगेच पुढील कोणतीही स्टेटमेंट्स *न* चालता फंक्शनचा अंत होतो. एका `return` स्टेटमेंट नंतर येणाऱ्या कोड-ला किंवा जिथे फ्लो-ऑफ-एक्सेक्युशन कधीच पोहोचू शकत नाही अशा कोणत्याही कोड-ला **डेड कोड** (**dead code**) म्हटले जाते. फलदायी फंक्शनमधून जाणारा प्रत्येक मार्ग अखेरीस एका रिटर्न स्टेटमेंटपर्यंत पोहोचतो ह्याची खबरदारी घेणे महत्त्वाचे आहे. उदा.:

```
def absolute_value(x):
    if x < 0:
        return -x
    if x > 0:
        return x
```

हे फंक्शन चूक आहे कारण `x` जर शून्य असेल तर कोणतीही कंडिशन (अट) सत्य ठरणार नाही आणि कोणत्याही रिटर्न स्टेटमेंटशी गाठ *न* पडताच फंक्शन संपेल. जर फ्लो-ऑफ-एक्सेक्युशन फंक्शनच्या शेवटी पोहोचला तर रिटर्न व्हॅल्यू ही `None` असते, आणि साहजिक आहे, शून्याचे केवलमूल्य (absolute value) `None` नाहीये.

```
>>> print(absolute_value(0))
None
```

तसे तर पायथॉन `abs` नावाचे केवलमूल्य काढणारे बिल्ट-इन फंक्शन पुरवते. सराव म्हणून `compare` नावाचे फंक्शन लिहा जे `x` आणि `y` हे दोन परॅमीटर्स घेते, आणि जर `x > y` असेल तर `1` रिटर्न करते, जर `x == y` असेल तर `0` रिटर्न करते, आणि जर `x < y` तर `-1`.

६.२ इन्क्रिमेंटल डेव्हेलपमेंट (Incremental development, तुकड्या-तुकड्यांनी विस्तार)

तुम्ही जसजशी मोठी फंक्शन्स लिहाल, तुम्हाला तसतसे हे दिसून येईल की तुमचा डीबगिंगमध्ये जास्ती वेळ जातोय.

गुंतागुंत वाढत जाणारा प्रोग्राम लिहिण्यासाठी **इन्क्रिमेंटल डेव्हेलपमेंट** (**incremental development**) ही प्रक्रिया फायदेशीर ठरते. ह्यात डीबगिंगवर सतत (एकाच वेळी) खूप वेळ *न* घालवणे हा उद्देश असतो. हे साधण्यासाठी सर्व कोड एकदम लिहिण्याऐवजी तुकड्यातुकड्यांनी वाढवला जातो.

उदा., समजा तुम्हाला (x_1, y_1) आणि (x_2, y_2) ह्या दोन बिंदूंमधील अंतर काढायचे आहे. तर पायथागोरसच्या प्रमेयानुसार (Pythagorean theorem), अंतर खालील सूत्राने दिले जाते:

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

पहिली पायरी म्हणजे `distance` हे फंक्शन पायथॉनमध्ये कसे असेल हा विचार करणे. म्हणजेच, इनपुट परॅमीटर्स कोणते आहेत आणि आउटपुट (रिटर्न व्हॅल्यू) काय आहे?

ह्याठिकाणी, दिलेले दोन बिंदू हे इनपुट आहे, जे आपण चार संख्या वापरून व्यक्त करू शकतो. आणि रिटर्न व्हॅल्यू म्हणजे अंतर दर्शवणारी फ्लोटिंग-पॉइंट व्हॅल्यू.

ह्यावरून लगेच तुम्ही त्या फंक्शनचा आराखडा पण तयार करू शकता:

```
def distance(x1, y1, x2, y2):
    return 0.0
```

साहजिक आहे की वरील फंक्शन अंतर शोधत नाहीये, ते नेहमी शून्य रिटर्न करते. पण त्याचा सिंटॅक्स बरोबर आहे, आणि ते चालते, म्हणजेच तुम्ही त्यात भर घालण्याआधी टेस्ट (test) करू शकता (तपासून बघू शकता).

आपले नवीन फंक्शन टेस्ट करण्यासाठी त्यास नमुन्यादाखल काही अर्ग्युमेंट्स पाठवा:

```
>>> distance(1, 2, 4, 6)
0.0
```

ह्या व्हॅल्यूझ निवडण्याचे कारण म्हणजे त्यांत आडवे अंतर ३ आहे आणि उभे अंतर ४ आहे; त्याअर्थी उत्तर ५ आहे, जी एका ३-४-५ काटकोन त्रिकोणाच्या कर्णाची लांबी आहे. टेस्टिंग (testing) करताना अपेक्षित उत्तर माहित असणे जरूरी आहे.

सध्या आपल्याला हे माहीतये की फंक्शनचा सिंटॅक्स बरोबर आहे, आणि आपण त्यात कोड-ची भर घालण्यास तयार आहोत. करण्याजोगी पुढची एक गोष्ट म्हणजे $x_2 - x_1$ आणि $y_2 - y_1$ ह्या वजाबाक्या शोधून काढणे. खालील सुधारित स्वरूप ह्या व्हॅल्यूझ टेंपररी व्हेरिएबल्समध्ये ठेवून त्यांना प्रिंट करते.

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print('dx is', dx)
    print('dy is', dy)
    return 0.0
```

जर फंक्शन ठीक चालत असेल तर `dx is 3` आणि `dy is 4` हे प्रिंट झाले पाहिजे. तसे होत असेल तर आपण आत्मविश्वासाने म्हणू शकतो की फंक्शनला बरोबर अर्ग्युमेंट्स मिळताहेत आणि ते बरोबर गणन करत आहे. आणि तसे होत नसेल तर फक्त थोड्याच ओळी तपासाव्या लागतील.

आता आपण `dx` आणि `dy` ह्यांच्या वर्गांची बेरीज शोधणार आहोत:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    print('dsquared is: ', dsquared)
    return 0.0
```

तुम्ही प्रोग्राम परत रन करून आउटपुट बरोबर आहे ह्याची खात्री करा (25 असले पाहिजे). शेवटी, `math.sqrt` वापरून तुम्ही उत्तर शोधून रिटर्न करू शकता:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    return result
```

जर हे नीट चालत असेल तर आपले काम झाले. नाहीतर तुम्ही `result` रिटर्न करायच्या आधी त्याची व्हॅल्यू प्रिंट करून बघू शकता.

ह्या फंक्शनचे अंतिम रूप काहीच प्रिंट करत नाही, ते फक्त एक व्हॅल्यू रिटर्न करते. आधी लिहिलेली `print` स्टेटमेंट्स ही डीबगिंगसाठी उपयोगी असली तरी एकदाचे फंक्शन व्यवस्थित चालू लागल्यावर ती काढून टाकली पाहिजेत. तसल्या कोड-ला **स्कॅफोल्डिंग** (**scaffolding**) म्हणतात कारण तो प्रोग्रामची बांधणी करायला उपयोगी आहे पण पूर्ण कोड-चा भाग नाही. (अनुवादकाची टिप्पणी: स्कॅफोल्डिंग म्हणजे इमारत बांधताना कारागीरांच्या येण्याजाण्यासाठी किंवा इतर कामांसाठी उंचावर बनवलेले लाकडी फलाट. जसे इमारत बांधताना त्यांचा उपयोग होतो पण ती बांधून झाल्यावर ते काढून टाकतात, तसेच ह्या प्रिंट स्टेटमेंट्सचे आहे.)

सुरुवातीला तुम्ही एका वेळी फक्त एक किंवा दोन ओळींची भर घालत चला. जसजसा तुमचा अनुभव वाढेल, तसतसे तुम्ही मोठे तुकडे लिहू आणि डीबग करू शकता. हे नक्की की इन्क्रिमेंटल डेव्हेलपमेंट तुमचा डीबगिंगचा खूप वेळ वाचवू शकते.

ह्या प्रक्रियेचे प्रमुख पैलू असे आहेत:

१. एका चालणाऱ्या लहान प्रोग्रामने सुरुवात करा आणि त्यात लहान सुधारणा करत त्याला वाढवा. कोणत्याही क्षणी एरर आला तर तुम्हाला चांगली कल्पना असेल तो कुठे आहे.

२. अंतरिम (दरम्यानच्या) व्हॅल्यूझ ठेवण्यासाठी व्हेरिएबल्स वापरा जेणेकरून तुम्ही ती प्रिंट करून तपासू शकता.

३. प्रोग्राम एकदाचा नीट चालू लागल्यावर तुम्ही स्कॅफोल्डिंग काढून टाकू शकता; आणि अनेक स्टेटमेंट्सचे कंपाउंड एक्स्प्रेशन्समध्ये एकत्रीकरण करू शकता, पण ह्याची काळजी घ्या की त्यानंतर प्रोग्राम समजायला अवघड होणार नाही.

सराव म्हणून इन्क्रिमेंटल डेव्हेलपमेंटने एका काटकोन त्रिकोणाच्या इतर दोन बाजूंची लांबी अर्ग्युमेंट्स म्हणून घेऊन त्याच्या कर्णाची लांबी रिटर्न करणारे `hypotenuse` नावाचे फंक्शन लिहा (अनुवादकाची टिप्पणी: hypotenuse म्हणजे काटकोन त्रिकोणातील कर्ण; चौरस, आयत किंवा इतर बहुभुजाकृतींमधील कर्णाला diagonal म्हणतात). ह्या प्रक्रियेतील प्रत्येक टप्प्याची पुढे जाताजाता नोंद ठेवा.

६.३ कॉंपझिशन (Composition)

आतापर्यंत तुम्हाला हे नक्कीच समजले असेल की आपण एक फंक्शन दुसऱ्या फंक्शनमधून कॉल करू शकतो. एक उदाहरण म्हणून आपण एक फंक्शन लिहूया जे दोन बिंदू घेते—पहिला म्हणजे वर्तुळाचे केंद्र आणि दुसरा म्हणजे वर्तुळावरील कोणताही एक बिंदू—आणि त्या वर्तुळाचे क्षेत्रफळ काढते.

असे समजा की केंद्रबिंदू `xc` आणि `yc` मध्ये तर वर्तुळावरील बिंदू `xp` आणि `yp` मध्ये आहे. पहिली पायरी म्हणजे वर्तुळाची त्रिज्या शोधणे; ह्याठिकाणी, वर्तुळाची त्रिज्या ही दिलेल्या दोन बिंदूंमधील अंतराइतकीच आहे. आपण वरच `distance` फंक्शन लिहिले जे नेमके आपल्याला पाहिजे तेच करते:

```
radius = distance(xc, yc, xp, yp)
```

वर, radius म्हणजे त्रिज्या.

पुढची पायरी म्हणजे ती त्रिज्या वापरून त्या वर्तुळाचे क्षेत्रफळ शोधणे; आणि ह्यासाठीसुद्धा आपण नुकतेच एक फंक्शन लिहिले आहे:

```
result = area(radius)
```

ह्या पायऱ्यांना एका फंक्शनमध्ये एन्कॅप्सुलेट (encapsulate) करूया:

```
def circle_area(xc, yc, xp, yp):
    radius = distance(xc, yc, xp, yp)
    result = area(radius)
    return result
```

इथे, `radius` आणि `result` ही टेंपररी व्हेरिएबल्स डेव्हेलपमेंट आणि डीबगिंगसाठी उपयुक्त असली तरी एकदाचा प्रोग्राम नीट चालू लागल्यावर आपण फंक्शन कॉल्स-ना गुंफून (compose करून) संक्षेपाने असे लिहू शकतो:

```
def circle_area(xc, yc, xp, yp):
    return area(distance(xc, yc, xp, yp))
```

६.४ बूलियन फंक्शन (Boolean function)

फंक्शन बूलियन व्हॅल्यू रिटर्न करू शकतात, आणि ही गोष्ट वापरून आपण गुंतागुंतीच्या चाचण्या फंक्शनमध्ये लपवू शकतो. उदा.:

```
def is_divisible(x, y):
    if x % y == 0:
        return True
    else:
        return False
```

बूलियन फंक्शनच्या नावासाठी हो/नाही उत्तर असलेला प्रश्न वापरण्याची पद्धत साधारणपणे पाळली जाते; `is_divisible` फंक्शन `x` ला `y` ने पूर्ण भाग जातो का नाही हे त्याच्या रिटर्न व्हॅल्यू `True` किंवा `False` वरून सांगते.

हे एक उदाहरण:

```
>>> is_divisible(6, 4)
False
>>> is_divisible(6, 3)
True
```

इथे == ऑपरेटरने शोधलेले उत्तर बूलियन असल्यामुळे आपण ते थेटपणे रिटर्न करून फंक्शन जास्त संक्षेपाने असे लिहू शकतो:

```
def is_divisible(x, y):
    return x % y == 0
```

बूलियन फंक्शन साधारणपणे कंडिश्नल स्टेटमेंटमध्ये वापरले जाते:

```
if is_divisible(x, y):
    print('x is divisible by y')
```

तुम्हाला वाटेल आपण असेही लिहू शकतो:

```
if is_divisible(x, y) == True:
    print('x is divisible by y')
```

पण ती अतिरिक्त तुलना अनावश्यक आहे.

सराव म्हणून `is_between(x, y, z)` नावाचे फंक्शन लिहा जे $x \leq y \leq z$ असेल तर `True` रिटर्न करते, नाहीतर `False` रिटर्न करते.

६.५ अजून रिकर्शन (More recursion)

आतापर्यंत, आपण पायथॉनचा थोडासाच भाग पाहिलेला आहे, पण तुम्हाला हे ऐकून कुतूहल वाटेल की हा भाग एक *कंप्लीट* (*complete, पूर्ण*) प्रोग्रामिंग लँग्वेज आहे. म्हणजे काय? तर ज्याही गोष्टीचे कॉंप्युटर गणन करू शकतो, त्या हा भाग वापरून आपण पायथॉनमध्ये करू शकतो. जगात आतापर्यंत लिहिला गेलेला कोणताही प्रोग्राम तुम्ही जितके पायथॉन शिकलेले आहात तितकेच वापरूनसुद्धा लिहिता येऊ शकतो; खरे तर तुम्हाला फक्त माऊस (mouse), डिस्क (disk), इ. डिव्हाइसेस (devices) नियंत्रित करणाऱ्या थोड्याफार कमांड्स (commands) लागतील, पण अजून फक्त तेच लागेल.

ह्याची (गणितीय) सिद्धता देण्याचे अकल्पनीय काम **ॲलन ट्युरिंग** (**Alan Turing**) [१] ह्यांनी केले. श्री. ट्युरिंग हे पहिल्या काही संगणक वैज्ञानिकांपैकी एक (काही जण त्यांना गणितज्ञ म्हणतील पण त्या काळातले खूप संगणक वैज्ञानिक आधी गणितज्ञ होते). आणि म्हणूनच वरच्या परिच्छेदातील विधानाला ट्युरिंग थीसिस (Turing Thesis) म्हणतात. ट्युरिंग थीसिसविषयी पूर्ण (आणि अजून व्यवस्थित) माहितीसाठी मायकल सिप्सर (Michael Sipser) ह्यांचे *Introduction to the Theory of Computation* नावाचे पुस्तक बघा.

[१] अनुवादकाची टिप्पणी: ॲलन ट्युरिंगना संगणक विज्ञानाचे वडील (father) मानले जाते. मी स्वतः पहिल्यांदा जेव्हा श्री. ट्युरिंग विषयी ऐकले होते तेव्हाच मला खूप प्रेरणा मिळाली होती आणि पुढे त्याचे रुपांतर मी थिअरेटिकल कॉंप्युटर सायन्स (theoretical computer science) मध्ये पीएचडी करण्याचा निर्णय घेण्यात झाले. त्यांच्या आयुष्यावर आधारित 'द इमिटेशन गेम' ('The Imitation Game') नावाचा चित्रपट २०१४ साली प्रदर्शित झाला.

तुम्ही काय करू शकता ह्याची तुम्हाला कल्पना येण्यासाठी आपण काही गणितीय रिकर्सिव्ह (recursive) फंक्शन्स पाहूया ज्यांची व्याख्या (definition) रिकर्सिव्ह म्हणजे स्वतःचा संदर्भ वापरून दिलेली असते (हे पुढे स्पष्ट होईल). मराठीत आपण ह्याला स्वसंदर्भयुक्त व्याख्या म्हणूया. शब्दशः आणि पूर्णपणे स्वसंदर्भयुक्त व्याख्या ही साधारणपणे तार्किकदृष्ट्या गोल-गोल फिरवणारी आणि अर्थहीन असते:

तात्विक: तात्विक गोष्टीचे वर्णन करण्यासाठी वापरलेला शब्द.

जर अशी व्याख्या तुम्ही शब्दकोशात पाहिली तर तुम्ही बहुतेक चिडालही. पण जर तुम्ही फॅक्टोरियल (factorial) फलनाची (म्हणजे गणितातील फंक्शनची[२]) व्याख्या बघितली, तर तुम्हाला खालील व्याख्या मिळू शकते, हे फलन ! चिन्हाने दर्शवले जाते:

$$0! = 1$$
$$n! = n(n-1)!$$

शब्दांत सांगायचे झाले तर, 0 चे फॅक्टोरियल 1 आहे, आणि इतर n चे फॅक्टोरियल हे n गुणिले $n-1$ चे फॅक्टोरियल आहे.

म्हणजे 3! हे 3 गुणिले 2!, जे आहे 2 गुणिले 1!, जे आहे 1 गुणिले 0!, जे आहे 1. सर्व माहिती एकत्र करून: 3! बरोबर 3 गुणिले 2 गुणिले 1 गुणिले 1, ज्याचे उत्तर 6 आहे.

तुम्ही कशाचीही स्वसंदर्भयुक्त व्याख्या (recursive definition) लिहू शकत असाल तर तुम्ही ते शोधून काढायला पायथॉन प्रोग्राम लिहून शकता. पहिली पायरी म्हणजे परॅमीटर्स ठरवणे. ह्याठिकाणी हे स्पष्ट आहे की `factorial` फंक्शन एक इंटिजर घेईल:

```
def factorial(n):
```

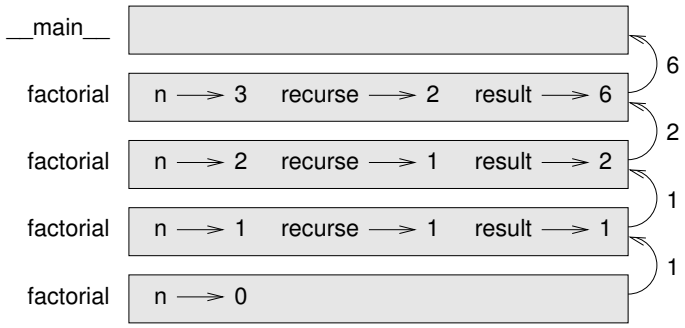
जर अर्ग्युमेंट शून्य असेल तर आपल्याला फक्त 1 ही व्हॅल्यू रिटर्न करावी लागते:

```
def factorial(n):
    if n == 0:
        return 1
```

नाहीतर—हा आहे कुतूहलजन्य भाग— आपल्याला $n-1$ चे फॅक्टोरियल शोधायला एक रिकर्सिव्ह कॉल करावा लागतो आणि त्या उत्तराला n ने गुणावे लागते:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        return result
```

[२]पुढील मराठी विश्वकोश लिंकवर फलनाविषयी अजून माहिती बघा: https://marathivishwakosh.org/21979/.



आकृती ६.१: स्टॅक डायग्राम (Stack diagram).

ह्या प्रोग्रामचा फ्लो-ऑफ-एक्सेक्युशन प्रकरण ५.८ मधल्या `countdown` फंक्शनसारखाच आहे. जर आपण `factorial` फंक्शन 3 व्हॅल्यू वापरून कॉल केले:

> 3 ही 0 हून वेगळी आहे; `n-1` चे फॅक्टोरियल शोधण्यासाठी दुसरी ब्रांच घ्या...
>
> > 2 ही 0 हून वेगळी आहे; `n-1` चे फॅक्टोरियल शोधण्यासाठी दुसरी ब्रांच घ्या...
> >
> > > 1 ही 0 हून वेगळी आहे; `n-1` चे फॅक्टोरियल शोधण्यासाठी दुसरी ब्रांच घ्या...
> > >
> > > > 0 ही 0 आहे; पहिली ब्रांच घ्या आणि अजून कोणतेही रिकर्सिव्ह कॉल्स *न* करता 1 रिटर्न करा.
> > >
> > > रिटर्न व्हॅल्यू 1 आहे; तिला n ने म्हणजेच 1 ने गुणून उत्तर रिटर्न करा.
> >
> > रिटर्न व्हॅल्यू 1 आहे; तिला n ने म्हणजेच 2 ने गुणून उत्तर रिटर्न करा.
>
> रिटर्न व्हॅल्यू 2 आहे; तिला n ने म्हणजेच 3 ने गुणून उत्तर रिटर्न करा; ह्याच फंक्शन कॉलने ही पूर्ण प्रक्रिया सुरू झाली म्हणून अंतिम उत्तर 6 आहे.

वरील फंक्शन कॉल्सशी संबंधित स्टॅक डायग्राम आकृती ६.१ मध्ये आहे.

स्टॅकमध्ये, रिटर्न व्हॅल्यूझ गोल-बाणांद्वारे वरच्या बाजूस पाठवलेल्या दाखवल्या आहेत. प्रत्येक फ्रेममध्ये रिटर्न व्हॅल्यू ही `result` व्हेरिएबलची व्हॅल्यू आहे, जी `n` आणि `recurse` ह्या व्हेरिएबल्स मधील व्हॅल्यूझचा गुणाकार आहे.

शेवटच्या फ्रेममध्ये `recurse` आणि `result` ही स्थानिक व्हेरिएबल्स अस्तित्वात नाहीत कारण ज्या ब्रांचमध्ये ती बनवली जातात ती एक्सेक्युटच होत नाही.

६.६ लीप ऑफ फेथ (Leap of faith, भरवशाची-झेप)

फ्लो-ऑफ-एक्सेक्युशनचा माग घेणे ही प्रोग्राम वाचण्याची एक पद्धत आहे; पण ती चटकन संभ्रमात पाडू शकते. ह्यावर उपाय म्हणजे 'लीप ऑफ फेथ' ('leap of faith', भरवशाची-झेप). असा माग घेताघेता तुम्ही जेव्हा एका फंक्शनवर येता, तेव्हा फ्लो-ऑफ-एक्सेक्युशनचा माग घेण्यापेक्षा तुम्ही असे *गृहीत* धरा (*assume* करा) की ते फंक्शन आपले काम बरोबर करते आणि बरोबर उत्तर रिटर्न करते.

वास्तविक पाहता, तुम्ही बिल्ट-इन फंक्शन्स वापरत होता तेव्हा त्यांच्यावर भरवसा ठेवलाच होता, नाही का? जेव्हा तुम्ही `math.cos` किंवा `math.exp` कॉल करता, तेव्हा तुम्ही त्यांच्या डेफनिशन्स जाऊन तपासत नाही. तुम्ही हे *गृहीत* धरून चालता की ती फंक्शन्स बरोबर आहेत कारण ती लिहिणारे प्रोग्रामर्स चांगले आहेत.

हेच, तुम्ही स्वतःचे फंक्शन कॉल करता तेव्हाही लागू होते. उदा., विभाग ६.४ मध्ये आपण `is_divisible` नावाचे फंक्शन लिहिले जे एका संख्येला दुसऱ्या संख्येने पूर्ण भाग जातो का हे तपासते. कोड नीट तपासून आणि टेस्ट करून

आपली एकदाची खात्री पटली की ते फंक्शन बरोबर आहे तर आपण त्या फंक्शनच्या डेफनिशनकडे परत कधीच न बघता निःसंकोचपणे वापरू शकतो.

आणि तेच रिकर्सिव्ह प्रोग्रामला पण लागू होते. जेव्हा तुम्ही रिकर्सिव्ह कॉलला पोहोचता, फ्लो-ऑफ-एक्सेक्युशनच्या मागे जाण्याऐवजी तुम्ही असे गृहीत धरले पाहिजे की रिकर्सिव्ह कॉल बरोबर चालतो (आणि बरोबर उत्तर पाठवतो); आणि मग स्वत:ला विचारा 'जर मी $n-1$ चे फॅक्टोरियल बरोबर शोधून काढू शकतो असे गृहीत धरले तर n चे फॅक्टोरियल बरोबर काढू शकतो का?' फॅक्टोरियलच्या व्याख्येनुसार हे सरळ आहे की हे करण्यासाठी तुम्ही n ने गुणू शकता.

साहजिक आहे, फंक्शन लिहून होण्याच्या आधीच ते व्यवस्थित चालते हे गृहीत धरणे थोडे विचित्रच आहे, पण त्यामुळेच आपण त्याला भरवशाची झेप म्हणतो!

६.७ अजून एक उदाहरण (One more example)

वरील `factorial` फंक्शननंतर स्वसंदर्भयुक्त व्याख्या असलेले सुप्रसिद्ध गणितीय फंक्शन म्हणजे `fibonacci` (फिबोनाची, इटालियन आडनाव), ज्याची व्याख्या खाली दिली आहे (पुढील विकिपीडिया लिंकसुद्धा बघा[3]: http://en.wikipedia.org/wiki/Fibonacci_number):

$$\begin{aligned}\text{fibonacci}(0) &= 0\\ \text{fibonacci}(1) &= 1\\ \text{fibonacci}(n) &= \text{fibonacci}(n-1) + \text{fibonacci}(n-2)\end{aligned}$$

ह्याचे भाषांतर आपण पायथॉनमध्ये असे करू शकतो:

```
def fibonacci(n):
    if n == 0:
        return 0
    elif  n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

जर तुम्ही इथे फ्लो-ऑफ-एक्सेक्युशनचा माग घेण्याचा प्रयत्न केला तर तुमचे डोके फुटेल. पण लीप-ऑफ-फेथ घेऊन तुम्ही हे गृहीत धरले की दोन रिकर्सिव्ह कॉल्स बरोबर चालतात तर हे स्पष्ट होते की आपल्याला त्यांची बेरीज करून बरोबर उत्तर मिळते.

६.८ टाइप तपासणे (Checking types)

जर आपण `factorial` कॉल केले आणि 1.5 हे अर्ग्युमेंट दिले तर काय होईल?

```
>>> factorial(1.5)
RuntimeError: Maximum recursion depth exceeded
```

हे तर इन्फिनेट रिकर्शन (infinite recursion) सारखे दिसतेय. पण असे कसे? फंक्शनमध्ये तर `n == 0` बेस-केस आहे. पण जर `n` इंटिजर दर्शवत नसेल तर आपली बेस-केस *चुकते (हुकते)* आणि अखंडपणे रिकर्शन होते.

पहिल्या रिकर्सिव्ह कॉलमध्ये `n` ची व्हॅल्यू 0.5 आहे. पुढच्यात -0.5 आहे, नंतर -1.5, आणि तिथून पुढे ती अजून कमी (अजून ऋण) होत जाते पण शून्य कधीच होत नाही.

[3] अनुवादकाची टिप्पणी: फिबोनाची नंबर्सची (Fibonacci numbers) भारतीय गणितज्ञांना खूप आधीच माहिती होती. विकिपीडिया-लिंकवर तुम्हाला अधिक माहिती मिळेल.

आपल्याकडे दोन पर्याय आहेत. आपण `factorial` फंक्शन जनरलाइझ (generalize) करून ते फ्लोटिंग-पॉइंट संख्याही घेईल असे बनवू शकतो, किंवा आपण `factorial` मध्ये अर्ग्युमेंटचा टाइप तपासू शकतो. पहिल्या पर्यायाला गॅमा (gamma, γ) फंक्शन म्हणतात आणि ते आपण ह्या पुस्तकात बघणार नाही आहोत. तर आपण दुसरा पर्याय निवडूया.

आपण `isinstance` नावाचे बिल्ट-इन फंक्शन वापरून अर्ग्युमेंटचा टाइप तपासू शकतो. आपल्याला अर्ग्युमेंट धन (positive) आहे ह्याचीसुद्धा खात्री करावी लागेल:

```
def factorial(n):
    if not isinstance(n, int):
        print('Factorial is only defined for integers.')
        return None
    elif n < 0:
        print('Factorial is not defined for negative integers.')
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

पहिली बेस-केस (base case) इंटिजर नसलेली अर्ग्युमेंट्स हाताळते; दुसरी ऋण इंटिजर्स हाताळते. दोन्हींत, प्रोग्राम एरर मेसेज दाखवून `None` रिटर्न करतो जेणेकरून कॉलर (caller) ला हे कळेल की काहीतरी बिघडले आहे:

```
>>> print(factorial('fred'))
Factorial is only defined for integers.
None
>>> print(factorial(-2))
Factorial is not defined for negative integers.
None
```

जर दोन्ही चाचण्या पास झाल्या तर आपल्याला हे खात्रीपूर्वक सांगू शकतो की n ऋण नसलेली पूर्णांक संख्या (non-negative integer) आहे, म्हणजेच रिकर्शनचा अखेरीस अंत होतो हे आपण सिद्ध करू शकतो.

हा प्रोग्राम **गार्डियन** (**guardian**, संरक्षक) नावाचा प्रोग्रामिंग पॅटर्न (pattern) दर्शवतो. पहिली दोन कंडिश्नल्स संरक्षकाचे काम करतात, जे पुढे येणाऱ्या कोड-चे इन्फिनेट रिकर्शन होण्यापासून संरक्षण करतात. गार्डियन्समुळे कोड-ची अचूकता सिद्ध करता येणे शक्य होते.

विभाग ११.४ मध्ये आपण एरर मेसेजेस दाखवण्यापेक्षा चांगला पर्याय बघणार आहोत: एक्सेप्शन रेझ्ड करणे (raise an exception).

६.९ डीबगिंग (Debugging)

मोठ्या प्रोग्रामचे लहान फंक्शन्समध्ये विभाजन केल्याने डीबगिंगसाठी ती फंक्शन्स चेकपॉइंट्सची (checkpointची, तपासनाक्याची) भूमिका करतात. जर एखादे फंक्शन नीट काम नसेल करत तर खालील तीन शक्यतांचा विचार करावा:

- फंक्शनला मिळणाऱ्या अर्ग्युमेंट्समध्ये चूक आहे; एखाद्या प्रीकंडिशनचे (precondition) उल्लंघन होतेय.
- फंक्शनमध्येच चूक आहे; एखाद्या पोस्टकंडिशनचे (postcondition) उल्लंघन होतेय.
- रिटर्न व्हॅल्यूमध्ये चूक आहे किंवा ती अपेक्षितपणे वापरली जात नाहीये.

पहिली शक्यता दूर करण्यासाठी तुम्ही फंक्शनच्या सुरुवातीला काही प्रिंट स्टेटमेंट्स टाकून परॅमीटर्सच्या व्हॅल्यूझ आणि टाइप्स बघू शकता. किंवा तुम्ही प्रीकंडिशन्स तपासणारा कोड प्रत्यक्षपणे लिहू शकता.

जर परॅमीटर्स ठीक दिसत असतील तर प्रत्येक रिटर्न स्टेटमेंटच्या आधी एक प्रिंट स्टेटमेंट लिहून रिटर्न व्हॅल्यू बघा. जर शक्य असेल तर अपेक्षित उत्तर स्वतःच शोधा. फंक्शनला अशा व्हॅल्यूझने कॉल करा ज्यांनी उत्तर तपासणे सोपे पडेल (विभाग ६.२ मध्ये पाहिल्याप्रमाणे).

जर फंक्शन ठीक चालत असेल तर फंक्शन कॉलकडे नीट लक्ष देऊन हे बघा की रिटर्न व्हॅल्यू व्यवस्थितपणे वापरली आहे (किंवा हे बघा ती वापरली तरी आहे ना!).

फंक्शनच्या सुरुवातीला आणि शेवटी प्रिंट स्टेटमेंट्स टाकल्याने फ्लो-ऑफ-एक्सेक्युशन जास्ती स्पष्ट दिसतो. उदा., आपण `factorial` मध्ये काही प्रिंट स्टेटमेंट्स टाकूया:

```
def factorial(n):
    space = ' ' * (4 * n)
    print(space, 'factorial', n)
    if n == 0:
        print(space, 'returning 1')
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        print(space, 'returning', result)
        return result
```

इथे `space` ही स्पेस कॅरेक्टर्सची स्ट्रिंग आहे जिच्याने आउटपुटचे व्यवस्थित इंडेंटेशन (indentation) होते (`n` च्या प्रमाणात स्पेसेस दिसतात):

```
                factorial 4
            factorial 3
         factorial 2
     factorial 1
 factorial 0
 returning 1
     returning 1
         returning 2
             returning 6
                 returning 24
```

जर तुमचा फ्लो-ऑफ-एक्सेक्युशनने गोंधळ उडाला असेल तर ह्याप्रकारचे आउटपुट उपयोगी पडू शकते. चांगली स्कॅफोल्डिंग (scaffolding) बनवायला थोडा वेळ लागतो पण तिच्याने डीबगिंगचा बराच वेळ वाचतो.

६.१० शब्दार्थ

टेंपररी व्हेरिएबल (temporary variable): एका मोठ्या कॅल्क्युलेशन (calculation) च्या दरम्यान लागणाऱ्या व्हॅल्यूझ ठेवण्यासाठी वापरलेले व्हेरिएबल.

डेड कोड (dead code): प्रोग्रामचा तो भाग जो कधीच रन नाही होऊ शकत, सहसा कारण की तो एका `return` स्टेटमेंटनंतर येतो.

इन्क्रिमेंटल डेव्हेलपमेंट (incremental development): असा प्रोग्राम डेव्हेलपमेंट प्लान (program development plan) ज्याचा उद्देश अतिडीबगिंग टाळणे हा आहे; ह्यात एका वेळी थोडाच कोड टाकून टेस्ट केला जातो.

स्कॅफोल्डिंग (scaffolding): प्रोग्राम डेव्हेलपमेंटच्या वेळी वापरला जाणारा कोड जो अंतिम प्रोग्रामचा भाग नसतो.

गार्डियन (guardian, संरक्षक): असा प्रोग्रामिंग पॅटर्न (pattern, म्हणजे कित्ता/नमुना) ज्यात कंडिश्नल स्टेटमेंट वापरून एरर घडवणाऱ्या परिस्थितींना हाताळले जाते.

६.११ प्रश्नसंच (Exercises)

प्रश्न ६.१. *खालील प्रोग्रामची स्टॅक डायग्राम काढा. हा प्रोग्राम काय प्रिंट करतो?*

```
def b(z):
    prod = a(z, z)
    print(z, prod)
    return prod

def a(x, y):
    x = x + 1
    return x * y

def c(x, y, z):
    total = x + y + z
    square = b(total)**2
    return square

x = 1
y = x + 1
print(c(x, y+3, x+y))
```

प्रश्न ६.२. *Ackermann फंक्शन $A(m,n)$ ची व्याख्या अशी आहे:*

$$A(m,n) = \begin{cases} n+1 & \text{जर } m = 0 \\ A(m-1,1) & \text{जर } m > 0 \text{ आणि } n = 0 \\ A(m-1,A(m,n-1)) & \text{जर } m > 0 \text{ आणि } n > 0. \end{cases}$$

पुढील लिंक बघा: http://en.wikipedia.org/wiki/Ackermann_function. आणि `ack` *नावाचे फंक्शन लिहा जे Ackermann फंक्शनची किंमत शोधून काढते. तुमचे फंक्शन वापरून* `ack(3, 4)` *शोधा, जे 125 असले पाहिजे. नंतर* `m` *आणि* `n` *च्या मोठ्या व्हॅल्यूझसाठी काय होते ते बघा. उत्तर: http://thinkpython2.com/code/ackermann.py.*

प्रश्न ६.३. *पॅलिंड्रोम (palindrome) म्हणजे असा इंग्रजी शब्द ज्याचे सरळ आणि उलटे स्पेलिंग सारखेच असते, जसे 'noon' आणि 'redivider'. ह्याची स्वसंदर्भयुक्त व्याख्या (recursive definition) अशी: जर पहिले आणि शेवटचे अक्षर सारखेच असले आणि मधला भाग पॅलिंड्रोम असला तर तो शब्द पॅलिंड्रोम असतो.*

खालील फंक्शन्स एक स्ट्रिंग अर्ग्युमेंट घेऊन अनुक्रमे त्यातील पहिले अक्षर, शेवटचे अक्षर, आणि मधला भाग रिटर्न करतात:

```
def first(word):
    return word[0]

def last(word):
    return word[-1]

def middle(word):
    return word[1:-1]
```

त्यांची कार्यपद्धती आपण प्रकरण ८ मध्ये पाहू.

१. `palindrome.py` *नावाच्या फाइलमध्ये ही फंक्शन्स लिहा आणि त्यांना टेस्ट करा. जर तुम्ही* `middle` *फंक्शनला दोन अक्षरांच्या स्ट्रिंगने कॉल केले तर काय होते? एक अक्षराच्या? आणि जर रिकाम्या स्ट्रिंगने, जी* `''` *अशी लिहिली जाते आणि ज्यात शून्य अक्षरे असतात?*

२. `is_palindrome` *नावाचे फंक्शन लिहा जे एक स्ट्रिंग अर्ग्युमेंट घेते आणि जर ती पॅलिंड्रोम असेल तर* `True` *रिटर्न करते, नाहीतर* `False`*. हे आठवा की तुम्ही* `len` *हे बिल्ट-इन फंक्शन वापरून स्ट्रिंगची लांबी मोजू शकता.*

उत्तर: http://thinkpython2.com/code/palindrome_soln.py.

प्रश्न ६.४. *अनुवादकाची नोंद: आपण हा प्रश्न बघण्याआधी मराठीत एक व्याख्या स्पष्ट करूया. समजा a आणि b ह्या पूर्णांक संख्या (integer) आहेत; आपण a ही b चा घात (power) आहे असे तेव्हा म्हणतो जेव्हा आपल्याला a ला $a = b^n$ असे व्यक्त करता येते. ह्याची पर्यायी स्वसंदर्भयुक्त व्याख्या (recursive definition) खाली आहे.*

a ही संख्या b चा घात (power) तेव्हा असते जेव्हा तिला b ने पूर्णपणे भाग जात असेल आणि a/b ही संख्यासुद्धा b चा घात असेल; `a` *आणि* `b` *परॅमीटर्स घेऊन जर* `a` *ही* `b` *चा घात (power) असेल तर* `True` *रिटर्न करणारे* `is_power` *नावाचे फंक्शन लिहा. नोंद: तुम्हाला बेस-केसचा काळजीपूर्वक विचार करावा लागेल.*

प्रश्न ६.५. *समजा a आणि b ह्या दिलेल्या दोन पूर्णांक संख्या (integers) आहेत; ह्यांचा महत्तम सामायिक विभाजक (मसावि, greatest common divisor, GCD) म्हणजे त्या दोन्ही संख्यांना पूर्णपणे (म्हणजे बाकी शून्य ठेवून) भागणारी सर्वांत मोठी पूर्णांक संख्या होय.*

मसावि (GCD) शोधण्याचा एक मार्ग पुढीलप्रमाणे. जर a ला b ने भागल्यावर बाकी r असेल तर $gcd(a,b) = gcd(b,r)$, शब्दांत a आणि b ह्यांचा मसावि आणि b आणि r ह्यांचा मसावि हे सारखेच असतात. बेस-केस (base case) म्हणून आपण $gcd(a,0) = a$ हे वापरू शकतो.

परॅमीटर्स `a` *आणि* `b` *घेऊन त्यांचा महत्तम सामायिक विभाजक काढणारे* `gcd` *नावाचे फंक्शन लिहा.*

आभार: हा प्रश्न आबेल्सन आणि सुस्मन (Abelson आणि Sussman) ह्यांच्या Structure and Interpretation of Computer Programs *पुस्तकातील एका उदाहरणावर आधारित आहे.*

प्रकरण ७

इटरेशन (Iteration)

हे प्रकरण इटरेशन (iteration) विषयी आहे, म्हणजे काही स्टेटमेंट्स पुनःपुन्हा रन करणे (पुनरावृत्ती करणे). आपण कोड-ची पुनरावृत्ती करण्याची एक पद्धत विभाग ५.८ मध्ये पाहिली आहे, ती म्हणजे रिकर्शन (recursion). अजून एक पद्धत आपण विभाग ४.२ मध्ये पाहिली आहे, ती म्हणजे `for` लूप (loop). ह्या प्रकरणात आपण आणखी एक पद्धत बघणार आहोत: `while` स्टेटमेंट. पण त्याआधी आपण व्हेरिएबल असाइनमेंट (variable assignment) विषयी अजून थोडी चर्चा करूया.

७.१ रीअसाइनमेंट (Reassignment)

तुम्हाला ह्याची कल्पना आलीच असेल की एका व्हेरिएबलला एकापेक्षा अधिक असाइनमेंट करणे वैध आहे. नवीन असाइनमेंटनंतर व्हेरिएबल नवीन व्हॅल्यू दर्शवते (आणि जुनी व्हॅल्यू दर्शवण्याचे थांबवते).

```
>>> x = 5
>>> x
5
>>> x = 7
>>> x
7
```

आपण पहिल्यांदा `x` दाखवतो तेव्हा त्याची व्हॅल्यू 5 आहे, आणि दुसऱ्यांदा 7.

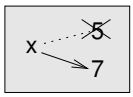
स्टेट डायग्राममध्ये **रीअसाइनमेंट** (**reassignment**) कशी दिसते हे आकृती ७.१ दर्शवते.

ह्याठिकाणी आपण एक कॉमन (साधारण) गैरसमज दूर करूया. पायथॉनमध्ये बरोबरचे चिन्ह (=) असाइनमेंटसाठी वापरत असल्यामुळे, आपण `a = b` ह्याचा अर्थ गणितीय समानता/एकरुपता (equality) म्हणजेच 'a आणि b हे सारखेच आहेत' असा लावण्याची दाट शक्यता आहे. पण हा अर्थ चुकीचा आहे.

पहिली गोष्ट म्हणजे, समानतेचा (equality) संबंध हा दोन्ही बाजूंनी सारखाच असतो पण असाइनमेंटचा नाही. उदा., गणितात जर $a = 7$ तर $7 = a$. पण पायथॉनमध्ये `a = 7` हे स्टेटमेंट वैध आहे आणि `7 = a` हे नाही.[१]

अजून म्हणजे, गणितात समानतेचे विधान हे एकतर नेहमी सत्य किंवा नेहमी असत्य असते. जर आता $a = b$ असेल तर a आणि b हे नेहमी सारखेच असतील. पायथॉनमध्ये असाइनमेंट स्टेटमेंट दोन व्हेरिएबल्सना सारखे बनवते, पण ते नेहमी सारखेच राहतील असे नाही:

[१] अनुवादकाची टिप्पणी: आपण पायथॉनमध्ये = हे चिन्ह ← असे आहे अशी कल्पना करू शकतो; आणि `a` ← 7 चा अर्थ असा लावायचा की `a` व्हेरिएबलला 7 व्हॅल्यू मिळाली. हे फक्त प्रोग्राम समजण्यासाठी; पायथॉनमध्ये ← हे चिन्ह अवैध आहे, तर ते वापरू नका.



आकृती ७.१: स्टेट डायग्राम.

```
>>> a = 5
>>> b = a    # a and b are now equal
>>> a = 3    # a and b are no longer equal
>>> b
5
```

तिसरी ओळ `a` ची व्हॅल्यू बदलते पण `b` ची व्हॅल्यू तीच राहते, म्हणून इथून पुढे ते सारखे नाहीयेत.

रीअसाइनमेंट जरी कधीकधी फायदेशीर असली तरी काळजीपूर्वक वापरली पाहिजे. जर व्हेरिएबल्सच्या व्हॅल्यूझ वारंवार बदलत असल्या तर कोड वाचायला आणि डीबग करायला अवघड ठरू शकतो.

७.२ व्हेरिएबल अपडेट करणे (Updating a variable)

सामान्यप्रकारची रीअसाइनमेंट म्हणजे **अपडेट** (**update**) ज्यात नवीन व्हॅल्यू जुन्या व्हॅल्यूनुसार ठरते.

```
>>> x = x + 1
```

ह्याचा अर्थ हा की '`x` ची सध्याची व्हॅल्यू घ्या, तिच्यात १ मिळवा, आणि `x` ला नवीन व्हॅल्यू देऊन अपडेट करा (`x` नवीन व्हॅल्यू दर्शवते).'

प्रोग्राममध्ये नसलेले व्हेरिएबल अपडेट करायचा प्रयत्न केला तर एरर येतो, कारण पायथॉन उजव्या बाजूची व्हॅल्यू पहिले शोधतो, मग ती `x`ला असाइन करतो:

```
>>> x = x + 1
NameError: name 'x' is not defined
```

एखादे व्हेरिएबल अपडेट करायच्या आधी तुम्ही त्याला **इनिशलाइझ** (**initialize**) करणे आवश्यक आहे, साधारणपणे एका साध्या असाइनमेंटने:

```
>>> x = 0
>>> x = x + 1
```

व्हेरिएबलमध्ये १ मिळवून त्याला अपडेट करण्याला **इन्क्रिमेंट** (**increment**) म्हणतात; १ वजा करण्याला **डिक्रिमेंट** (**decrement**) म्हणतात.

७.३ `while` स्टेटमेंट (The `while` statement)

वारंवार करायचे काम ऑटोमेट (automate, स्वयंचलीकरण) करायला कॉंप्युटर वापरला जातो. वारंवार तेच किंवा सारखेच काम चुका *न* करता करणे ह्यात माणसे वाईट आणि कॉंप्युटर्स कुशल असतात. कॉंप्युटर प्रोग्राममध्ये पुनःपुन्हा करण्याला **इटरेशन** (**iteration**) म्हणतात.

आपण `countdown` आणि `print_n` ही दोन फंक्शन्स पाहिली आहेत ज्यात रिकर्शन वापरून काहीतरी पुनःपुन्हा केले आहे. इटरेशन हे नेहमी लागणारे असल्यामुळे पायथॉनमध्ये त्यासाठी काही सोयी आहेत. एक म्हणजे `for` स्टेटमेंट जे आपण विभाग ४.२ मध्ये पाहिले. ते आपण नंतर परत पाहू.

दुसरे म्हणजे `while` स्टेटमेंट. खाली `countdown` फंक्शन `while` वापरून लिहिले आहे:

```
def countdown(n):
    while n > 0:
        print(n)
        n = n - 1
    print('Blastoff!')
```

तुम्ही `while` स्टेटमेंट इंग्रजीमध्ये असल्यासारखेच वाचू शकता (while म्हणजे जोपर्यंत). वरील `while` स्टेटमेंटचा अर्थ असा आहे: 'While `n` is greater than 0, display the value of `n` and then decrement `n`. When you get to 0, display the word `Blastoff!`' ('जोपर्यंत `n` शून्याहून मोठे आहे, `n` ची व्हॅल्यू दाखवा आणि नंतर `n`ला डिक्रिमेंट करा. शून्यास पोहोचल्यावर `Blastoff!` असे दाखवा.')

जास्ती तंतोतंतपणे, एका `while` स्टेटमेंटचा फ्लो-ऑफ-एक्सेक्युशन असा असतो:

१. कंडिशन सत्य आहे का असत्य ते शोधा.

२. जर असत्य असेल तर `while` स्टेटमेंटमधून ताबडतोब बाहेर पडा आणि पुढील स्टेटमेंटपासून एक्सेक्युशन चालू ठेवा.

३. जर सत्य असेल तर बॉडी रन करा आणि परत पायरी १ वर जा.

अशा प्रकारच्या कृतीला लूप (loop, वळसा) म्हटले जाते कारण तिसरी पायरी वळसा घालून परत वर जाते.

लूप-च्या बॉडीमध्ये काही व्हेरिएबल्सची व्हॅल्यू बदलली गेली पाहिजे जेणेकरून कंडिशन अखेरीस असत्य ठरून लूप-चा अंत होईल. नाहीतर लूप अखंडपणे चालू राहील, ज्याला **इन्फिनेट लूप** (**infinite loop**) म्हणतात. संगणक वैज्ञानिकांना शँपूवरील 'lather, rinse, repeat' ह्या सूचनांच्या इन्फिनेट-लूप-ची गंमत वाटते.

आपण `countdown` फंक्शनमधल्या लूप-चा अंत होतो हे सिद्ध करू शकतो: जर `n` शून्य किंवा ऋण असेल तर लूप रनच होत नाही. नाहीतर प्रत्येक वेळा लूपमधून गेल्यावर `n` लहान होतो, म्हणजे अखेरीस शून्य होतो.

पण काही लूप्सबद्दल अशी सिद्धता देणे इतके सोपे नसते. उदा.:

```
def sequence(n):
    while n != 1:
        print(n)
        if n % 2 == 0:        # n is even
            n = n / 2
        else:                 # n is odd
            n = n*3 + 1
```

ह्या लूप-ची कंडिशन `n != 1` आहे, म्हणजे जोपर्यंत `n` ची व्हॅल्यू `1` होत नाही तोपर्यंत लूप चालू राहील.

लूपमधून जाताना प्रत्येकवेळी प्रोग्राम `n` ची व्हॅल्यू दाखवतो आणि तपासतो `n` सम आहे का विषम. जर सम असेल तर त्याला २ ने भागले जाते, आणि नाहीतर विषम असेल आणि त्याला ३ ने गुणून त्यात १ मिळवला जाईल, म्हणजेच त्याला `n*3 + 1` ने बदलले जाईल. उदा., जर `sequence` फंक्शन 3 व्हॅल्यूने कॉल केले तर, `n` च्या व्हॅल्यूझ अनुक्रमे $3, 10, 5, 16, 8, 4, 2, 1$ असतील.

आणि `n` कधीकधी वाढतो आणि कधीकधी कमी होतो म्हणून '`n` अखेरीस 1 होईल आणि प्रोग्रामचा अंत होईल' ह्याची उघड सिद्धता दिसत नाही. पण `n` च्या काही विशिष्ठ किंमतींसाठी आपण वरील प्रोग्रामचा अंत होतो हे सिद्ध करू शकतो. उदा., जर सुरुवातीची `n` ची व्हॅल्यू जर २ चा घात असेल (a power of 2, म्हणजेच तिला 2^k ह्या स्वरुपात लिहिता येत असेल), तर लूपमधून प्रत्येकवेळी जाताना `n` ची व्हॅल्यू समच राहील आणि अखेरीस 1 होईल. वरील परिच्छेदातील उदाहरणात शेवटचा भाग असाच, 16 पासून सुरू होणारा आहे.

खरा अवघड प्रश्न हा आहे: आपण हे सिद्ध करू शकतो का की 'हा प्रोग्राम *कोणतीही* धन पूर्णांक संख्या त्यास अर्ग्युमेंट म्हणून `n` मध्ये दिली तरी त्याचा अंत होतो.' आणि तुम्हाला गंमत वाटेल पण आतापर्यंत कोणीही ह्याची सिद्धता देऊ शकले नाहीये *आणि* कोणीही हे सिद्ध करू शकले नाहीये की हे विधान असत्य आहे! पुढील लिंक बघा: http://en.wikipedia.org/wiki/Collatz_conjecture.

अनुवादकाची टिप्पणी: सुप्रसिद्ध अशा व्हेरीटासियम (Veritasium) ह्या युट्युब चॅनेलमध्ये ह्यावर 'The Simplest Math Problem No One Can Solve - Collatz Conjecture' नावाचा व्हिडिओ प्रकाशित केला आहे; त्याची लिंक https://youtu.be/094y1Z2wpJg.)

सराव म्हणून विभाग ५.८ मधील `print_n` नावाचे फंक्शन रिकर्शन ऐवजी इटरेशन वापरून लिहा.

७.४ `break` (ब्रेक)

कधीकधी आपल्याला लूप-च्या बॉडीच्या मध्यात पोहोचेपर्यंत सांगता येत नाही की लूप संपवायचा आहे. त्याठिकाणी तुम्ही `break` स्टेटमेंट वापरून लूप-च्या बाहेर उडी मारू शकता.

उदा., जर तुम्हाला युझरकडून `done` लिहित नाही तोपर्यंत इनपुट घ्यायचे आहे. तुम्ही असे लिहू शकता:

```
while True:
    line = input('> ')
    if line == 'done':
        break
    print(line)

print('Done!')
```

लूप-ची कंडिशन `True` आहे, जी नेहमीच सत्य असते, म्हणून जोपर्यंत ब्रेक स्टेटमेंट लागत नाही तोपर्यंत लूप चालू राहतो.

लूपमधून प्रत्येकवेळी जाताना युझरला > असा प्रॉम्प्ट दाखवला जातो. जर युझरने `done` असे लिहिले तर `break` स्टेटमेंट लूपमधून बाहेर काढते. नाहीतर जे पण युझरने लिहिलेले आहे प्रोग्राम त्याची कॉपी दाखवतो आणि परत लूप-च्या वर जातो. एका रन-चा नमुना:

```
> not done
not done
> done
Done!
```

अशाप्रकारे `while` लूप लिहिणे कॉमन आहे कारण तुम्ही कंडिशन लूपमध्ये कुठेही तपासू शकता (फक्त सुरुवातीलाच नाही) आणि तुम्ही थांबण्याची कंडिशन नकारार्थी स्वरुपात व्यक्त करण्याऐवजी ('हे होईपर्यंत चालू द्या' ऐवजी) होकारार्थी स्वरुपात व्यक्त करू शकता ('हे झाले की थांबा').

७.५ वर्गमूळ (Square root)

संख्यात्मक गणन करणाऱ्या प्रोग्राममध्ये एका एस्टिमेट (estimate, अंदाजी-उत्तर) पासून सुरुवात करून त्यास हळूहळू सुधारण्यासाठी लूप वापरतात.

उदा., न्यूटनची पद्धत वापरून आपण वर्गमूळ शोधू शकतो. समजा तुम्हाला a चे वर्गमूळ शोधायचे आहे. तुमचा एस्टिमेट x असेल तर तुम्ही त्याला खालील सूत्राने सुधारू शकता (y हा नवीन एस्टिमेट आहे):

$$y = \frac{x + a/x}{2}$$

उदा., जर a ची किंमत 4 असेल आणि x ची 3 तर:

```
>>> a = 4
>>> x = 3
```

```
>>> y = (x + a/x) / 2
>>> y
2.16666666667
```

हे उत्तर बरोबर उत्तराच्या जास्ती जवळ आहे ($\sqrt{4} = 2$). जर आपण हीच क्रिया नवीन एस्टिमेट वापरून परत केली तर अजून जवळ जाऊ:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00641025641
```

अजून, परत काहीवेळा केल्यानंतरचा एस्टिमेट जवळजवळ अचूकच आहे:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00001024003
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00000000003
```

साधारणपणे, आपल्याला हे किती वेळा करावे लागेल ह्याची पूर्वकल्पना नसते, पण आपण तिथे पोहोचल्यावर ते ओळखू शकतो कारण आपला एस्टिमेट बदलायचा थांबतो:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
```

जेव्हा `y == x` होते, तेव्हा आपण थांबू शकतो. खालचा लूप सुरुवातीच्या `x` ह्या एस्टिमेटने सुरू होतो आणि बदलायचा थांबत नाही तोपर्यंत त्या एस्टिमेटला सुधारत राहतो:

```
while True:
    print(x)
    y = (x + a/x) / 2
    if y == x:
        break
    x = y
```

बहुतांश `a`च्या व्हॅल्यूझ साठी हे चालून जाते, पण साधारणपणे `float` ची समानता तपासणे हे असुरक्षित आहे. फ्लोटिंग-पॉइंट व्हॅल्यू फक्त अंदाजे बरोबर असते: बहुतांश परिमेय संख्या (rational numbers), उदा., $1/3$ आणि अपरिमेय संख्या, उदा., $\sqrt{2}$ ह्या `float` ने अचूकपणे व्यक्त करता येत नाहीत.

आपण `x` आणि `y` हे तंतोतंतपणे सारखे आहेत का हे तपासण्याऐवजी, `abs` बिल्ट-इन फंक्शन वापरून त्यांच्या फरकाचे केवलमूल्य (absolute value) तपासणे जास्त सुरक्षित आहे:

```
    if abs(y-x) < epsilon:
        break
```

इथे `epsilon`ची `0.0000001` व्हॅल्यू आपल्याला खऱ्या उत्तराच्या किती जवळ जायचे आहे ते दर्शवते.

७.६ अल्गोरिदम (Algorithm)

न्यूटनची पद्धत ही **अल्गोरिदम** (**algorithm**)चे उदाहरण आहे: ती प्रॉब्लेम्सच्या कोणत्याही एका विशिष्ठ समूहासाठी (ह्याठिकाणी वर्गमूळ शोधणे ह्यासाठी) उत्तर शोधून काढण्याची एक यंत्रवत प्रक्रिया आहे. (अनुवादकाची टिप्पणी: यंत्रवत म्हणजे तंतोतंतपणे पार पाडणे, यंत्रांशी संबंधित नाही.)

अल्गोरिदम काय आहे हे समजून घेण्यासाठी आपण आधी अल्गोरिदम काय नाही ते पाहूया. लहानपणी तुम्ही एक अंकी संख्यांचा गुणाकार पाढे पाठ करून शिकलात. एकंदरित, तुम्ही १०० उत्तरे पाठ केलीत. ह्या ज्ञानाला अल्गोरिदमिक (algorithmic) नाही म्हणता येणार.

पण जर तुम्ही 'आळशी' होता, तर तुम्ही काही युक्त्या शिकला असाल. उदा., n आणि 9 चा गुणाकार शोधण्यासाठी तुम्ही $n-1$ ला पहिला अंक आणि $10-n$ ला दुसरा अंक असे लिहून उत्तर शोधू शकता (उदा., $8 \times 9 = 72$, पहिला अंक $8-1 = 7$ आणि दुसरा $10-8 = 2$). ही युक्ती कोणत्याही एक अंकी संख्येला 9 ने गुणण्यासाठी चालते. हा एक अल्गोरिदम आहे!

त्याचप्रमाणे, हातचा घेऊन बेरीज, वजाबाकी, किंवा भागाकार करण्याच्या पद्धती ह्या अल्गोरिदम्सच आहेत. अल्गोरिदमचे एक वैशिष्ट्य असे की त्याला पार पाडायला कोणत्याही हुशारीची गरज लागत नाही. त्या यंत्रवत करायच्या प्रक्रिया आहेत जिथे कृतीची प्रत्येक पायरी आधीच्या पायरीपासून एकदम सोपे नियम वापरून पार पाडता येते.

अल्गोरिदम्स एक्सेक्युट करणे कंटाळवाणे काम आहे, पण डिझाइन करणे खूपच मजेशीर, आव्हानात्मक, आणि कुतूहलजन्य असून त्यास संगणक विज्ञानात मध्यवर्ती स्थान आहे.

आपण काही गोष्टी विचार *न* करता सहजपणे करतो त्यांना अल्गोरिदमिकली (algorithmically) व्यक्त करणे अतिशय अवघड असते. नैसर्गिक भाषांना समजणे हे एक उत्तम उदाहरण आहे. आपल्या सर्वांना नैसर्गिक भाषा समजतात, पण आतापर्यंत कोणीही आपण हे *कसे* करतो हे एका अल्गोरिदमच्या स्वरुपात विशद करू शकलेले नाही.

७.७ डीबगिंग (Debugging)

तुम्ही जसजसे मोठे प्रोग्राम्स लिहित जाल, तसतसा तुम्ही डीबगिंगवर जास्ती वेळ खर्च कराल. जास्त कोड म्हणजे एरर होण्याची जास्त शक्यता आणि बग्सना लपण्यासाठी मोठी जागा.

डीबगिंगचा वेळ कमी करण्याचा एक मार्ग म्हणजे 'दुभाजनाने डीबगिंग' ('debugging by bisection'). उदा., जर तुमच्या प्रोग्राममध्ये १०० ओळी असतील आणि तुम्ही त्या एकेक करून तपासल्या तर तुम्हाला १०० आवर्तने करावी लागतील.

त्याऐवजी, प्रोग्रामचे दोन भागात विभाजन करा. प्रोग्रामच्या अंदाजे मध्यभागी एखादी व्हॅल्यू तपासून बघा. तिथे एक `print` स्टेटमेंट टाका (किंवा असा बदल करा ज्याचा परिणाम तुम्हाला तपासता येईल) आणि प्रोग्राम रन करा.

जर ही मध्यबिंदू चाचणी नापास झाली तर प्रोग्रामच्या पहिल्या अर्ध्या भागात नक्की काहीतरी चूक आहे. जर पास झाली तर चूक ही खालच्या भागात आहे.

जेव्हाही तुम्ही अशी चाचणी करता, तपासाव्या लागणाऱ्या ओळींची संख्या अर्धी होते. असे सहावेळा (म्हणजे १०० पेक्षा कितीतरी कमीवेळा) केल्यावर, तुमच्या ओळींची संख्या एक किंवा दोन होऊन जाईल (निदान तात्विकदृष्ट्या तरी).

पण खरे तर 'प्रोग्रामचा मध्यभाग' कोणता हे स्पष्ट नसते आणि तिथे तपास करणे शक्य नसते. अचूक मध्यभाग शोधणे देखील निरर्थक आहे. त्याऐवजी प्रोग्राममध्ये कुठे बग असण्याची जास्त संभाव्यता आहे आणि कुठे तपास करणे सोपे पडेल ह्याचा (तुमच्या अंदाजाने) विचार करा. नंतर अशा ठिकाणी चाचणी करा की बग त्याच्या आधी आणि नंतर असण्याची जवळजवळ सारखीच शक्यता असावी.

७.८ शब्दार्थ

रीअसाइनमेंट (reassignment): आधीच असलेल्या व्हेरिएबलला नवीन व्हॅल्यू असाइन करणे.

अपडेट (update): अशी असाइनमेंट जिथे व्हेरिएबलची नवीन व्हॅल्यू त्याच्या जुन्या व्हॅल्यूवर अवलंबून असते.

इनिशलाइझेशन (initialization): अशी असाइनमेंट जी एका व्हेरिएबलला सुरुवातीची (पहिली) व्हॅल्यू देते, जी नंतर अपडेटही केली जाऊ शकते. (इनिशलाइझेशन *न* केलेले व्हेरिएबल अपडेट नाही करू शकत.)

इन्क्रिमेंट (increment): असा अपडेट जो व्हेरिएबलची व्हॅल्यू (सहसा एकने) वाढवतो.

डिक्रिमेंट (decrement): असा अपडेट जो व्हेरिएबलची व्हॅल्यू (सहसा एकने) कमी करतो.

इटरेशन (iteration): काही स्टेटमेंट्सचे रिकर्शन किंवा लूप वापरून पुनःपुन्हा केलेले एक्सेक्युशन.

इन्फिनेट लूप (infinite loop): असा लूप ज्याला संपवणारी कंडिशन कधीच पूर्ण होत नाही.

अल्गोरिदम (algorithm): एका विशिष्ठ समूहातील प्रॉब्लेम्स सोडवण्यासाठी दिलेली कृती.

७.९ प्रश्नसंच (Exercises)

प्रश्न ७.१. *विभाग ७.५ मधला लूप कॉपी करा आणि त्याला* `mysqrt` *नावाच्या फंक्शनमध्ये एन्कॅप्सुलेट (encapsulate) करा; हे फंक्शन* `a` *परॅमीटर घेऊन,* `x`*ची योग्य व्हॅल्यू निवडून,* `a`*च्या वर्गमुळाचा एक एस्टिमेट रिटर्न करेल असे लिहा.*

त्याला टेस्ट करण्यासाठी `test_square_root` *नावाचे फंक्शन लिहा जे असा तक्ता दाखवेल:*

```
a   mysqrt(a)     math.sqrt(a)  diff
-   ---------     ------------  ----
1.0 1.0           1.0           0.0
2.0 1.41421356237 1.41421356237 2.22044604925e-16
3.0 1.73205080757 1.73205080757 0.0
4.0 2.0           2.0           0.0
5.0 2.2360679775  2.2360679775  0.0
6.0 2.44948974278 2.44948974278 0.0
7.0 2.64575131106 2.64575131106 0.0
8.0 2.82842712475 2.82842712475 4.4408920985e-16
9.0 3.0           3.0           0.0
```

पहिला रकाना a *ही संख्या दाखवतो; दुसरा* a*चे* `mysqrt` *ने काढलेले वर्गमूळ दाखवतो; तिसरा* `math.sqrt` *ने काढलेले वर्गमूळ; आणि चौथा दोन्ही एस्टिमेटमधील फरकाचे केवलमूल्य (absolute value).*

प्रश्न ७.२. *एक स्ट्रिंग अर्ग्युमेंट घेऊन* `eval` *हे बिल्ट-इन फंक्शन त्या स्ट्रिंगला पायथॉन इंटरप्रिटर वापरून इव्हॅल्युएट (evaluate) करते. उदा.:*

```
>>> eval('1 + 2 * 3')
7
>>> import math
>>> eval('math.sqrt(5)')
2.2360679774997898
>>> eval('type(math.pi)')
<class 'float'>
```

एक `eval_loop` *नावाचे फंक्शन लिहा जे पुनःपुन्हा प्रॉम्प्ट करून युझरकडून इनपुट घेते आणि ते* `eval` *वापरून इव्हॅल्युएट करते, आणि उत्तर दाखवते.*

युझर जोपर्यंत `'done'` *असे लिहित नाही तोपर्यंत हे चालू ठेवा, आणि त्यानंतर शेवटच्या एक्स्प्रेशनची व्हॅल्यू रिटर्न करा.*

प्रश्न ७.३. *गणितज्ञ श्रीनिवास रामानुजन ह्यांनी एका इन्फिनेट सिरीझचा (infinite series) शोध लावला जी वापरून* $1/\pi$ *ची अंदाजी किंमत काढता येते:*

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}.$$

`estimate_pi` *नावाचे फंक्शन लिहा जे हे सूत्र वापरून* π*चा एस्टिमेट शोधून रिटर्न करते. त्यात* `while` *लूप वापरून मोठ्या बेरजेतील टर्म्स (terms, पदे) तोपर्यंत शोधा जोपर्यंत शेवटची टर्म* `1e-15` *पेक्षा लहान असत नाही (पायथॉनमध्ये* `1e-15` *म्हणजे* 10^{-15}*). उत्तर तपासण्यासाठी तुम्ही* `math.pi` *शी तुलना करू शकता.*

उत्तर: http://thinkpython2.com/code/pi.py.

प्रकरण ८

स्ट्रिंग (String)

स्ट्रिंग ही इंटिजर, फ्लोट, किंवा बूलियन सारखी नसते. स्ट्रिंग एक **सीक्वेन्स** (**sequence**, क्रमिका, यादी) असते, म्हणजेच क्रमाने दिलेल्या अक्षरांचा संच. ह्या प्रकरणात तुम्ही स्ट्रिंगची कॅरेक्टर्स कशी वापरायची ते आणि स्ट्रिंगच्या काही मेथड्स (methods) शिकणार आहात.

८.१ स्ट्रिंग ही एक क्रमिका आहे (A string is a sequence)

स्ट्रिंग म्हणजे कॅरेक्टर्सचा सीक्वेन्स. तुम्ही ती कॅरेक्टर्स एकेक करून ब्रॅकेट (bracket, कंस) ऑपरेटरद्वारे वापरू शकता:

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

दुसरे स्टेटमेंट `fruit` मधून कॅरेक्टर क्रमांक १ घेऊन ते `letter` ला असाइन करते.

ब्रॅकेट्समधल्या एक्स्प्रेशनला **इंडेक्स** (**index**, निर्देशांक) म्हणतात. सीक्वेन्समधील कोणत्या क्रमांकाचे कॅरेक्टर तुम्ही दर्शवत आहात हे इंडेक्स सांगते (म्हणून निर्देशांक हे नाव).

पण तुम्हाला जे अपेक्षित आहे ते नाही मिळणार:

```
>>> letter
'a'
```

(गणितज्ञांसह) बहुतांश लोकांसाठी `'banana'` चे पहिले अक्षर `b` आहे, `a` नाही. पण संगणक वैज्ञानिकांसाठी, इंडेक्स म्हणजे स्ट्रिंगच्या सुरुवातीपासून असलेले ऑफसेट (offset, म्हणजे अंतर). आणि पहिल्या अक्षराचे ऑफसेट शून्य आहे.

```
>>> letter = fruit[0]
>>> letter
'b'
```

म्हणजे `b` हे `'banana'` चे ० वे अक्षर (‘शून्य-वे’), `a` हे १ वे अक्षर (‘एक-वे’), आणि `n` हे २ वे अक्षर (‘दोन-वे’). (मूळ इंग्रजी वाक्य: So `b` is the 0th letter (“zero-eth”) of `'banana'`, `a` is the 1th letter (“one-eth”), and `n` is the 2th letter (“two-eth”).)

तुम्ही व्हेरिएबल्स आणि ऑपरेटर्स असलेले एक्स्प्रेशन इंडेक्स म्हणून वापरू शकता:

```
>>> i = 1
>>> fruit[i]
'a'
>>> fruit[i+1]
'n'
```

पण इंडेक्सची व्हॅल्यू ही इंटिजर असली पाहिजे. नाहीतर तुम्हाला हा एरर मिळेल:

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

८.२ `len`

`len` हे बिल्ट-इन फंक्शन स्ट्रिंगमध्ये किती कॅरेक्टर्स आहेत हे सांगते:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

स्ट्रिंगचे शेवटचे कॅरेक्टर मिळवण्यासाठी तुम्हाला असे करावेसे वाटेल:

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

`IndexError` मिळण्याचे कारण हे की `'banana'` मध्ये 6 इंडेक्स असणारे कॅरेक्टरच नाहीये. आपण शून्यापासून मोजायला सुरुवात केल्याने सहा अक्षरांचे क्रमांक हे 0 ते 5 आहेत. शेवटचे कॅरेक्टर मिळवण्यासाठी, तुम्हाला `length` मधून 1 वजा करावा लागेल:

```
>>> last = fruit[length-1]
>>> last
'a'
```

तुम्ही ऋण इंडेक्ससुद्धा वापरू शकता, जी स्ट्रिंगच्या शेवटापासून मोजते; `fruit[-1]` हे एक्स्प्रेशन शेवटचे कॅरेक्टर देते, `fruit[-2]` शेवटून दुसरे, इत्यादी.

८.३ `for` लूप-ने मागोवा (Traversal with a `for` loop)

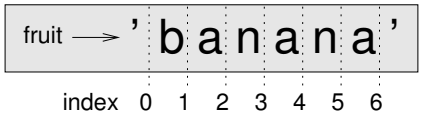
खूपदा, तुम्हाला स्ट्रिंगची कॅरेक्टर्स एकेक करून वापरावी लागतात. सहसा त्याची सुरुवात पहिल्यापासून होते: आळीपाळीने प्रत्येक कॅरेक्टर निवडा, त्याचे काहीतरी करा, आणि शेवटपर्यंत असेच चालू द्या. प्रोग्रामिंगच्या ह्या पॅटर्नला **ट्रव्हर्सल** (**traversal**, मार्गक्रमण) असे म्हणतात. ट्रव्हर्सल लिहिण्याची एक पद्धत म्हणजे `while` लूप:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

वरील लूप स्ट्रिंग ट्रव्हर्स करतो आणि प्रत्येक अक्षर स्वतंत्र ओळीवर दाखवतो. लूप कंडिशन `index < len(fruit)` आहे, म्हणून `index` ची व्हॅल्यू जेव्हा स्ट्रिंगच्या लांबीइतकी होते तेव्हा कंडिशन असत्य होते आणि लूप रन होत नाही. सर्वांत शेवटी वापरलेल्या कॅरेक्टरची इंडेक्स `len(fruit)-1` आहे, जी स्ट्रिंगचे शेवटचे कॅरेक्टर दर्शवते.

सराव प्रश्न: एक स्ट्रिंग अर्ग्युमेंट घेणारे आणि त्यातील अक्षरे उलट्या क्रमाने स्वतंत्र ओळींवर दाखवणारे फंक्शन लिहा.

ट्रव्हर्सल लिहिण्याची अजून एक पद्धत म्हणजे `for` लूप:



आकृती ८.१: स्लाइस इंडेक्स (Slice indices).

```
for letter in fruit:
    print(letter)
```

लूपमधून प्रत्येकवेळी जाताना स्ट्रिंगमधील पुढील कॅरेक्टर `letter` व्हेरिएबलला असाइन होते. जोपर्यंत कॅरेक्टर्स आहेत तोपर्यंत लूप चालू राहतो.

खालील उदाहरण कन्कॅटनेशन (concatenation) आणि `for` लूप वापरून एक abecedarian series (म्हणजे वर्णक्रमानुसार) कशी बनवायची हे दाखवते. Robert McCloskey च्या *Make Way for Ducklings* (बदकाच्या पिल्लांची वाट सोडा) नावाच्या पुस्तकात बदकाच्या पिल्लांची नावे Jack, Kack, Lack, Mack, Nack, Ouack, Pack, आणि Quack अशी आहेत. खालील लूप ही नावे क्रमाने दाखवतो:

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'

for letter in prefixes:
    print(letter + suffix)
```

आणि आउटपुट असे आहे:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

पण हे चूक आहे, कारण 'Ouack' आणि 'Quack' ह्यांचे स्पेलिंग चुकले आहे. सराव म्हणून, ही चूक सुधारण्यासाठी प्रोग्राममध्ये योग्य बदल करा.

८.४ स्ट्रिंगचे काप (String slices)

स्ट्रिंगच्या एका तुकड्याला **स्लाइस** (**slice**, काप) म्हणतात. स्लाइस निवडणे हे कॅरेक्टर निवडण्यासारखेच आहे:

```
>>> s = 'Monty Python'
>>> s[0:5]
'Monty'
>>> s[6:12]
'Python'
```

`[n:m]` हा ऑपरेटर स्ट्रिंगच्या 'n-व्या' कॅरेक्टरपासून ते 'm-व्या' कॅरेक्टरपर्यंतचा स्ट्रिंगचा भाग रिटर्न करतो—पहिल्या कॅरेक्टरसह पण शेवटचे कॅरेक्टर सोडून. हे थोडे विचित्र वाटेल, पण आकृती ८.१ मध्ये दाखवल्याप्रमाणे इंडेक्स कॅरेक्टर्सच्या *मध्ये* दर्शवते अशी कल्पना केल्याने हे समजणे सोपे जाईल.

जर तुम्ही पहिली इंडेक्स (अपूर्णविरामाच्या, म्हणजे ':' च्या आधीची) दिली नाहीत, तर स्लाइस स्ट्रिंगच्या सुरुवातीपासून सुरू होतो. दुसरी इंडेक्स नाही दिलीत तर स्लाइस स्ट्रिंगच्या शेवटपर्यंत जातो:

```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

जर पहिली इंडेक्स दुसरीपेक्षा लहान नसेल तर उत्तर हे **एम्प्टी स्ट्रिंग** (**empty string**, रिक्त स्ट्रिंग) असते, जी दोन अवतरण चिन्हांनी दर्शवली जाते:

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

एम्प्टी स्ट्रिंगमध्ये कॅरेक्टर्स नसतात आणि तिची लांबी (length) शून्य असते, पण हे सोडून ती अन्य स्ट्रिंगसारखीच असते.

हे उदाहरण चालू ठेवूया; `fruit[:]` चा काय अर्थ होतो? करून पहा.

८.५ स्ट्रिंग इम्युटबल असते (A string is immutable)

तुम्हाला असे वाटेल की, `[]` ऑपरेटर असाइनमेंटच्या डाव्या बाजूला वापरून आपण स्ट्रिंगचे एक कॅरेक्टर बदलू शकतो. उदा.:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

ह्याठिकाणी 'ऑब्जेक्ट' म्हणजे स्ट्रिंग आणि आणि 'आयटम' ('item') म्हणजे ज्या कॅरेक्टरला तुम्ही असाइन करायचा प्रयत्न केला ते. आतापुरते असे समजा की ऑब्जेक्ट आणि व्हॅल्यू म्हणजे एकच गोष्ट पण ही व्याख्या आपण नंतर (विभाग १०.१० मध्ये) दुरूस्त करणार आहोत.

वरील एररचे कारण म्हणजे स्ट्रिंग **इम्युटबल** (**immutable**) म्हणजे बदलता *न* येणारी) असते. पण तुम्ही मूळ स्ट्रिंग वापरून एक वेगळी स्ट्रिंग बनवू शकता:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> new_greeting
'Jello, world!'
```

ह्या उदाहरणात, `'J'` ह्या नवीन अक्षराला `greeting` स्ट्रिंगचा `greeting[1:]` हा स्लाइस जोडून एक नवीन स्ट्रिंग बनवली आहे. मूळ स्ट्रिंगवर, म्हणजे `greeting` दर्शवते त्या स्ट्रिंगवर ह्याचा काहीच परिणाम होत नाही.

८.६ शोध (Searching)

खालील फंक्शन काय करते?

```
def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            return index
        index = index + 1
    return -1
```

एकप्रकारे, `find` फंक्शन `[]` ऑपरेटरच्या उलट आहे. एक इंडेक्स घेऊन तिथले कॅरेक्टर उचलण्याऐवजी, ते फंक्शन एक कॅरेक्टर घेऊन ते कॅरेक्टर ज्या इंडेक्सवर मिळेल ती इंडेक्स रिटर्न करते. आणि जर कॅरेक्टर नाही मिळाले तर `-1` रिटर्न करते.

एका लूपमध्ये `return` स्टेटमेंट असणारे पहिलेच उदाहरण आपण ह्याठिकाणी बघत आहोत; `word[index] == letter` ही अट पूर्ण झाल्याझाल्या ताबडतोब लूपमधून बाहेर पडून फंक्शन रिटर्न होते.

जर कॅरेक्टर स्ट्रिंगमध्ये नसेल तर प्रोग्राम लूपमधून साधारणपणे बाहेर पडतो आणि फंक्शन `-1` व्हॅल्यू रिटर्न करते.

ह्या प्रोग्रामिंग पॅटर्नला—म्हणजेच, एक सीक्वेन्स ट्रव्हर्स करून जे शोधतोय ते मिळाल्यावर रिटर्न करण्याला—**सर्च** (**search**, शोध) म्हणतात.

सराव प्रश्न: `find` फंक्शनमध्ये बदल करून त्याला तिसरा पॅरमीटर घ्यायला लावा जो `word` मधील ती इंडेक्स दर्शवतो जिथून तुम्हाला सर्च-ला सुरुवात करायची आहे.

८.७ लूपिंग आणि काउंटिंग (Looping and counting)

खालील प्रोग्राम स्ट्रिंगमध्ये `a` किती वेळा आहे ते मोजतो:

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

हा प्रोग्राम **काउंटर** (**counter**) नावाचा प्रोग्रामिंग पॅटर्न दाखवतो. इथे `count` व्हेरिएबल 0 व्हॅल्यूने इनिशलाइझ केले आहे आणि आपल्याला जेव्हाही `a` अक्षर मिळते तेव्हा `count` व्हेरिएबल १ ने इन्क्रिमेंट केले जाते. जेव्हा लूप संपतो तेव्हा `count` मध्ये उत्तर असते—ते म्हणजे `a` अक्षराची एकूण संख्या.

सराव प्रश्न: ह्या कोड-ला `count` नावाच्या फंक्शनमध्ये एन्कॅप्सुलेट (encapsulate) करा, आणि त्याला जनरलाइझ करा जेणेकरून ते एक स्ट्रिंग आणि एक कॅरेक्टर अर्ग्युमेंट्स म्हणून घेईल.

नंतर ह्या फंक्शनमध्ये असा बदल करा: ह्या फंक्शनमध्ये, स्ट्रिंग ट्रव्हर्स करण्याऐवजी मागच्या विभागातील (सराव-प्रश्नातील) तीन पॅरमीटर्स घेणारे `find` फंक्शन वापरा.

८.८ स्ट्रिंग मेथड्स (String methods)

स्ट्रिंगवर विविध प्रकारच्या उपयुक्त क्रिया करण्यासाठी मेथड्स आहेत. मेथड ही फंक्शनसारखीच असते—ती अर्ग्युमेंट्स घेते आणि व्हॅल्यू रिटर्न करते—पण तिचा सिंटॅक्स वेगळा असतो. उदा., `upper` मेथड स्ट्रिंगच्या सर्व कॅरेक्टर्सना कॅपिटल (capital, uppercase) बनवून, एका नवीन स्ट्रिंगमध्ये टाकून, ती नवीन स्ट्रिंग रिटर्न करते.

पण `upper(word)` असा फंक्शन सिंटॅक्स *न* वापरता `word.upper()` असा मेथड सिंटॅक्स वापरतात.

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> new_word
'BANANA'
```

ह्या प्रकारच्या डॉट नोटेशनमध्ये (dot notation) मेथडचे नाव (इथे, `upper`) आणि ज्या स्ट्रिंगला मेथड लावायची आहे तिचे नाव (इथे, `word`) वापरतात. रिकामे कंस हे दर्शवतात की ही मेथड कोणतीच अर्ग्युमेंट्स घेत नाही.

मेथड कॉल-ला **इन्व्होकेशन** (**invocation**, इथे 'बोलावणे' असा अर्थ) म्हणतात; ह्या उदाहरणात आपण असे म्हणू शकतो की `word` व्हेरिएबल जी स्ट्रिंग दर्शवते त्यावर आपण `upper` मेथड इन्व्होक (invoke) करतोय.

खरे तर स्ट्रिंगमध्ये `find` नावाची एक मेथड आहे जी आपण नुकत्याच लिहिलेल्या `find` फंक्शनसारखीच आहे:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> index
1
```

ह्या उदाहरणात आपण `word` वर `find` इन्व्होक करतो आणि जे अक्षर शोधायचे आहे ते अर्ग्युमेंट म्हणून पाठवतो.

वास्तविक पाहता `find` मेथडची व्यापकता आपल्या फंक्शनहून जास्ती आहे; ती फक्त कॅरेक्टर्सच नाही तर सब्स्ट्रिंग (substring, उप-स्ट्रिंग, म्हणजे स्ट्रिंगचा सलग भाग) पण शोधू शकते:

```
>>> word.find('na')
2
```

साधारणपणे, `find` मेथड स्ट्रिंगच्या सुरुवातीपासून शोधायला सुरू करते; पण ती कोणत्या इंडेक्सपासून शोधायला सुरू करायचे आहे हे सांगणारे दुसरे अर्ग्युमेंटही घेऊ शकते:

```
>>> word.find('na', 3)
4
```

हे **ऑप्शनल अर्ग्युमेंट** (**optional argument**, पर्यायी अर्ग्युमेंट) चे उदाहरण आहे; `find` मेथड तिसरे अर्ग्युमेंटही घेऊ शकते जे कोणत्या इंडेक्सआधी शोधायला थांबवायचे आहे हे दर्शवते:

```
>>> name = 'bob'
>>> name.find('b', 1, 2)
-1
```

हा सर्च अपयशी ठरतो कारण `'b'` अक्षर `1` ते `2` ह्या इंडेक्सच्या *पट्ट्यात* नाहीये; कारण ह्या *पट्ट्यात* `2` चा समावेश नाहीये. शोध घेताना दुसऱ्या इंडेक्सचा समावेश *न* करता अगदी आधीपर्यंत बघण्यामुळे `find` मेथड आणि स्लाइस ऑपरेटर एकमेकांशी सुसंगत ठरतात.

८.९ `in` ऑपरेटर (`in` operator)

इंग्रजीमधला `in` शब्द बूलियन ऑपरेटर आहे; हा ऑपरेटर दोन स्ट्रिंग्स घेतो आणि जर पहिली दुसरीची सब्स्ट्रिंग (substring, उप-स्ट्रिंग) असेल तर `True` रिटर्न करतो:

```
>>> 'a' in 'banana'
True
>>> 'nan' in 'banana'
True
>>> 'seed' in 'banana'
False
```

उदा., खालील फंक्शन `word1` मधील ती सर्व अक्षरे दाखवतो जी `word2` मध्ये सुद्धा आहेत:

```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print(letter)
```

जर व्हेरिएबलची नावे व्यवस्थितपणे निवडली तर पायथॉन कधीकधी इंग्रजीसारखेच वाटते. तुम्ही ह्या लूप-ला इंग्रजीमध्ये असे वाचू शकता: 'for (each) letter in (the first) word, if (the) letter (appears) in (the second) word, print (the) letter.'

जर तुम्ही सफरचंद (apples) आणि संत्र्यांची (oranges) तुलना केली तर असे होते (अनुवादकाची टिप्पणी: इंग्रजीमध्ये 'comparing apples and oranges' ह्या वाक्प्रचाराचा असा अर्थ होतो की दोन अतिभिन्न गोष्टींची अनुचित तुलना करणे.):

```
>>> in_both('apples', 'oranges')
a
e
s
```

८.१० स्ट्रिंग तुलना (String comparison)

रिलेश्नल ऑपरेटर्स (relational operators) स्ट्रिंगवरही चालतात. दोन स्ट्रिंग्स सारख्याच आहेत का हे तपासण्यासाठी:

```
if word == 'banana':
    print('All right, bananas.')
```

इतर रिलेश्नल ऑपरेटर्सचा शब्दांची वर्णक्रमानुसार (alphabetical) मांडणी करायला उपयोग होतो:

```
if word < 'banana':
    print('Your word, ' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word, ' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```

पायथॉनमध्ये कॅपिटल आणि स्मॉल अक्षरे वेगळ्याप्रकारे हाताळली जातात. सर्व कॅपिटल अक्षरे स्मॉल अक्षरांच्या आधी येतात, म्हणून:

```
Your word, Pineapple, comes before banana.
```

हा मुद्दा हाताळण्यासाठी आपण सर्व स्ट्रिंग्स स्मॉलमध्ये घेऊन मग तुलना करू शकतो.

अनुवादकाची टिप्पणी: पायथॉन लँग्वेज ही Guido van Rossum ह्यांनी बनवली आहे. त्यांनी पायथॉन हे नाव ब्रिटिश विनोदी गट 'Monty Python' वर आधारून ठेवले आहे. तुम्ही जर पायथॉनशी संबंधित साहित्य बघितले, जसे हेच पुस्तक, तर त्यात तुम्हाला 'Monty Python' मधील काही विनोद आढळून येतील. असाच एक विनोद मूळ लेखकाने इथे केला होता, ज्याचे मी साहजिकच भाषांतर नाही करू शकणार, कारणकी त्याचे 'lost-in-translation' होऊन जाईल. ती ओळ आहे, 'Keep that in mind in case you have to defend yourself against a man armed with a Pineapple.'

८.११ डीबगिंग (Debugging)

जेव्हा तुम्ही सीक्वेन्समधल्या व्हॅल्यूझ ट्रव्हर्स (traverse) करण्यासाठी इंडेक्स वापरता, तेव्हा त्याची सुरुवात आणि शेवट योग्यपणे निवडण्यात तुमचा घोळ होऊ शकतो. खालील फंक्शनमध्ये आपल्याला दोन शब्दांची तुलना करून जर पहिला दुसऱ्याचा उलटा असेल तर `True` रिटर्न करायचे आहे पण त्यात दोन एरर्स (errors) आहेत:

```
def is_reverse(word1, word2):
    if len(word1) != len(word2):
        return False

    i = 0
    j = len(word2)
```

```
    while j > 0:
        if word1[i] != word2[j]:
            return False
        i = i+1
        j = j-1

    return True
```

पहिले `if` स्टेटमेंट हे तपासते की दोन्ही शब्दांची लांबी सारखीच आहे की नाही. जर नसेल तर आपण ताबडतोब `False` रिटर्न करू शकतो. नाहीतर उर्वरीत फंक्शनमध्ये आपण हे गृहीत धरू शकतो की शब्दांची लांबी सारखीच आहे. हे विभाग ६.८ मध्ये पाहिलेल्या गार्डियन पॅटर्नचे उदाहरण आहे.

आता, `i` आणि `j` ह्या दोन इंडसीस (indices, indexचे अनेकवचन) आहेत: `i` ही `word1`ला सरळ ट्रव्हर्स करते आणि `j` ही `word2`ला उलटे ट्रव्हर्स करते. जर आपल्याला दोन वेगळी अक्षरे मिळाली तर आपण ताबडतोब `False` रिटर्न करू शकतो. जर आपण पूर्ण लूपमधून गेलो आणि सर्व अक्षरे सारखी असली तर आपण `True` रिटर्न करू शकतो.

जर आपण हे फंक्शन 'pots' आणि 'stop' ह्या शब्दांनी टेस्ट केले तर उत्तर `True` असे यायला पाहिजे, पण आपल्याला IndexError मिळतो:

```
>>> is_reverse('pots', 'stop')
...
  File "reverse.py", line 15, in is_reverse
    if word1[i] != word2[j]:
IndexError: string index out of range
```

ह्याप्रकारचा एरर डीबग करण्यासाठी, सर्वांत आधी ज्या ओळीवर एरर येतोय त्याच्या अगदी एक ओळ आधी इंडसीस (indices) च्या व्हॅल्यूझ प्रिंट करणे फायदेशीर ठरते.

```
    while j > 0:
        print(i, j)        # print here

        if word1[i] != word2[j]:
            return False
        i = i+1
        j = j-1
```

आता जर आपण प्रोग्राम परत रन केला, तर आपल्याला अधिक माहिती मिळते:

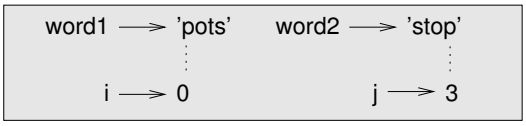
```
>>> is_reverse('pots', 'stop')
0 4
...
IndexError: string index out of range
```

लूप-च्या पहिल्याच वेळेला `j` ची व्हॅल्यू 4 आहे, जी `'pots'` मधल्या इंडसीसच्या *पट्ट्याच्या* (म्हणजे $\{0,1,2,3\}$ ह्या वैध इंडसीसच्या संचाच्या) बाहेर आहे. शेवटच्या कॅरेक्टरची इंडेक्स 3 आहे, म्हणजेच `j` ची सुरुवातीची व्हॅल्यू `len(word2)-1` असायला हवी.

जर आपण ती चूक दुरूस्त करून प्रोग्राम परत रन केला, तर आपल्याला मिळते:

```
>>> is_reverse('pots', 'stop')
0 3
1 2
2 1
True
```

ह्यावेळी आपल्याला बरोबर उत्तर मिळाले, पण असे दिसतेय की लूप फक्त तीन वेळा चालला; हे थोडे संशयास्पद दिसतेय. काय होते आहे हे जाणून घेण्यासाठी स्टेट डायग्राम (state diagram) काढूया. पहिल्या इटरेशन (iteration) दरम्यानची `is_reverse` ची फ्रेम आकृती ८.२ मध्ये दाखवली आहे.



आकृती ८.२: स्टेट डायग्राम (State diagram).

सादरीकरणाच्या सोयीसाठी ह्या आकृतीत थोडे बदल केलेत: व्हेरिएबल्स थोडी पद्धतशीरपणे मांडली आहेत; `i` आणि `j` अनुक्रमे `word1` आणि `word2` मधली कोणती कॅरेक्टर्स दर्शवत आहेत हे तुटक रेषांनी दाखवले आहे.

ह्या आकृतीपासून सुरुवात करून प्रोग्राम कागदावर चालवा; प्रत्येक इटरेशनमध्ये `i` आणि `j` च्या व्हॅल्यूझ योग्यपणे बदला. आणि ह्या फंक्शनमधला दुसरा बग (bug) शोधून काढा.

८.१२ शब्दार्थ

ऑब्जेक्ट (object): एखादे व्हेरिएबल दर्शवते ती गोष्ट. सध्या तुम्ही 'ऑब्जेक्ट' आणि 'व्हॅल्यू' म्हणजे एकच गोष्ट असे मानून चला.

सीक्वेन्स (sequence, क्रमिका): व्हॅल्यूझची क्रमाने दिलेली यादी ज्यात प्रत्येक व्हॅल्यू एका इंटिजर इंडेक्स ने दर्शवली (निर्देशित केली) जाते.

आयटम (item): सीक्वेन्समधली व्हॅल्यू.

इंडेक्स (index, निर्देशांक): सीक्वेन्समधील एक आयटम निवडण्यासाठी वापरलेली इंटिजर व्हॅल्यू; सीक्वेन्समधील आयटमचे उदाहरण म्हणजे स्ट्रिंगमधील एक कॅरेक्टर. पायथॉनमध्ये इंडसीस (indices) शून्यापासून सुरू होतात.

स्लाइस (slice): इंडसीस (indices) देऊन दर्शवलेला स्ट्रिंगचा भाग.

एम्प्टी स्ट्रिंग (empty string, रिकामी स्ट्रिंग): एकही कॅरेक्टर नसलेली, शून्य लांबीची स्ट्रिंग; ही दोन अवतरण चिन्हांनी दर्शवली जाते.

इम्युटबल (immutable, बदलता न येणे): आयटम *न* बदलता येऊ शकण्याचा सीक्वेन्सचा गुणधर्म.

ट्रव्हर्स (traverse, मार्गक्रमण करणे): सीक्वेन्समधील आयटम्सवरून इटरेट (iterate) करून प्रत्येकावर सारखी क्रिया करणे.

सर्च (search): ट्रव्हर्सलचा एक पॅटर्न ज्यात शोधतोय ते मिळाल्यावर आपण लगेच थांबतो.

काउंटर (counter): काहीतरी मोजण्यासाठी वापरलेले व्हेरिएबल; सहसा शून्यने इनिशलाइझ करतात आणि नंतर इन्क्रिमेंट करतात.

इन्व्होकेशन (invocation): मेथड कॉल करणारे स्टेटमेंट.

ऑप्शनल अर्ग्युमेंट (optional argument, पर्यायी अर्ग्युमेंट): फंक्शन किंवा मेथडचे असे अर्ग्युमेंट जे देणे गरजेचे नाहीये.

८.१३ प्रश्नसंच (Exercises)

प्रश्न ८.१. *पुढील लिंकवर स्ट्रिंग मेथड्सचे डॉक्युमेंटेशन (documentation) वाचा: http://docs.python.org/3/library/stdtypes.html#string-methods.*

त्या मेथड्स वापरून काही प्रयोग करा म्हणजे तुम्हाला त्या नीट समजतील; `strip` *आणि* `replace` *ह्या विशेषतः उपयोगी आहेत.*

डॉक्युमेंटेशनमध्ये समजायला थोडा अवघड पडू शकेल असा सिंटॅक्स वापरला आहे. उदा., `find(sub[, start[, end]])` *मध्ये ब्रॅकेट्स (brackets, '[' आणि ']') ऑप्शनल अर्ग्युमेंट (optional argument) दर्शवतात. म्हणजे* `sub` *अनिवार्य आहे, पण* `start` *पर्यायी, आणि जर तुम्ही* `start` *दिला, तर* `end` *पर्यायी आहे.*

प्रश्न ८.२. *तिथे* `count` *नावाची मेथड आहे जी आपण विभाग ८.७ मध्ये बघितलेल्या फंक्शनसारखी आहे. ह्या मेथडचे डॉक्युमेंटेशन वाचा; तिला इन्व्होक (invoke) करून* `'banana'` *मध्ये किती* `a` *आहेत हे शोधा.*

प्रश्न ८.३. *स्ट्रिंग स्लाइस ऑपरेटर 'स्टेप साइझ' ('step size') दर्शवणारी तिसरी इंडेक्स घेऊ शकते; म्हणजे पुढील कॅरेक्टर घ्यायच्या आधी किती कॅरेक्टर्स सोडायची. स्टेप साइझ* 2 *असेल तर एक सोडून एक कॅरेक्टर,* 3 *असेल तर प्रत्येक तिसरे कॅरेक्टर, इत्यादी. उदा.,*

```
>>> fruit = 'banana'
>>> fruit[0:5:2]
'bnn'
```

जर स्टेप साइझ -1 *दिली तर स्ट्रिंग उलटी ट्रव्हर्स (traverse) केली जाते, म्हणजेच* `[::-1]` *उलट्या नवीन स्ट्रिंगची निर्मिती करते. हे वापरून प्रश्न ६.३ मधील* `is_palindrome` *फंक्शन एका ओळीत लिहा.*

प्रश्न ८.४. *खालील सर्व फंक्शन्सचा उद्देश दिलेल्या स्ट्रिंगमध्ये स्मॉल (lowercase) अक्षरे आहेत का हे तपासणे हा आहे, पण त्यांतली काही फंक्शन्स चुकीची आहेत. (परॅमीटर स्ट्रिंग आहे हे गृहीत धरून) प्रत्येक फंक्शन काय करते त्याचे वर्णन करा.*

```
def any_lowercase1(s):
    for c in s:
        if c.islower():
            return True
        else:
            return False

def any_lowercase2(s):
    for c in s:
        if 'c'.islower():
            return 'True'
        else:
            return 'False'

def any_lowercase3(s):
    for c in s:
        flag = c.islower()
    return flag

def any_lowercase4(s):
    flag = False
    for c in s:
        flag = flag or c.islower()
    return flag

def any_lowercase5(s):
    for c in s:
```

```
        if not c.islower():
            return False
    return True
```

प्रश्न ८.५. *सीझर सांकेतिक लिपी (Caesar cypher) ही मजकूर सुरक्षित ठेवण्यासाठी बनवलेली एक अप्रभावी लिपी आहे. ती वापरून मजकूर सांकेतिक रुपात बदलण्यासाठी प्रत्येक इंग्रजी अक्षर हे एका विशिष्ठ वेळा 'फिरवले' जाते. म्हणजेच, त्याला अल्फाबेट (alphabet) मध्ये पुढे सरकवले जाते आणि गरज पडली तर परत A पासून सुरुवात केली जाते. उदा., 'A' ला 3 वेळा फिरवल्यावर 'D' मिळतो, 'Z' ला 1 वेळा फिरवल्यावर 'A' मिळतो.*

एक पूर्ण शब्द फिरवण्यासाठी प्रत्येक अक्षर एकसारख्याच वेळा फिरवा. उदा., 'cheer' ला 7 वेळा फिरवले तर 'jolly' मिळते आणि 'melon' ला −10 वेळा फिरवले तर 'cubed' मिळते. एक गंमतीशीर गोष्ट: 2001: A Space Odyssey *ह्या चित्रपटात जहाजाच्या काँप्युटरचे नाव HAL आहे, जे IBM ला −1 वेळा फिरवल्यावर मिळते.*

एक स्ट्रिंग आणि एक इंटिजर अर्ग्युमेंट्स म्हणून घेणारे, आणि मूळ स्ट्रिंग दिलेल्या इंटिजर इतक्या वेळा फिरवल्यावर मिळणारी स्ट्रिंग रिटर्न करणारे `rotate_word` *नावाचे फंक्शन लिहा.*

ह्याठिकाणी, तुम्ही `ord` *हे बिल्ट-इन फंक्शन वापरू शकता; हे फंक्शन कॅरेक्टरचे कोड-संख्येत रुपांतर करते; आणि* `chr` *फंक्शन कोड-संख्येचे कॅरेक्टरमध्ये रुपांतर करते. ह्या फंक्शन्समध्ये अल्फाबेट (alphabet) मधल्या अक्षरांना संलग्न कोड-संख्या ह्या अल्फाबेटिकल क्रमानेच नियुक्त केलेल्या आहेत (म्हणजे, a, b, c, ... क्रमाने), आणि नियुक्त केलेल्या कोड-संख्या सलग आहेत. उदा.:*

```
>>> ord('c') - ord('a')
2
```

इथे `'c'` *हे अल्फाबेट चे '२-वे' अक्षर आहे. पण नोंद घ्या: कॅपिटल अक्षरांच्या कोड-संख्या वेगळ्या आहेत.*

इंटरनेट नवीन होते तेव्हा (म्हणजे काही दशकांआधी) आक्षेपार्ह विनोद[१] *ROT13 सायफर (cypher) वापरून सांकेतिक लिपीत रुपांतरित करून इंटरनेटवर वितरित केले जाई. ROT13 म्हणजेच सीझर सायफर वापरून शब्द १३ वेळा फिरवणे. असले विनोद तुम्हाला खटकत नसतील तर तुम्ही गंमत म्हणून इंटरनेटवर असले विनोद शोधून डीकोड (decode) करू शकता.*

उत्तर: http://thinkpython2.com/code/rotate.py.

[१] म्हणजेच politically incorrect.

प्रकरण ९

केस स्टडी: शब्दांची कोडी (Case study: word play)

ह्या प्रकरणात आपण दुसरी केस स्टडी पाहणार आहोत; ह्या स्टडीत आपण काही वैशिष्ट्यपूर्ण शब्द शोधून शब्दांची काही कोडी सोडवणार आहोत. उदा., आपण इंग्रजीमधील सर्वांत मोठी पॅलिंड्रोम (palindrome) शोधणार आहोत, असे शब्द शोधणार आहोत ज्यांची अक्षरे अल्फाबेटिकल (वर्णक्रमानुसार) आहेत. आणि आपण अजून एक प्रोग्राम डेव्हेलपमेंट प्लान बघणार आहोत: तो म्हणजे *आधी सोडवलेल्या प्रॉब्लेममध्ये रुपांतर.*

९.१ शब्दयादी वाचन (Reading a word list)

ह्या प्रकरणातील प्रश्नांसाठी आपल्याला इंग्रजी शब्दांची एक यादी लागेल. इंटरनेटवर अशा बऱ्याच याद्या उपलब्ध आहेत, पण आपण ग्रेडी वार्ड (Grady Ward) ह्यांनी मोबी शब्दकोश प्रकल्पाअंतर्गत[१] पब्लिक डोमेनला (public domain) दिलेली यादी वापरणार आहोत. ही यादी १,१३,८०९ अधिकृत शब्दकोड्यांपासून बनवलेली आहे; म्हणजे असे शब्द जे शब्दकोड्यांसाठी आणि इतर शब्दांच्या खेळांसाठी वैध आहेत. मोबी संकलनामध्ये, फाइलचे नाव `113809of.fic` आहे, पण तुम्ही `words.txt` ह्या सोप्या नावाची त्याची कॉपी पुढील लिंक वरून डाऊनलोड करू शकता: http://thinkpython2.com/code/words.txt.

ही साधी टेक्स्ट फाइल आहे (text file, म्हणजे आपण पायथॉन स्क्रिप्ट ठेवतो त्याच स्वरुपात). म्हणजेच तुम्ही ती टेक्स्ट एडिटर (text editor, उदा., gedit, नोटपॅड, emacs, vim) मध्ये उघडू शकता, पण तुम्ही ती पायथॉनमधून पण वाचू शकता; `open` नावाचे बिल्ट-इन फंक्शन फाइलचे नाव अर्ग्युमेंट म्हणून घेऊन **फाइल ऑब्जेक्ट** (**file object**) रिटर्न करते; हा ऑब्जेक्ट वापरून तुम्ही ती फाइल वाचू शकता.

```
>>> fin = open('words.txt')
```

जी फाइल इनपुट म्हणून वापरतो, तिच्या फाइल ऑब्जेक्टसाठी `fin` नाव कॉमन आहे. त्या ऑब्जेक्टला अनेक उपयुक्त मेथड्स आहेत; उदा., `readline` मेथड नवीन ओळ लागेपर्यंत फाइलमधून कॅरेक्टर्स वाचून उत्तर स्ट्रिंगस्वरुपात रिटर्न करते:

```
>>> fin.readline()
'aa\n'
```

ह्या यादीतील पहिला शब्द 'aa' आहे जो एकप्रकारचा शिलारस (lava) आहे. `\n` हे नवीन ओळ दर्शवणारे कॅरेक्टर आहे; ह्याला न्यूलाइन कॅरेक्टर (newline character) म्हणतात, आणि ते कॅरेक्टर सध्याचा शब्द पुढच्या शब्दापासून विलग करते.

[१] Moby lexicon project (http://wikipedia.org/wiki/Moby_Project बघा).

आपण सध्या फाइलमध्ये कुठे आहोत ह्याची फाइल ऑब्जेक्ट व्यवस्थित नोंद ठेवतो, म्हणजे जर तुम्ही परत `readline` मेथड कॉल केली तर तुम्हाला पुढचा शब्द मिळतो:

```
>>> fin.readline()
'aah\n'
```

पुढचा शब्द 'aah' आहे; विचित्र नजरेने पुस्तकाकडे बघणे थांबवा, हा पूर्णपणे वैध शब्द आहे. जर तुम्हाला न्यूलाइन कॅरेक्टरमुळे त्रास होत असेल तर त्याची आपण `strip` मेथडने सोय लावू शकतो:

```
>>> line = fin.readline()
>>> word = line.strip()
>>> word
'aahed'
```

आपण फाइल ऑब्जेक्ट `for` लूपमध्ये देखील वापरू शकतो. खालील प्रोग्राम `words.txt` फाइल वाचून प्रत्येक शब्द स्वतंत्र ओळीवर दाखवतो:

```
fin = open('words.txt')
for line in fin:
    word = line.strip()
    print(word)
```

९.२ प्रश्नसंच (Exercises)

ह्या प्रश्नांची उत्तरे पुढच्या विभागात आहेत. ती बघण्याआधी तुम्ही प्रत्येक प्रश्न सोडवण्याचा निदान प्रयत्न तरी करा.

प्रश्न ९.१. *असा प्रोग्राम लिहा जो* `words.txt` *फाइल वाचून त्यातील फक्त २० पेक्षा जास्ती अक्षरे असलेले शब्द प्रिंट करतो.*

प्रश्न ९.२. *अर्नेस्ट विन्सेंट राइट (Ernest Vincent Wright) ह्यांनी १९३९ साली ५०,००० शब्दांची* Gadsby *नावाची कादंबरी प्रकाशित केली; गंमत म्हणजे ह्या कादंबरीत 'e' अक्षर नाहीये! 'e' अक्षर इंग्रजीमधले सर्वांत जास्त वापरले जाणारे अक्षर असल्यामुळे हे काम सोपे नाहीये.*

वास्तविक पाहता (इंग्रजीमध्ये) त्या अक्षराशिवाय एक विचारही करणे अवघड आहे. सुरुवातीला खूप हळूहळू करता येते, पण तासंतास काळजीपूर्वक प्रयत्नांनी ते शक्य आहे.

होहो, अजून नाही बोलत ह्याविषयी.

दिलेल्या शब्दात 'e' नसेल तर `True` *रिटर्न करणारे* `has_no_e` *नावाचे फंक्शन लिहा.*

असा प्रोग्राम लिहा जो `words.txt` *फाइल वाचून त्यातील फक्त 'e' नसलेले शब्द प्रिंट करतो. यादीतील किती टक्के शब्दांमध्ये 'e' नाहीये?*

प्रश्न ९.३. *असे फंक्शन लिहा जे एक शब्द आणि एक निषिद्ध अक्षरांची स्ट्रिंग घेते, आणि जर शब्दात एकही निषिद्ध अक्षर नसेल तर* `True` *रिटर्न करते. ह्या फंक्शनला* `avoids` *हे नाव द्या.*

असा प्रोग्राम लिहा जो युझरला निषिद्ध अक्षरांची स्ट्रिंग इनपुट म्हणून द्यायला सांगतो आणि मग (आपल्या यादीतील) किती शब्दांत ती निषिद्ध अक्षरे नाहीयेत ते प्रिंट करतो (ह्या शब्दांना आपण स्वीकारार्ह शब्द म्हणूया). तुम्ही अशी ५ निषिद्ध अक्षरे शोधू शकता का जी वापरून आपल्या शब्दयादीतील जास्तीतजास्त शब्द स्वीकारार्ह ठरतील?

प्रश्न ९.४. *एक शब्द आणि अक्षरांची एक स्ट्रिंग घेणारे* `uses_only` *नावाचे फंक्शन लिहा; आणि जर दिलेल्या शब्दात फक्त दिलेलीच अक्षरे असतील, आणि इतर कोणतीही नसतील, तर हे फंक्शन* `True` *रिटर्न करेल. तुम्ही फक्त* `acefhlo` *ही अक्षरे वापरून काही वाक्ये बनवू शकता का? ('Hoe alfalfa' हे सोडून.)*

प्रश्न ९.५. *एक शब्द आणि अक्षरांची एक स्ट्रिंग घेणारे* `uses_all` *नावाचे फंक्शन लिहा; आणि दिलेल्या शब्दात दिलेले प्रत्येक अक्षर जर कमीतकमी एकदा येत असेल तर हे फंक्शन* `True` *रिटर्न करेल. असे किती शब्द आहेत जे सर्व स्वर अक्षरे, म्हणजेच* `aeiou`*, वापरतात? आणि* `aeiouy`*?*

प्रश्न ९.६. *एक शब्द घेणारे* `is_abecedarian` *नावाचे फंक्शन लिहा; आणि जर शब्दातील अक्षरे अल्फाबेटिकल क्रमाने (वर्णक्रमानुसार) असतील तर हे फंक्शन* `True` *रिटर्न करेल (एक अक्षर सलगपणे अनेक वेळा असले तरी चालेल). आपल्या यादीत किती* *abecedarian* *शब्द आहेत?*

९.३ सर्च (Search)

मागच्या विभागातील सर्व प्रश्नांमध्ये एक सामायिकता आहे; ते सर्व प्रश्न आपण विभाग ८.६ मध्ये पाहिलेल्या सर्च पॅटर्नने (search pattern) सोडवू शकतो. सोपे उदाहरण म्हणजे:

```
def has_no_e(word):
    for letter in word:
        if letter == 'e':
            return False
    return True
```

इथे `for` लूप `word` मधल्या अक्षरांवरून जातो. जर 'e' मिळाले तर आपण ताबडतोब `False` रिटर्न करू शकतो, नाहीतर आपल्याला पुढील अक्षर बघावे लागेल. जर आपण लूपमधून बाहेर पडलो तर त्याचा हा अर्थ की आपल्याला 'e' नाही मिळाले, आणि त्याठिकाणी आपण `True` रिटर्न करतो.

हे फंक्शन तुम्ही `in` ऑपरेटर वापरून जास्त संक्षेपाने लिहू शकता; पण वरची पद्धत दाखवण्याचे कारण म्हणजे अशाप्रकारे लिहिण्याने सर्च पॅटर्नचे लॉजिक (logic, युक्तिवाद) स्पष्ट होते.

खाली दिलेले `avoids` फंक्शन `has_no_e` फंक्शनपेक्षा जास्ती व्यापक (general) आहे पण त्या दोन्ही फंक्शन्सची रचना सारखीच आहे:

```
def avoids(word, forbidden):
    for letter in word:
        if letter in forbidden:
            return False
    return True
```

आपण निषिद्ध अक्षर सापडल्यावर ताबडतोब `False` रिटर्न करू शकतो; आणि जर लूप-च्या शेवटी पोहोचलो तर आपण `True` रिटर्न करतो.

खाली दिलेले `uses_only` फंक्शनपण जवळजवळ वरील फंक्शनसारखेच आहे, फक्त कंडिशन उलट केली आहे:

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

ह्याठिकाणी आपल्याकडे निषिद्ध अक्षरांच्या यादीऐवजी उपलब्ध (available) अक्षरांची यादी आहे. जर आपल्याला `word` मध्ये असे अक्षर मिळाले जे `available` मध्ये नाहीये, तर आपण `False` रिटर्न करू शकतो.

आणि, खालील `uses_all` फंक्शनही जवळजवळ `uses_only` फंक्शनसारखेच आहे, फक्त आपण शब्द आणि अक्षरांची स्ट्रिंग ह्यांच्या भूमिकांची अदलाबदल करतो:

```
def uses_all(word, required):
    for letter in required:
        if letter not in word:
            return False
    return True
```

ह्याठिकाणी, `word` मधील अक्षरांवरून जाण्याऐवजी, लूप अनिवार्य (required) अक्षरांवरून जातो. जर कोणतेही अनिवार्य अक्षर शब्दात आले नाही, तर आपण `False` रिटर्न करू शकतो.

जर तुम्ही खरेच एका संगणक वैज्ञानिकाप्रमाणे विचार केला असता, तर तुम्ही ओळखले असते की `uses_all` फंक्शन हे आधी सोडवलेल्या प्रॉब्लेमचाच नमुना आहे, आणि तुम्ही असे उत्तर लिहिले असते:

```
def uses_all(word, required):
    return uses_only(required, word)
```

आधी सोडवलेल्या प्रॉब्लेममध्ये रुपांतर (**reduction to a previously solved problem**) नावाच्या प्रोग्राम डेव्हेलपमेंट प्लानचे हे उदाहरण आहे. म्हणजेच, तुम्ही जो प्रॉब्लेम सोडवण्याचा प्रयत्न करत आहात तो आधी सोडवलेल्या प्रॉब्लेमसारखाच असून आधीच्या उत्तराचा पुनर्वापर केला जाऊ शकतो हे तुम्ही ओळखता.

९.४ इंडेक्स आणि लूप (Looping with index)

मागच्या विभागात आपण `for` लूप वापरला कारण आपल्याला स्ट्रिंग्समधली फक्त कॅरेक्टर्स पाहिजे होती, त्या कॅरेक्टर्सची इंडेक्स नव्हती पाहिजे.

पण `is_abecedarian` साठी आपल्याला लगतच्या कॅरेक्टर्सची तुलना करावी लागते; आणि हे `for` लूप वापरून थोडेसे अवघड आहे:

```
def is_abecedarian(word):
    previous = word[0]
    for c in word:
        if c < previous:
            return False
        previous = c
    return True
```

एक पर्याय आहे, तो म्हणजे रिकर्शन:

```
def is_abecedarian(word):
    if len(word) <= 1:
        return True
    if word[0] > word[1]:
        return False
    return is_abecedarian(word[1:])
```

आणखी एक पर्याय म्हणजे `while` लूप:

```
def is_abecedarian(word):
    i = 0
    while i < len(word)-1:
        if word[i+1] < word[i]:
            return False
        i = i+1
    return True
```

वरील लूप `i=0` ला सुरू होऊन `i=len(word)-1` झाले की संपतो. लूपमधून प्रत्येकवेळी जाताना, iवे कॅरेक्टर आणि $i+1$ वे (म्हणजे पुढचे) कॅरेक्टर ह्यांची तुलना केली जाते.

जर पुढचे कॅरेक्टर सध्यापेक्षा लहान (म्हणजे वर्णक्रमानुसार आधी) असेल तर हा आपल्याला abecedarian प्रवाहात एक खंड मिळाल्याचा पुरावा ठरतो, म्हणूनच आपण `False` रिटर्न करतो.

जर आपण लूप-च्या शेवटापर्यंत काहीही बिघाड *न* शोधता पोहोचलो, तर तो शब्द आपली चाचणी पास करतो. लूप बरोबर संपतो ह्याची खात्री करण्यासाठी `'flossy'` हे उदाहरण बघा. शब्दाची लांबी ६ आहे, म्हणजेच लूप जेव्हा

शेवटच्या वेळी रन होतो तेव्हा `i` ची व्हॅल्यू 4 असते, जी शेवटून दुसऱ्या कॅरेक्टरची इंडेक्स आहे. शेवटच्या इटरेशन (iteration) मध्ये तो शेवटून दुसऱ्या कॅरेक्टरची तुलना शेवटच्या कॅरेक्टरशी करतो, आणि नेमके हेच वर्तन आपल्याला अपेक्षित आहे.

(प्रश्न ६.३ बघा;) खाली `is_palindrome` फंक्शनचे दुसरे असे रूप आहे ज्यात दोन इंडसीसचा वापर केला आहे; एक सुरुवातीपासून सुरू होते आणि वाढत जाते तर दुसरी शेवटी सुरू होते आणि कमी होत जाते.

```
def is_palindrome(word):
    i = 0
    j = len(word)-1

    while i<j:
        if word[i] != word[j]:
            return False
        i = i+1
        j = j-1

    return True
```

किंवा, आपण ह्या प्रॉब्लेमचे आधी सोडवलेल्या प्रॉब्लेममध्ये रुपांतर करून असे लिहू शकतो:

```
def is_palindrome(word):
    return is_reverse(word, word)
```

विभाग ८.११ मधील `is_reverse` वापरून.

९.५ डीबगिंग (Debugging)

प्रोग्रामची टेस्टिंग करणे अवघड असते. ह्या प्रकरणातील फंक्शन्स टेस्ट करायला सोपी होती कारण त्यांची उत्तरे तुम्ही सहजपणे तपासू शकता. तरीही, सर्वप्रकारचे एरर्स टेस्ट करण्यासाठी शब्दसमूह निवडणे हे अवघड/अशक्यच आहे.

आपण `has_no_e` फंक्शनचेच उदाहरण घेऊ; दोन सरळ केसेस (cases) तपासायच्या आहेत: असे शब्द ज्यांत 'e' आहे त्यांच्यासाठी `False` रिटर्न झाले पाहिजे, आणि ज्यांत 'e' नाही त्यांच्यासाठी `True`. ह्यांची प्रत्येकी एक टेस्ट शोधून काढणे तुम्हाला नक्कीच सोपे जाईल.

प्रत्येक केसमध्ये सहजासहजी लक्षात येऊ *न* शकणाऱ्या अवघड सबकेसेस (subcases, उप'केसेस') आहेत. ज्यांत 'e' आहे, त्या शब्दांतून तुम्ही 'e' ज्यांच्यात सुरुवातीला आहे, शेवटी आहे, आणि कुठेतरी मध्ये आहे अशाप्रकारचे शब्द टेस्ट केले पाहिजेत. आणखी, मोठे शब्द, लहान शब्द, एकदम लहान शब्द, जसे एम्प्टी स्ट्रिंग (empty string) हेही तुम्ही टेस्ट केले पाहिजे. एम्प्टी स्ट्रिंग ही **स्पेशल केस** (**special case**) चे उदाहरण आहे; ही एक चटकन लक्षात *न* येणारी केस आहे आणि तिथे सहसा एरर लपून राहतात.

तुम्ही तयार केलेल्या टेस्ट केसेसमध्ये भर म्हणून `words.txt` सारखी शब्दयादी वापरूनही तुम्ही तुमचा प्रोग्राम टेस्ट करू शकता. आउटपुट बघून तुम्हाला काही एरर पकडता येतील, पण काळजी घ्या: तुम्ही फक्त एकाच प्रकारचे एरर पकडू शकाल (जे शब्द नाही घेतले पाहिजे पण घेतले गेलेत) आणि दुसऱ्या प्रकारचे सुटतील (जे शब्द घेतले पाहिजे पण घेतले नाही गेलेत).

साधारणपणे टेस्टिंगने बग्स मिळू शकतात, पण चांगल्या टेस्ट केसेस (test cases) तयार करणे हे अवघड काम आहे, आणि जरी तुम्ही चांगल्या टेस्ट केसेस बनवल्या तरी तुम्ही ही खात्री कधीच नाही देऊ शकत की तुमचा प्रोग्राम पूर्णपणे बरोबर आहे. एका अलौकिक संगणक वैज्ञानिकाचे असे म्हणणे आहे:

> Program testing can be used to show the presence of bugs, but never to show their absence!
>
> — Edsger W. Dijkstra
>
> प्रोग्राम टेस्टिंग वापरून बग्स आहेत हे दाखवता येते, पण ते नाहीयेत हे कधीच दाखवता येत नाही!
>
> — एड्सगर डब्ल्यू. डैक्स्ट्रा

९.६ शब्दार्थ

फाइल ऑब्जेक्ट (file object): उघडलेली फाइल दर्शवणारी व्हॅल्यू.

आधी सोडवलेल्या प्रॉब्लेममध्ये रुपांतर (reduction to a previously solved problem): दिलेल्या प्रॉब्लेमचे आधी सोडवलेल्या प्रॉब्लेममध्ये रुपांतर करून तो सोडवणे.

स्पेशल केस (special case): सहजासहजी लक्षात येऊ न शकणारी किंवा निराळी टेस्ट केस (अशी टेस्ट केस नीट हाताळलेली असण्याची शक्यता कमी असते).

९.७ प्रश्नसंच (Exercises)

प्रश्न ९.७. *हा प्रश्न कार टॉक (*Car Talk*) नावाच्या रेडिओवरील कार्यक्रमात प्रसारित झालेल्या एका कोड्यावर आधारित आहे (http://www.cartalk.com/content/puzzlers):*

> *मला असा एक शब्द सांगा ज्यात सलग तीन दुहेरी अक्षरे आहेत. मी तुम्हाला दोन शब्द सांगते जे जवळजवळ पात्र होतात पण पूर्ण नाही. उदा., committee, c-o-m-m-i-t-t-e-e. हा चालला असता पण 'i' च्या डोकावण्यामुळे नाही चालत. किंवा Mississippi: M-i-s-s-i-s-s-i-p-p-i. ह्यातले 'i' काढता आले असते तर हा चालला असता. पण असा एक शब्द आहे ज्यात अक्षरांच्या तीन सलग जोड्या आहेत, आणि माझ्या माहितीप्रमाणे हा एकमेव शब्द असावा. अजून ५०० ही असू शकतील पण मला एकच माहीत आहे. असा कोणता शब्द आहे?*

तो शोधण्यासाठी प्रोग्राम लिहा (`words.txt` *शब्दयादी वापरा).*

उत्तर: http://thinkpython2.com/code/cartalk1.py.

प्रश्न ९.८. *अजून एक* Car Talk *कोडे (http://www.cartalk.com/content/puzzlers):*

> *'मी परवा हायवेवर ड्राइव्ह करत होते आणि माझे ओडॉमीटरकडे लक्ष गेले. इतर ओडॉमीटरसारखेच ते सहा आकडे दाखवते, पूर्ण मैलात. म्हणजे उदा., माझ्या कारचे ३,००,००० मैल झाले असतील तर मला 3-0-0-0-0-0 दिसेल.*
>
> *'आता, मी त्या दिवशी जे पाहिले ते मजेशीर होते. माझ्या लक्षात आले की शेवटचे चार आकडे पॅलिंड्रोमिक (palindromic) होते; म्हणजे, ते सरळ आणि उलटे सारखेच होते. उदा., 5-4-4-5 एक पॅलिंड्रोम आहे, तर माझे ओडॉमीटर 3-1-5-4-4-5 असे असू शकले असेल.*
>
> *'एका मैलानंतर, शेवटचे ५ आकडे पॅलिंड्रोमिक होते. उदा., ते 3-6-5-4-5-6 असे असू शकले असेल. अजून एका मैलानंतर ६ पैकी मधले ४ पॅलिंड्रोमिक होते. आणि तुम्हाला आश्चर्य वाटेल पण अजून एका मैलानंतर, सर्व ६ पॅलिंड्रोमिक होते!*
>
> *'प्रश्न हा आहे की त्यादिवशी पहिल्यांदा मी ओडॉमीटर पाहिले तेव्हा ते काय दाखवत होते?'*

सर्व सहा-अंकी संख्या बघून त्यांपैकी वर दिलेल्या अटी पूर्ण करणाऱ्या सर्व संख्या प्रिंट करणारा पायथॉन प्रोग्राम लिहा.

उत्तर: http://thinkpython2.com/code/cartalk2.py.

प्रश्न ९.९. *आणखी एक* Car Talk *कोडे जे तुम्ही सर्च वापरून सोडवू शकता (http://www.cartalk.com/content/puzzlers):*

> *'नुकतेच मी माझ्या आईला भेटलो आणि आमच्या लक्षात आले की माझ्या वयाचे दोन आकडे उलटे केले की तिचे वय मिळते. उदा., जर तिचे वय ७३ असेल तर माझे ३७ आहे. आम्ही विचार केला की असे किती वेळा झाले असावे, पण आमचे विषयांतर झाले आणि आम्हाला उत्तर शोधता आले नाही.*

> *'घरी आल्यावर माझ्या लक्षात आले की आमच्या वयांचे आकडे आतापर्यंत सहा वेळेस एकमेकांच्या उलटे होऊन गेलेत. माझ्या हेही लक्षात आले की जर आमच्या नशीबात असेल तर असे काही वर्षांत परत होईल, आणि आमचे नशीब खूपच चांगले असेल तर त्यानंतर असे अजून एकदा होईल. दुसऱ्या शब्दांत, तोपर्यंत असे ८ वेळा होऊन गेले असेल. तर प्रश्न असा आहे की आता माझे वय काय?'*

ह्या कोड्याची उत्तरे शोधणारा पायथॉन प्रोग्राम लिहा. टीप: `zfill` *ही स्ट्रिंग मेथड उपयोगी पडू शकेल.*

उत्तर: *http://thinkpython2.com/code/cartalk3.py*.

प्रकरण १०

लिस्ट (List)

हे प्रकरण पायथॉनचा एक अतिशय उपयोगी बिल्ट-इन टाइप सादर करते, तो म्हणजे लिस्ट (list). तुम्ही ऑब्जेक्ट्सविषयी अजून शिकाल, आणि हेही शिकाल की एकाच ऑब्जेक्टची दोन नावे असतील तर काय होऊ शकते.

१०.१ लिस्ट म्हणजे सीक्वेन्स (A list is a sequence)

लिस्ट (**list**, यादी) ही स्ट्रिंगसारखीच व्हॅल्यूझचा एक सीक्वेन्स दर्शवते. स्ट्रिंगमधील व्हॅल्यूझ कॅरेक्टर्स असतात; लिस्टमध्ये कोणत्याही टाइपची व्हॅल्यू असू शकते. लिस्टमधल्या व्हॅल्यूला **एलेमेंट** (**element**) किंवा कधीकधी **आयटम** (**item**) म्हणतात.

लिस्ट बनवण्याच्या अनेक पद्धती आहेत; सर्वांत सोपी म्हणजे एलेमेंट्सना चौकटी कंसात टाकणे (square brackets, `[` आणि `]`):

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

पहिले उदाहरण चार इंटिजर्सची लिस्ट दाखवते आणि दुसरे तीन स्ट्रिंग्सची लिस्ट दाखवते. लिस्टचे एलेमेंट्स एकाच टाइपचे असणे अनिवार्य नाहीये. खालील लिस्टमध्ये आहे—एक स्ट्रिंग, एक फ्लोट, एक इंटिजर, आणि (लक्ष द्या, हो!) अजून एक लिस्ट:

```
['spam', 2.0, 5, [10, 20]]
```

लिस्टमधल्या लिस्ट-ला **नेस्टेड** (**nested**) म्हणतात.

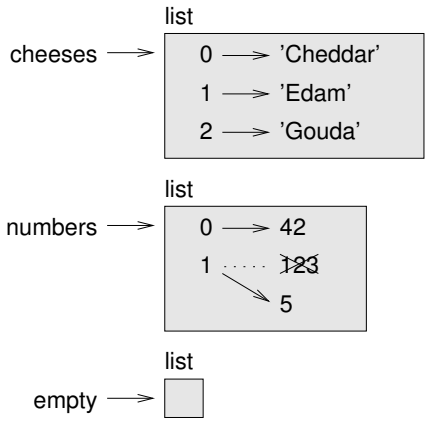
एलेमेंट्स नसलेल्या लिस्टला एम्प्टी लिस्ट म्हणतात; तुम्ही ती रिकाम्या कंसांनी बनवू शकता, `[]`.

तुम्ही हे ओळखलेच असेल की, आपण लिस्ट व्हॅल्यू व्हेरिएबलला असाइन करू शकतो:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [42, 123]
>>> empty = []
>>> print(cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [42, 123] []
```

१०.२ लिस्ट म्युटबल असते (A list is mutable)

लिस्टचे एलेमेंट्स बघण्याचा सिंटॅक्स स्ट्रिंगचे कॅरेक्टर्स बघण्याच्या सिंटॅक्ससारखाच आहे—ब्रॅकेट ऑपरेटर. ब्रॅकेट्समधले एक्स्प्रेशन इंडेक्स दर्शवते. लक्षात असू द्या की इंडेक्स शून्यापासून सुरू होते:



आकृती १०.१: स्टेट डायग्राम (State diagram).

```
>>> cheeses[0]
'Cheddar'
```

लिस्ट म्युटबल असते पण स्ट्रिंग इम्युटबल (immutable). जेव्हा ब्रॅकेट ऑपरेटर असाइनमेंटच्या डाव्या बाजूला येतो तेव्हा तो लिस्टमधला कोणता एलेमेंट असाइन होत आहे हे दर्शवतो.

```
>>> numbers = [42, 123]
>>> numbers[1] = 5
>>> numbers
[42, 5]
```

इथे, `numbers` लिस्टचा १वा एलेमेंट जो असाइनमेंटच्याआधी 123 होता तो आता 5 आहे.

आकृती १०.१ `cheeses`, `numbers` आणि `empty` ह्यांची स्टेट डायग्राम दाखवते.

लिस्ट्स ह्या बॉक्समध्ये (म्हणजे आयतामध्ये) दाखवल्या आहेत, त्या बॉक्सच्या बाहेर 'list' हा शब्द लिहिला आहे, आणि लिस्टचे एलेमेंट्स बॉक्समध्ये दाखवले आहेत. `cheeses` व्हेरिएबल तीन एलेमेंट्सची लिस्ट दर्शवते; त्या एलेमेंट्सची इंडेक्स 0, 1, आणि 2 आहे. `numbers` मध्ये दोन एलेमेंट्स आहेत आणि आकृती हे दाखवते की 1 इंडेक्स असलेल्या एलेमेंटला आधीची 123 सोडून 5 ही नवीन व्हॅल्यू असाइन झालेली आहे. `empty` व्हेरिएबल एकही एलेमेंट नसलेली लिस्ट दर्शवते.

लिस्ट इंडसीस (indices) ह्या स्ट्रिंग इंडसीस सारख्याच चालतात:

- कोणतेही इंटिजर एक्स्प्रेशन इंडेक्स म्हणून वापरता येते.
- जर लिस्टमध्ये नसणारा एलेमेंट तुम्ही वाचायचा किंवा लिहायचा प्रयत्न केलात तर तुम्हाला `IndexError` मिळतो.
- जर इंडेक्सची व्हॅल्यू ऋण (negative) असली तर ते लिस्टच्या शेवटापासून उलटे मोजले जाते.

आणि `in` ऑपरेटर लिस्टवरही चालतो.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

१०.३ लिस्ट ट्रव्हर्स करणे (Traversing a list)

लिस्ट ट्रव्हर्स करण्याचा सर्वांत कॉमन मार्ग म्हणजे `for` लूप. त्याचा सिंटॅक्स स्ट्रिंग्ससाठी जसा आहे तसाच आहे:

```
for cheese in cheeses:
    print(cheese)
```

जर एलेमेंट्स फक्त बघायचे असतील तर हे चालते, पण ते बदलायचे असतील तर इंडसीस (indices) लागतात. ते करण्याचा एक कॉमन मार्ग म्हणजे `range` आणि `len` ही बिल्ट-इन फंक्शन्स एकत्र वापरणे:

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

हा लूप, लिस्ट ट्रव्हर्स करून प्रत्येक एलेमेंट अपडेट (update) करतो. लिस्टमध्ये किती एलेमेंट्स आहेत हे `len` फंक्शन सांगते, आणि `range` फंक्शन `0` ते `len(numbers) - 1` इंटिजर्स असलेली लिस्ट पाठवते. लूपमधून प्रत्येकवेळी जाताना `i`ला पुढच्या एलेमेंटची इंडेक्स मिळते. बॉडीमधील असाइनमेंट स्टेटमेंट `i` व्हेरिएबलद्वारे एलेमेंटची जुनी व्हॅल्यू वाचून नवीन व्हॅल्यू असाइन करते.

एम्प्टी लिस्टवरचा `for` लूप बॉडीला कधीच रन करत नाही:

```
for x in []:
    print('This never happens.')
```

एका लिस्टमध्ये जरी दुसरी लिस्ट असेल तरी ती (नेस्टेड, nested) लिस्ट एलेमेंटच समजली जाते. खालील लिस्टची लांबी चार आहे:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

१०.४ लिस्टवरील क्रिया (List operations)

लिस्ट जोडण्यासाठी (concatenate करण्यासाठी) + ऑपरेटर वापरतात:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> c
[1, 2, 3, 4, 5, 6]
```

आणि * ऑपरेटर वापरून लिस्टची किती वेळा पुनरावृत्ती (repeat) करायची आहे ते सांगता येते:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

पहिले उदाहरण चार वेळा `[0]` ही लिस्ट रिपीट करते आणि दुसरे तीन वेळा `[1, 2, 3]` ही लिस्ट.

१०.५ लिस्ट स्लाइस (List slice)

स्लाइस ऑपरेटर लिस्टवर पण चालतो:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

पहिली इंडेक्स वगळली तर स्लाइस सुरुवातीपासून सुरू होतो. दुसरी वगळली तर तो शेवटपर्यंत जातो. म्हणजेच, दोन्ही वगळल्या तर स्लाइस पूर्ण लिस्टची कॉपी देतो:

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

लिस्ट म्युटबल असल्यामुळे कधीकधी लिस्टमधील एलेमेंट्स बदलण्याआधी तिची कॉपी बनवलेले फायदेशीर ठरते.

असाइनमेंटच्या डाव्या बाजूला स्लाइस ऑपरेटर वापरून अनेक एलेमेंट्स एकदमच अपडेट करता येतात:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> t
['a', 'x', 'y', 'd', 'e', 'f']
```

१०.६ लिस्टच्या मेथड्स (List methods)

पायथॉनमध्ये, लिस्टवर चालणाऱ्या अनेक मेथड्स आहेत. उदा., `append` (अपेंड) मेथड लिस्टच्या शेवटी एक नवीन एलेमेंट जोडते:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> t
['a', 'b', 'c', 'd']
```

आणि `extend` मेथड एक लिस्ट अर्ग्युमेंट घेऊन त्यातील सर्व एलेमेंट्स जोडते:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
```

वरील उदाहरणात `t2` मध्ये काहीच बदल होत नाही.

पुढे, `sort` मेथड लिस्टमधील एलेमेंट्स खालून वरपर्यंत (चढत्या क्रमाने) मांडते:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> t
['a', 'b', 'c', 'd', 'e']
```

बहुतांश लिस्ट मेथड्स व्हॉय्ड (void) आहेत; लिस्ट बदलून त्या `None` रिटर्न करतात. जर तुम्ही चुकून `t = t.sort()` लिहिले तर उत्तराने तुमची मोठी निराशा होईल.

१०.७ मॅप, फिल्टर, आणि रिड्यूस (Map, filter and reduce)

एका लिस्टमधील सर्व संख्यांची बेरीज करण्यासाठी तुम्ही लूप वापरून हे करू शकता:

```
def add_all(t):
    total = 0
    for x in t:
        total += x
    return total
```

इथे, `total` हे 0 ला इनिशलाइझ केले आहे. लूपमधून प्रत्येकवेळी जाताना `x` ला लिस्टमधील एक एलेमेंट मिळतो. आणि += हा ऑपरेटर एक व्हेरिएबल अपडेट करायची संक्षिप्त पद्धत देतो. हे **ऑगमेंटेड असाइनमेंट स्टेटमेंट** (**augmented assignment statement**),

```
total += x
```

खालील स्टेटमेंटसारखेच आहे

```
total = total + x
```

जसा लूप रन होतो, `total` मध्ये एलेमेंट्सची बेरीज जमा (accumulate) होते; अशा प्रकारे वापरलेल्या व्हेरिएबलला कधीकधी **अक्युम्युलेटर** (**accumulator**) म्हणतात.

लिस्टमधील एलेमेंट्सची बेरीज शोधणे हे इतके कॉमन आहे की पायथॉन त्यासाठी `sum` नावाचे बिल्ट-इन फंक्शन पुरवतो:

```
>>> t = [1, 2, 3]
>>> sum(t)
6
```

असे ऑपरेशन जे एलेमेंट्सच्या सीक्वेन्सचे एका व्हॅल्यूमध्ये रुपांतर करते त्याला कधीकधी **रिड्यूस** (**reduce**) म्हणतात.

कधीकधी एक लिस्ट ट्रव्हर्स करताना दुसरी बनवावी लागते. उदा., खालील फंक्शन स्ट्रिंग्सची लिस्ट घेऊन एक नवीन लिस्ट रिटर्न करते ज्यात त्या स्ट्रिंग्सचे पहिले अक्षर कॅपिटल केलेले आहे:

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

`res` हे एम्प्टी लिस्टने इनिशलाइझ केलेले आहे; लूपमधून प्रत्येकवेळी जाताना आपण पुढील एलेमेंट जोडतो. म्हणजे `res` हा अजून एका प्रकारचा अक्युम्युलेटर आहे.

कधीकधी `capitalize_all` सारख्या ऑपरेशनला **मॅप** (**map**) म्हणतात. (अनुवादकाची टिप्पणी: ह्याठिकाणी map चा अर्थ त्याच्या संबंधित क्रियापदाच्या 'सांगड घालणे' ह्या अर्थाशी निगडीत आहे; नकाशा नाही.) ह्याचे कारण म्हणजे असे ऑपरेशन एका फंक्शनला (ह्याठिकाणी `capitalize_all` ह्या मेथडला) सीक्वेन्समधील प्रत्येक एलेमेंटवर 'मॅप' करते (त्या फंक्शनची प्रत्येक एलेमेंटशी सांगड घातली जाते, म्हणजेच ते फंक्शन प्रत्येक एलेमेंटवर कॉल केले जाते).

अजून एक कॉमन ऑपरेशन म्हणजे लिस्टमधील काही एलेमेंट्स निवडून त्यांची सबलिस्ट (sublist, उप'लिस्ट') रिटर्न करणे. उदा., खालील फंक्शन स्ट्रिंग्सची लिस्ट घेऊन फक्त सर्व अक्षरे कॅपिटल असलेल्या स्ट्रिंग्सची लिस्ट रिटर्न करते:

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

`isupper` ही एक स्ट्रिंग मेथड आहे जी स्ट्रिंग्समध्ये फक्त कॅपिटल अक्षरे असतील तरच `True` रिटर्न करते.

ह्याठिकाणी `only_upper` सारख्या ऑपरेशनला **फिल्टर** (**filter**, गाळणक्रिया) म्हणतात कारण ते काही एलेमेंट्स निवडते आणि बाकीचे गाळते.

लिस्टची सर्वांत कॉमन ऑपरेशन्स ही मॅप, फिल्टर, आणि रिड्यूस ह्यांच्या एकत्रित वापराने पार पाडता येतात.

१०.८ एलेमेंट्स डिलीट करणे (काढून टाकणे, Deleting elements)

लिस्टमधून एलेमेंट्स डिलीट करण्याचे अनेक मार्ग आहेत. जर तुम्हाला एलेमेंटची इंडेक्स माहित असेल तर तुम्ही `pop` वापरू शकता:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> t
['a', 'c']
>>> x
'b'
```

`pop` लिस्टमध्ये बदल घडवून आणते आणि काढून टाकलेला एलेमेंट रिटर्न करते. जर तुम्ही इंडेक्स दिली नाहीत तर ते शेवटचा एलेमेंट डिलीट करून रिटर्न करते.

जर तुम्हाला काढून टाकलेली व्हॅल्यू नसेल पाहिजे तर तुम्ही `del` ऑपरेटर वापरू शकता:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> t
['a', 'c']
```

जर तुम्हाला कोणता एलेमेंट काढून टाकायचा आहे ते माहीत आहे (पण त्याची इंडेक्स नाही), तर तुम्ही `remove` वापरू शकता:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> t
['a', 'c']
```

`remove` ची रिटर्न व्हॅल्यू `None` आहे.

एकाहून अधिक एलेमेंट काढून टाकण्यासाठी तुम्ही स्लाइस इंडेक्स आणि `del` वापरू शकता:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> t
['a', 'f']
```

नेहमीप्रमाणे, स्लाइस दुसऱ्या इंडेक्सपर्यंत पण ती सोडून सर्व एलेमेंट्स घेतो.

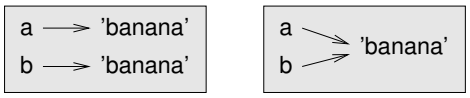
१०.९ लिस्ट्स आणि स्ट्रिंग्स (Lists and strings)

स्ट्रिंग ही कॅरेक्टर्सचा सीक्वेन्स असते तर लिस्ट ही व्हॅल्यूझचा, पण कॅरेक्टर्सची लिस्ट ही स्ट्रिंग नसते. एका स्ट्रिंगचे कॅरेक्टर्सच्या लिस्टमध्ये रुपांतर करण्यासाठी तुम्ही `list` हे फंक्शन वापरू शकता.

```
>>> s = 'spam'
>>> t = list(s)
>>> t
['s', 'p', 'a', 'm']
```

`list` हे एका बिल्ट-इन फंक्शनचे नाव असल्यामुळे तुम्ही ते व्हेरिएबलचे नाव म्हणून वापरणे टाळले पाहिजे. मी `l` सुद्धा टाळतो कारण ते `1` सारखेच दिसते. म्हणून मी `t` वापरतो.

`list` फंक्शन स्ट्रिंगचे स्वतंत्र कॅरेक्टर्समध्ये विभाजन करते. जर तुम्हाला स्ट्रिंगचे शब्दांमध्ये विभाजन करायचे असेल तर तुम्ही `split` ही मेथड वापरू शकता:



आकृती १०.२: स्टेट डायग्राम (State diagram).

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> t
['pining', 'for', 'the', 'fjords']
```

एक **डीलिमिटर** (**delimiter**) नावाचा ऑप्शनल अर्ग्युमेंट कोणती अक्षरे शब्द वेगळे करण्यासाठी हद्द म्हणून वापरायची आहेत हे दर्शवतो. खालील उदाहरणात '-' (hyphen, dash, जोडचिन्ह) हे डीलिमिटर म्हणून वापरले आहे:

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> t = s.split(delimiter)
>>> t
['spam', 'spam', 'spam']
```

`join` हे `split` च्या विरुद्ध आहे. ते स्ट्रिंग्सची लिस्ट घेऊन एलेमेंट्सना जोडते. `join` ही स्ट्रिंग मेथड आहे, म्हणून तुम्हाला ती डीलिमिटरवर इन्व्होक करून लिस्ट ही अर्ग्युमेंट म्हणून पाठवावी लागते:

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> s = delimiter.join(t)
>>> s
'pining for the fjords'
```

ह्याठिकाणी डीलिमिटर हा एक स्पेस कॅरेक्टर आहे, `join` शब्दांमध्ये एक स्पेस घालते. स्पेसेस शिवाय स्ट्रिंग्स जोडायच्या असतील तर तुम्ही `''` म्हणजेच एम्प्टी स्ट्रिंग डीलिमिटर म्हणून वापरू शकता.

१०.१० ऑब्जेक्ट्स आणि व्हॅल्यूझ (Objects and values)

जर आपण ही असाइनमेंट स्टेटमेंट्स रन केली:

```
a = 'banana'
b = 'banana'
```

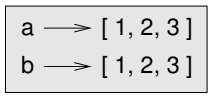
तर आपण हे सांगू शकतो की `a` आणि `b` दोन्ही एक स्ट्रिंग दर्शवतात पण आपण हे नाही सांगू शकत की ती **एकच** (*same*) स्ट्रिंग दर्शवतात. आकृती १०.२ मध्ये दाखवल्याप्रमाणे दोन शक्यता आहेत.

पहिलीत `a` आणि `b` ही दोन वेगळी ऑब्जेक्ट्स दर्शवतात ज्यांची व्हॅल्यू सारखीच आहे. दुसरीत ती एकच ऑब्जेक्ट दर्शवतात.

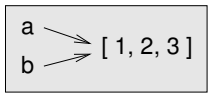
दोन व्हेरिएबल्स एकच ऑब्जेक्ट दर्शवतात का हे तपासण्यासाठी तुम्ही `is` ऑपरेटर वापरू शकता.

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

ह्या उदाहरणात पायथॉनने एकच स्ट्रिंग ऑब्जेक्ट बनवला आणि `a` आणि `b` दोन्ही तोच ऑब्जेक्ट दर्शवतात.



आकृती १०.३: स्टेट डायग्राम (State diagram).



आकृती १०.४: स्टेट डायग्राम (State diagram).

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

तर स्टेट डायग्राम आकृती १०.३ मध्ये दाखवली आहे.

ह्याठिकाणी आपण असे म्हणतो की दोन लिस्ट्स **अक्विव्हलंट**(**equivalent**, समतुल्य) आहेत कारण त्यांत सारखेच एलेमेंट्स आहेत, पण **आयडेंटिकल** (**identical**, एकच) नाहीत, कारण त्या एकच ऑब्जेक्ट नाहीत. जर दोन ऑब्जेक्ट्स आयडेंटिकल असतील तर ते अक्विव्हलंट असतात; पण जर ते अक्विव्हलंट असतील तर ते आयडेंटिकल असतीलच असे नाही.

आतापर्यंत आपण 'ऑब्जेक्ट' ('object') आणि 'व्हॅल्यू' ('value') हे एकाच अर्थाने वापरत होतो, पण असे म्हणणे जास्त बरोबर आहे की एका ऑब्जेक्टची एक व्हॅल्यू आहे. जर तुम्ही `[1, 2, 3]` हे इव्हॅल्यूएट केले तर तुम्हाला एक लिस्ट ऑब्जेक्ट मिळेल ज्याची व्हॅल्यू ही इंटिजर्सचा एक सीक्वेन्स आहे. जर दुसऱ्या कोणत्या लिस्टमध्ये हेच एलेमेंट्स असतील तर आपण म्हणतो की तिची व्हॅल्यू सारखीच आहे, पण ती वेगळा ऑब्जेक्ट आहे.

१०.११ एलिअसिंग (Aliasing)

जर `a` एक ऑब्जेक्ट दर्शवत असेल आणि तुम्ही `b = a` असे असाइन केले तर दोन्ही व्हेरिएबल्स एकच ऑब्जेक्ट दर्शवितात:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

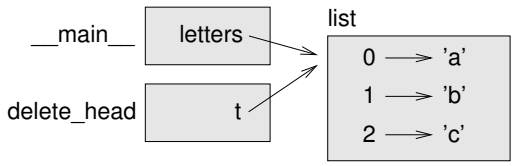
स्टेट डायग्राम आकृती १०.४ मध्ये दिली आहे.

एका व्हेरिएबल आणि एका ऑब्जेक्टमधल्या संबंधाला **रेफ्रन्स** (**reference**) असे म्हणतात. ह्या उदाहरणात एका ऑब्जेक्टला दोन रेफ्रन्सेस (references) आहेत.

एकाहून अधिक रेफ्रन्सेस असणाऱ्या ऑब्जेक्टला तितकीच नावे असतात, म्हणून आपण असे म्हणतो की तो ऑब्जेक्ट **एलिअस्ड** (**aliased**, अनेक 'ऊर्फ' नावांचा) आहे.

जर एलिअस्ड ऑब्जेक्ट म्युटबल (mutable) असेल तर एक एलिअस (नाव) वापरून केलेल्या बदलांमुळे दुसऱ्यांवर पण परिणाम होतो:

```
>>> b[0] = 42
>>> a
[42, 2, 3]
```



आकृती १०.५: स्टॅक डायग्राम (Stack diagram).

जरी हा गुणधर्म फायदेशीर असला तरी त्यामुळे चुका होऊ शकतात. साधारणपणे, म्युटबल ऑब्जेक्ट्स वापरत असताना एलिअसिंग टाळणे जास्त सुरक्षित असते.

स्ट्रिंगसारख्या इम्युटबल ऑब्जेक्ट्ससाठी एलिअसिंगने काही गडबड होत नाही. खालील उदाहरणात:

```
a = 'banana'
b = 'banana'
```

a आणि b हे एकच स्ट्रिंग दर्शवतात की नाही ह्याने क्वचितच फरक पडतो.

१०.१२ लिस्ट अर्ग्युमेंट्स (List arguments)

जेव्हा तुम्ही एका फंक्शनला लिस्ट पाठवता, तेव्हा त्या फंक्शनला त्या लिस्टचा एक रेफ्रन्स मिळतो. जर फंक्शनने लिस्टमध्ये काही बदल केला तर कॉलरला तो बदल दिसतो. उदा.delete_head लिस्टमधील पहिला एलेमेंट काढून टाकते:

```
def delete_head(t):
    del t[0]
```

ते असे वापरले जाते:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> letters
['b', 'c']
```

परॅमीटर t आणि व्हेरिएबल letters हे एकाच ऑब्जेक्टचे एलिअसेस आहेत. त्यांची स्टॅक डायग्राम आकृती १०.५ मध्ये दिली आहे.

ती लिस्ट दोन्ही फ्रेम्समध्ये वापरली असल्यामुळे त्यांच्यामध्ये काढली आहे.

लिस्ट्सना बदलणारी ऑपरेशन्स आणि नवीन लिस्ट्स बनवणारी ऑपरेशन्स ह्यांच्यातील फरक समजून घेणे महत्त्वाचे आहे. उदा., append लिस्टला बदलते पण + ऑपरेटर नवीन लिस्ट बनवतो.

खाली append चे उदाहरण आहे:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> t1
[1, 2, 3]
>>> t2
None
```

append ची रिटर्न व्हॅल्यू None आहे.

आणि हे + ऑपरेटर चे उदाहरण:

```
>>> t3 = t1 + [4]
>>> t1
[1, 2, 3]
>>> t3
[1, 2, 3, 4]
```

ह्या ऑपरेशनचे उत्तर एक नवीन लिस्ट आहे, आणि मूळ लिस्ट बदलत नाही.

तुम्ही लिस्ट बदलणारे फंक्शन्स लिहिताना हा फरक महत्त्वाचा ठरतो. उदा., खालील फंक्शन लिस्टचा पहिला एलेमेंट डिलीट *करत नाही*:

```
def bad_delete_head(t):
    t = t[1:]               # WRONG!
```

स्लाइस ऑपरेटर नवीन लिस्ट बनवतो आणि असाइनमेंटनंतर `t` त्या नवीन लिस्टला रेफर (refer) करते (ती नवीन लिस्ट दर्शवते), पण त्याचा कॉलरवर काहीच परिणाम होत नाही.

```
>>> t4 = [1, 2, 3]
>>> bad_delete_head(t4)
>>> t4
[1, 2, 3]
```

इथे `bad_delete_head` च्या सुरुवातीला `t` आणि `t4` एकाच लिस्टला रेफर करतात. शेवटी, `t` एका नवीन लिस्टला रेफर करते पण `t4` अजूनही मूळ, *न* बदललेल्या लिस्टला रेफर करते.

नवीन लिस्ट बनवून रिटर्न करणारे फंक्शन लिहिणे हा एक पर्याय आहे. उदा., `tail` पहिला सोडून लिस्टचे इतर सर्व एलेमेंट्स रिटर्न करते:

```
def tail(t):
    return t[1:]
```

हे फंक्शन मूळ लिस्ट बदलत नाही. ते असे वापरता येऊ शकते:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> rest
['b', 'c']
```

१०.१३ डीबगिंग (Debugging)

लिस्ट्सचा (आणि अन्य म्युटबल ऑब्जेक्ट्सचा) निष्काळजीपणे केलेला वापर हा डीबगिंगच्या अनेक तासांसाठी कारणीभूत ठरू शकतो. काही अज्ञात धोके आणि त्यांना टाळण्याचे मार्ग असे:

१. बहुतांश लिस्ट मेथड्स अर्ग्युमेंटमध्ये बदल घडवतात आणि `None` रिटर्न करतात. हे स्ट्रिंग मेथड्सच्या विरुद्ध आहे, ज्या नवीन स्ट्रिंग रिटर्न करतात आणि मूळ स्ट्रिंगला धक्का नाही लावत.

   जर तुम्हाला असा स्ट्रिंग कोड लिहायची सवय असेल:

   ```
   word = word.strip()
   ```

   तर तुम्हाला असा लिस्ट कोड लिहायचा मोह होऊ शकतो:

   ```
   t = t.sort()            # WRONG!
   ```

   `sort` हे `None` रिटर्न करत असल्यामुळे, `t` वरील पुढील ऑपरेशन अयशस्वी होण्याची दाट शक्यता आहे.

   लिस्ट मेथड्स आणि ऑपरेटर्स वापरण्याआधी तुम्ही डॉक्युमेंटेशन काळजीपूर्वक वाचून त्यांना इंटरॅक्टिव्ह मोडमध्ये टेस्ट केले पाहिजे.

२. एक पद्धत निवडा आणि ती एकसारखेपणाने वापरा.

लिस्टचा पेच असा आहे की एकच गोष्ट करण्याच्या अनेक पद्धती आहेत. उदा., एका लिस्टमधून एखादा एलेमेंट काढून टाकण्यासाठी तुम्ही `pop`, `remove`, `del`, किंवा स्लाइस असाइनमेंटसुद्धा वापरू शकता.

एखादा एलेमेंट घालण्यासाठी तुम्ही `append` मेथड किंवा + ऑपरेटर वापरू शकता. जर `t` ही एक लिस्ट असेल आणि `x` हा लिस्ट एलेमेंट, तर खालील पर्याय बरोबर आहेत.

```
t.append(x)
t = t + [x]
t += [x]
```

आणि हे चूक:

```
t.append([x])          # WRONG!
t = t.append(x)        # WRONG!
t + [x]                # WRONG!
t = t + x              # WRONG!
```

वरील प्रत्येक उदाहरण इंटरॅक्टिव्ह मोडमध्ये तपासून बघा आणि तुम्हाला ते काय करतात हे समजले आहे ह्याची खात्री करून घ्या. ह्याची नोंद घ्या की फक्त शेवटच्यानेच रनटाइम एरर येतो; बाकीचे जरी वैध असले तरी ते चुकीची गोष्ट करतात.

३. एलिअसिंग (aliasing) टाळण्यासाठी कॉपीझ (copies) बनवा.

जर तुम्हाला `sort` सारखी मेथड वापरायची असेल जी पाठवलेल्या लिस्टमध्ये बदल घडवून आणते, आणि जर तुम्हाला मूळ लिस्ट तशीच ठेवायची असेल तर तुम्ही तिची कॉपी बनवू शकता.

```
>>> t = [3, 1, 2]
>>> t2 = t[:]
>>> t2.sort()
>>> t
[3, 1, 2]
>>> t2
[1, 2, 3]
```

ह्या उदाहरणात तुम्ही `sorted` हे बिल्ट-इन फंक्शन वापरू शकता, जे नवीन सॉर्ट केलेली लिस्ट मूळ लिस्टला धक्का *न* लावता पाठवते.

```
>>> t2 = sorted(t)
>>> t
[3, 1, 2]
>>> t2
[1, 2, 3]
```

१०.१४ शब्दार्थ

लिस्ट (list): व्हॅल्यूझचा सीक्वेन्स.

एलेमेंट (element): लिस्ट (किंवा अन्य सीक्वेन्स) मधील एक व्हॅल्यू; आयटम (item) सुद्धा म्हणतात.

नेस्टेड लिस्ट (nested list): अशी लिस्ट जी दुसऱ्या लिस्टची एलेमेंट आहे.

अक्युम्युलेटर (accumulator): एका लूपमध्ये वापरले गेलेले व्हेरिएबल जे बेरजेसाठी किंवा उत्तर जमा करण्यासाठी (to accumulate) वापरतात.

ऑगमेंटेड असाइनमेंट (augmented assignment): असे स्टेटमेंट जे एका व्हेरिएबलची व्हॅल्यू += सारखा ऑपरेटर वापरून अपडेट करते.

रिड्यूस (reduce): एक प्रक्रियेचा पॅटर्न जो एका सीक्वेन्सच्या एलेमेंट्सची माहिती (तो सीक्वेन्स ट्रव्हर्स करून) एका उत्तरात जमा (accumulate) करतो.

मॅप (map): एक प्रक्रियेचा पॅटर्न जो एका सीक्वेन्सच्या प्रत्येक एलेमेंटवर (तो सीक्वेन्स ट्रव्हर्स करून) काहीतरी ऑपरेशन करतो.

फिल्टर (filter): एक प्रक्रियेचा पॅटर्न जो एक सीक्वेन्स ट्रव्हर्स करून त्यातून दिलेल्या अटी पूर्ण करणारे एलेमेंट्स निवडतो.

ऑब्जेक्ट (object): असे जे एक व्हेरिएबल रेफर करू (दर्शवू) शकतो. एका ऑब्जेक्टला एक टाइप आणि एक व्हॅल्यू असतात.

अक्विव्हलंट (equivalent): सारखी व्हॅल्यू असणे.

आयडेंटिकल (identical): एकच ऑब्जेक्ट असणे (जे अक्विव्हलन्सही सूचित करते).

रेफ्रन्स (reference): एका व्हेरिएबल आणि त्याच्या व्हॅल्यूमधला संबंध.

एलिअसिंग (aliasing): अशी परिस्थिती जिथे दोन किंवा जास्ती व्हेरिएबल्स एकाच ऑब्जेक्टला रेफर करतात.

डीलिमिटर (delimiter): एक कॅरेक्टर किंवा स्ट्रिंग जे (हद्दीसारखे) वापरून एका स्ट्रिंगचे विभाजन करतात.

१०.१५ प्रश्नसंच (Exercises)

ह्या प्रश्नांची उत्तरे तुम्ही पुढील लिंकवरून डाऊनलोड करू शकता http://thinkpython2.com/code/list_exercises.py. You can download

प्रश्न १०.१. *एक* `nested_sum` *नावाचे फंक्शन लिहा जे इंटिजर्सच्या लिस्ट्सची लिस्ट घेते आणि सर्व नेस्टेड लिस्ट्समधील सर्व एलेमेंट्सची बेरीज रिटर्न करते. उदा:*

```
>>> t = [[1, 2], [3], [4, 5, 6]]
>>> nested_sum(t)
21
```

प्रश्न १०.२. *एक* `cumsum` *नावाचे फंक्शन लिहा जे संख्यांची लिस्ट घेते आणि त्यांची क्युम्युलेटिव्ह (cumulative) बेरीज रिटर्न करते; जी म्हणजे एक नवीन लिस्ट ज्यात इंडेक्स i असलेला एलेमेंट हा मूळ लिस्टमधील पहिल्या $i + 1$ एलेमेंट्सची बेरीज असतो. उदा:*

```
>>> t = [1, 2, 3]
>>> cumsum(t)
[1, 3, 6]
```

प्रश्न १०.३. *एक* `middle` *नावाचे फंक्शन लिहा जे एक लिस्ट घेते आणि एक नवीन लिस्ट रिटर्न करते ज्यात पहिला आणि शेवटचा सोडून इतर सर्व एलेमेंट्स असतील. उदा:*

```
>>> t = [1, 2, 3, 4]
>>> middle(t)
[2, 3]
```

प्रश्न १०.४. *एक* `chop` *नावाचे फंक्शन लिहा जे एक लिस्ट घेते आणि त्यातील पहिला आणि शेवटचा एलेमेंट काढून टाकते (म्हणजे मूळ लिस्ट बदलते) आणि* `None` *रिटर्न करते. उदा:*

```
>>> t = [1, 2, 3, 4]
>>> chop(t)
>>> t
[2, 3]
```

प्रश्न १०.५. *एक* `is_sorted` *नावाचे फंक्शन लिहा जे एक लिस्ट घेते आणि जर ती लिस्ट चढत्या क्रमाने असेल तर* `True` *रिटर्न करते नाहीतर* `False`*. उदा:*

```
>>> is_sorted([1, 2, 2])
True
>>> is_sorted(['b', 'a'])
False
```

प्रश्न १०.६. *दोन शब्द हे अॅनाग्राम्स (anagrams) असतात जर एकाच्या अक्षरांची पुनर्रचना करून दुसरा मिळवता येत असेल. उदा., night आणि thing हे अॅनाग्राम्स आहेत. एक* `is_anagram` *नावाचे फंक्शन लिहा जे दोन स्ट्रिंग्स घेऊन त्या जर अॅनाग्राम्स असतील तर* `True` *रिटर्न करते.*

प्रश्न १०.७. *एक* `has_duplicates` *नावाचे फंक्शन लिहा जे एक लिस्ट घेऊन तिच्यात जर असा एलेमेंट असेल जो एकाहून अधिक वेळा येतो तर* `True` *रिटर्न करते. त्या फंक्शनने मूळ लिस्ट बदलली नाही पाहिजे.*

प्रश्न १०.८. *हा प्रश्न बर्थडे पॅराडॉक्स (Birthday Paradox) वर आधारित आहे, ज्याबद्दल तुम्ही पुढील लिंक वर वाचू शकता: http://en.wikipedia.org/wiki/Birthday_paradox.*

जर तुमच्या वर्गात २३ विद्यार्थी/विद्यार्थिनी असतील तर त्यांपैकी दोन जणांचे वाढदिवस सारखे असण्याची काय शक्यता आहे? तुम्ही ही संभाव्यता (probability) एस्टिमेट करण्यासाठी २३ random samples (समसंभाविक नमुने) गोळा करून जोड्या तपासू शकता. टीप: random वाढदिवस काढण्यासाठी तुम्ही `random` *मोड्युलमधील* `randint` *फंक्शन वापरू शकता.*

उत्तर: http://thinkpython2.com/code/birthday.py.

प्रश्न १०.९. *एक* `words.txt` *फाइल वाचून त्या शब्दांची (प्रत्येक शब्दासाठी एक एलेमेंट अशी) लिस्ट बनवणारे फंक्शन लिहा. हे फंक्शन दोन स्वरुपांत लिहा, एक* `append` *मेथड वापरून आणि दुसरे* `t = t + [x]` *अशा पद्धतीने. कोणते जास्ती हळू चालते? का?*

उत्तर: http://thinkpython2.com/code/wordlist.py.

प्रश्न १०.१०. *दिलेला शब्द शब्दयादीमध्ये आहे का हे तपासण्यासाठी तुम्ही* `in` *हा ऑपरेटर वापरू शकता, पण ते संथ असेल कारण तो शब्द दिलेल्या क्रमाने बघतो.*

पण ती यादी अल्फाबेटिकल क्रमाने असल्यामुळे आपण हा शोध बायसेक्शन सर्च (बायनरी सर्च, binary search, असेही म्हणतात) वापरून वेगवान करू शकतो, जसे तुम्ही डिक्शनरी (dictionary, शब्दकोश) मध्ये शब्द शोधताना करता. (अनुवादकच्या मनात शंका आहे की आजकाल कोणी अशी खरी, पुस्तकरुपातील डिक्शनरी वापरत असेल का. हे पुस्तक वाचणाऱ्यांपैकी तर नक्कीच नाही.) तुम्ही मध्यभागी बघून तपासता की तुम्हाला जो शब्द पाहिजे आहे तो बघितलेल्या शब्दाच्या आधी आहे का नंतर. आधी असेल तर तुम्ही पहिल्या अर्ध्या भागात शोधता, नाहीतर दुसऱ्या.

ह्या-नाहीतर-त्या-प्रकारे तुम्ही शोधक्षेत्र अर्धे करता. जर शब्दयादीत १,१३,८०९ शब्द असतील तर शब्द सापडायला किंवा तो नाहीये असा निष्कर्ष काढायला हे १७ वेळा करावे लागेल.

सॉर्टेड लिस्ट आणि शोधायची व्हॅल्यू घेऊन ती सापडली तर `True` *रिटर्न करणारे नाही तर* `False` *रिटर्न करणारे* `in_bisect` *नावाचे फंक्शन लिहा.*

किंवा तुम्ही `bisect` *मोड्युलचे डॉक्युमेंटेशन वाचून ते वापरू शकता! उत्तर http://thinkpython2.com/code/inlist.py.*

प्रश्न १०.११. दोन शब्द हे 'उलट जोडी' ('reverse pair') असतात जर ते एकमेकांच्या उलट असतील. दिलेल्या शब्दयादीतील सर्व 'reverse pairs' शोधणारा एक प्रोग्राम लिहा. उत्तर: http://thinkpython2.com/code/reverse_pair.py.

प्रश्न १०.१२. दोन शब्द 'interlock' होतात जेव्हा दोन्हीतून आळीपाळीने अक्षरे घेतली तर नवीन अर्थपूर्ण शब्द बनतो. उदा., 'shoe' आणि 'cold' ह्यांपासून 'schooled' बनतो. उत्तर: http://thinkpython2.com/code/interlock.py. आभार: हा प्रश्न http://puzzlers.org वरील एका उदाहरणावर आधारित आहे.

१. Interlock होणाऱ्या सर्व जोड्या शोधणारा प्रोग्राम लिहा. टीप: सर्व जोड्यांची गणती करायची गरज नाही!

२. तुम्ही three-way-interlock असलेले शब्द शोधू शकता का? म्हणजे पहिल्या, दुसऱ्या, किंवा तिसऱ्या अक्षरापासून सुरुवात करून आणि प्रत्येक तिसरे अक्षर घेऊन (म्हणजे दोन अक्षरे सोडून) नवीन अर्थपूर्ण शब्द बनतो.

प्रकरण ११

डिक्शनरी (Dictionary)

हे प्रकरण डिक्शनरी नावाचा अजून एक बिल्ट-इन टाइप सादर करते. डिक्शनरी हे पायथॉनचे एक खास वैशिष्ट्य आहे. ते वापरून अनेक कार्यक्षम आणि सुरेख अल्गोरिदम्स लिहिले गेलेत.

११.१ डिक्शनरी संबंध जुळवते (A dictionary is a mapping)

डिक्शनरी (**dictionary**) ही लिस्टसारखी पण जास्ती व्यापक (general) असते. लिस्टमध्ये इंडसीस ह्या इंटिजर्सच असू शकतात; डिक्शनरीमध्ये त्या (जवळजवळ) कोणत्याही टाइपच्या असू शकतात.

एका डिक्शनरीमध्ये इंडसीसचा एक संच असतो ज्याला **keys** (**कीझ**, किल्ल्या, चाव्या) म्हणतात आणि व्हॅल्यूझचा एक संच असतो. प्रत्येक key[१] ही एकमेव व्हॅल्यूशी संलग्न/संबंधित असते. ह्या key आणि व्हॅल्यूच्या संबंधाला **key-व्हॅल्यू जोडी** (**key-value pair**) किंवा कधीकधी **आयटम** (**item**) म्हणतात.

गणितीय भाषेत डिक्शनरी ही **मॅपिंग** (**mapping**, function, फलन) व्यक्त करते[२]. म्हणजे तुम्ही असे म्हणू शकता की प्रत्येक key एका व्हॅल्यू 'ला map' होते (इंग्रजीमध्ये, each key 'maps to' a value). एक उदाहरण म्हणून आपण इंग्रजी शब्द स्पॅनिश शब्दांना मॅप करणारी एक डिक्शनरी बनवणार आहोत, तर सर्व keys आणि व्हॅल्यूझ स्ट्रिंग्स असतील.

(कोणतेही आयटम्स नसलेली) नवीन डिक्शनरी बनवण्यासाठी `dict` हे फंक्शन वापरतात. आणि `dict` हे एका बिल्ट-इन फंक्शनचे नाव असल्यामुळे तुम्ही ते व्हेरिएबलचे नाव म्हणून वापरणे टाळले पाहिजे.

```
>>> eng2sp = dict()
>>> eng2sp
{}
```

इथे `{}` हे महिरपी कंस एम्प्टी (रिक्त) डिक्शनरी दर्शवतात. तिच्यात आयटम्स घालण्यासाठी तुम्ही चौकटी कंस वापरू शकता:

```
>>> eng2sp['one'] = 'uno'
```

ही ओळ `'one'` ही key `'uno'`[३] ह्या व्हॅल्यूला मॅप करणारा आयटम बनवते. जर आपण डिक्शनरी परत प्रिंट केली तर आपल्याला key-व्हॅल्यू जोडी ही key आणि व्हॅल्यू ह्यांच्या मधील अपूर्णविरामासहित दिसेल:

---

[१] अनुवादकाची टिप्पणी: ह्या शब्दाचे देवनागरीकरण 'की' हे एका मराठीशब्दासारखेच असल्यामुळे मी हा इंग्रजी शब्द लॅटिन लिपीतच लिहिला आहे. बाकी इंग्रजी शब्दांचे देवनागरीकरण करण्याचे कारण म्हणजे आपल्या मेंदूला वाचताना एका लिपीतून दुसऱ्या लिपीत संदर्भ बदलताना काही कार्य करावे लागते. नवीन माहिती शिकताना ह्यामुळे थोडी अडचण होऊ शकते. त्यामुळे मी इंग्रजी शब्दांबरोबरच संबंधित मराठी/देवनागरी शब्दही देण्याचा ह्या पुस्तकात प्रयत्न केला आहे. मूळ इंग्रजी मजकूरसाठी तुम्ही मूळ पुस्तक नक्कीच पाहू शकता, जे मी नक्कीच सुचवेन.

[२] Function/फलन विषयी मराठीतून सविस्तर माहिती पुढील लिंकवर https://marathivishwakosh.org/21979/

[३] म्हणजेच स्पॅनिशमध्ये 'एक'.

```
>>> eng2sp
{'one': 'uno'}
```

ह्या आउटपुटचे जसे स्वरूप आहे तसेच इनपुटचेही आहे. उदा., तुम्ही तीन आयटम्सची नवीन डिक्शनरी अशी बनवू शकता:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

पण जर तुम्ही `eng2sp` प्रिंट केले तर तुम्हाला चकित व्हाल:

```
>>> eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

key-व्हॅल्यू जोड्यांचा क्रम वेगळा असू शकतो. हेच उदाहरण जर तुम्ही तुमच्या काँप्युटरवर तपासून बघितले तर तुम्हाला वेगळे उत्तर मिळू शकते. साधारणपणे, डिक्शनरीमधील आयटम्सचा क्रम वर्तवता येत नाही.

पण ही तशी असुविधा नाहीये कारण एखादा आयटम तुम्ही तो ह्या क्रमात कितवा आहे ह्याने नाही तर त्याच्या keyने बघता:

```
>>> eng2sp['two']
'dos'
```

इथे `'two'` ही `'dos'` ला मॅप होते म्हणजेच आयटम्सच्या क्रमाचा काहीच फरक पडत नाही.

जर डिक्शनरीमध्ये key नसेल तर एक्सेप्शन मिळते:

```
>>> eng2sp['four']
KeyError: 'four'
```

`len` फंक्शन डिक्शनरीझ वर पण चालते; ते key-व्हॅल्यू जोड्यांची संख्या पाठवते:

```
>>> len(eng2sp)
3
```

आणि `in` ऑपरेटर हाही डिक्शनरीझ वर चालतो; तो एखादी गोष्ट त्या डिक्शनरीमध्ये *key म्हणून* आहे का (व्हॅल्यू म्हणून असेल किंवा नसेल).

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

एखादी गोष्ट डिक्शनरीमध्ये व्हॅल्यू म्हणून आहे का हे तपासण्यासाठी तुम्ही `values` मेथड वापरू शकता, जी व्हॅल्यूझचा संच पाठवते, आणि नंतर `in` ऑपरेटर वापरू शकता:

```
>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

हा `in` ऑपरेटर लिस्ट्स आणि डिक्शनरीझसाठी वेगवेगळे अल्गोरिदम्स वापरतो. लिस्ट्ससाठी तो दिलेला एलेमेंट लिस्टमधील क्रमाने शोधतो, जसे आपण विभाग ८.६ मध्ये पाहिले. जसजशी लिस्ट मोठी होत जाते, तसतसा शोधायला लागणारा वेळ समप्रमाणात वाढतो (direct proportion).

पायथॉन डिक्शनरी ही **हॅश-टेबल** (**hashtable**) नावाचे डेटा स्ट्रक्चर (data structure) ज्याचा एक विलक्षण गुणधर्म आहे: डिक्शनरीमध्ये कितीही आयटम्स असले तरी `in` ऑपरेटरला सारखाच वेळ लागतो (लागणारा वेळ ही राशि स्थिर असते). हे कसे शक्य आहे ते आपण विभाग प.२.४ मध्ये बघणार आहोत, पण ते समजण्यासाठी तुम्हाला अजून काही प्रकरणे वाचावी लागतील.

११.२ डिक्शनरीचा काउंटर्सचा संच म्हणून वापर (Dictionary as a collection of counters)

समजा तुम्हाला एक स्ट्रिंग दिली आहे आणि तुम्हाला मोजायचे (काउंट[४] करायचे) आहे की प्रत्येक अक्षर किती वेळा येते. हे अनेक प्रकारे करता येऊ शकते:

१. तुम्ही २६ व्हेरिएबल्स बनवू शकता, अल्फाबेटच्या प्रत्येक अक्षरासाठी एक. आणि तुम्ही स्ट्रिंग ट्रव्हर्स करून तिच्यातील प्रत्येक अक्षराशी संबंधित व्हेरिएबल एकने इन्क्रिमेंट करू (वाढवू) शकता, शक्यतो (एकात एक) गुंफलेले कंडिश्नल्स वापरून.

२. तुम्ही २६ एलेमेंट्स असलेली लिस्ट बनवू शकता. मग तुम्ही प्रत्येक अक्षराचे एका इंटिजरमध्ये रुपांतर करू शकता (`ord` हे बिल्ट-इन फंक्शन वापरून), आणि तो इंटिजर लिस्टची इंडेक्स म्हणून वापरून तिथला काउंटर वाढवू शकता.

३. तुम्ही एका डिक्शनरीमध्ये अक्षरांना keys आणि काउंटर्सना व्हॅल्यूझ म्हणून वापरू शकता. एक अक्षर पहिल्यांदा दिसले की तुम्ही त्याला डिक्शनरीमध्ये आयटम म्हणून टाकू शकता. त्यानंतर ते अक्षर परत दिसले की त्याची व्हॅल्यू तुम्ही एकने वाढवू शकता.

वरील प्रत्येक पर्याय सारखेच काँप्युटेशन (गणन) करतो पण प्रत्येक पर्यायाचे इंप्लेमेंटेशन (implementation, कार्यवाही) वेगवेगळ्या प्रकारचे आहे.

इंप्लेमेंटेशन (**implementation**) म्हणजे एखादे काँप्युटेशन करण्याची पद्धत; काही इंप्लेमेंटेशन्स इतरांपेक्षा श्रेष्ठ असतात. उदा., डिक्शनरी इंप्लेमेंटेशनचा एक फायदा हा आहे की आपल्याला स्ट्रिंगमध्ये कोणती अक्षरे आहेत ह्याची पूर्वकल्पना असण्याची गरज नाही. जसजशी अक्षरे दिसत जातील तसतशी आपण त्यांच्यासाठी डिक्शनरीमध्ये जागा बनवू शकतो.

कोड असा दिसू शकतो:

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

ह्या फंक्शनचे नाव `histogram` (हिस्टोग्राम, स्तंभालेख) आहे; हा एक सांख्यिकीशी संबंधित शब्द (statistical term) आहे, ज्याचा अर्थ काउंटर्स (किंवा वारंवारता, frequencies) ह्यांचा संच असा होतो.

फंक्शनची पहिली ओळ रिकामी डिक्शनरी बनवते. मग `for` लूप स्ट्रिंग ट्रव्हर्स करतो. लूपमधून प्रत्येकवेळी जाताना जर `c` हे कॅरेक्टर डिक्शनरीमध्ये नसेल तर आपण `c` ही key वापरून एक नवीन आयटम बनवतो आणि त्याला 1 ही सुरुवातीची व्हॅल्यू देतो (कारण आपण हे कॅरेक्टर एकदा पाहिले आहे). जर `c` आधीपासूनच डिक्शनरीमध्ये असेल तर आपण `d[c]` एकने वाढवतो.

हे असे चालते:

```
>>> h = histogram('brontosaurus')
>>> h
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

[४] हे आठवा: काउंटर (counter) म्हणजे काहीतरी मोजण्यासाठी वापरलेले व्हेरिएबल.

हा हिस्टोग्राम दर्शवतो की 'a' आणि 'b' एकदा येतात; 'o' दोनदा येते, अशाप्रकारे.

डिक्शनरीझमध्ये get नावाची मेथड आहे जी एक key आणि एक डीफॉल्ट व्हॅल्यू घेते. (अनुवादकाची टिप्पणी: ह्याठिकाणी 'default' चा अर्थ म्हणजे दुसरा पर्याय नसल्यामुळे निवडलेला प्रमाण (standard) पर्याय.) जर डिक्शनरीमध्ये key असेल तर get संबंधित व्हॅल्यू रिटर्न करते, नाहीतर डीफॉल्ट व्हॅल्यू. उदा:

```
>>> h = histogram('a')
>>> h
{'a': 1}
>>> h.get('a', 0)
1
>>> h.get('c', 0)
0
```

सराव म्हणून get वापरून histogram जास्ती संक्षिप्तपणे लिहा. तुम्हाला तिकडचे if स्टेटमेंट काढून टाकता येईल.

११.३ डिक्शनरीझ आणि लूप्स (Looping and dictionaries)

तुम्ही जर एक डिक्शनरी for स्टेटमेंटमध्ये वापरली तर ते त्या डिक्शनरीमधील keys ट्रव्हर्स करते. उदा., खालील print_hist प्रत्येक key आणि संबंधित व्हॅल्यू प्रिंट करते:

```
def print_hist(h):
    for c in h:
        print(c, h[c])
```

आउटपुट असे दिसते:

```
>>> h = histogram('parrot')
>>> print_hist(h)
a 1
p 1
r 2
t 1
o 1
```

परत बघा, keys काही विशिष्ठ क्रमाने नाहीयेत. जर keys सॉर्टेड क्रमाने ट्रव्हर्स करायच्या असतील तर तुम्ही sorted हे बिल्ट-इन फंक्शन वापरू शकता:

```
>>> for key in sorted(h):
...     print(key, h[key])
a 1
o 1
p 1
r 2
t 1
```

११.४ उलटा लूक-अप (Reverse lookup)

जर डिक्शनरी d आणि key k माहीत असेल तर संबंधित व्हॅल्यू v = d[k] शोधणे सोपे आहे. ह्या ऑपरेशनला **लूक-अप** (**lookup**, शोध) म्हणतात.

पण जर तुमच्याकडे v असेल आणि तुम्हाला k शोधायचे असेल? तुमच्यासमोर दोन अडथळे आहेत: पहिला, एकाहून अधिक keys v ला मॅप होत असतील. गरजेनुसार तुम्ही एक key निवडू शकाल, किंवा तुम्हाला सर्व संबंधित keys ची

लिस्ट बनवावी लागेल. दुसरा अडथळा हा की अशा **उलट्या लूक-अप** (**reverse lookup**) साठी सोयीस्कर सिंटॅक्स नाहीये; तुम्हाला सर्च करावे लागेल.

खालील फंक्शन एक व्हॅल्यू घेऊन त्या व्हॅल्यूला मॅप होणारी पहिली key रिटर्न करते:

```
def reverse_lookup(d, v):
    for k in d:
        if d[k] == v:
            return k
    raise LookupError()
```

हे फंक्शन सर्च पॅटर्नचे अजून एक उदाहरण आहे, पण ते असे एक वैशिष्ट्य वापरते जे आपण आधी पाहिले नाहीये, `raise`. **रेय्झ स्टेटमेंट** (**raise statement**) एक्सेप्शन निर्माण करते; ह्याठिकाणी ते `LookupError` पाठवते, जे लूक-अप विफल झाल्याचे दर्शवणारे एक बिल्ट-इन एक्सेप्शन आहे.

जर आपण लूप-च्या शेवटी पोहोचलो तर त्याचा अर्थ हा होतो की `v` डिक्शनरीमध्ये व्हॅल्यू म्हणून येत नाही, तर आपण एक एक्सेप्शन रेय्झ करतो.

खालील उदाहरणात सफल झालेला उलटा लूक-अप आहे:

```
>>> h = histogram('parrot')
>>> key = reverse_lookup(h, 2)
>>> key
'r'
```

आणि एक विफल:

```
>>> key = reverse_lookup(h, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in reverse_lookup
LookupError
```

तुम्ही एक्सेप्शन रेय्झ केल्याचा परिणाम पायथॉनने रेय्झ केल्यासारखाच होतो: एक ट्रेसबॅक (traceback) आणि एक एरर मेसेज प्रिंट होतो.

जेव्हा तुम्ही एक एक्सेप्शन रेय्झ करता तेव्हा तुम्ही सविस्तर माहिती असलेला एरर मेसेज एक पर्यायी अर्ग्युमेंट म्हणून देऊ शकता. उदा.:

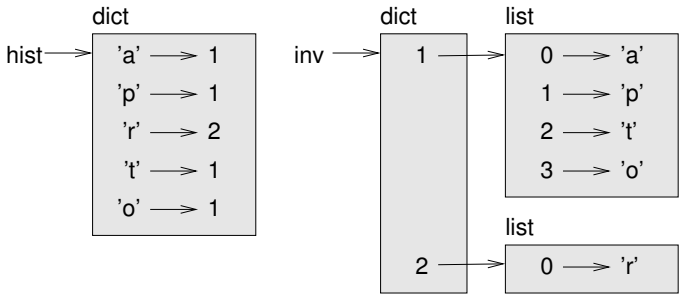
```
>>> raise LookupError('value does not appear in the dictionary')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
LookupError: value does not appear in the dictionary
```

उलटा लूक-अप सरळ लूक-अप पेक्षा खूप संथ असतो; जर तुम्हाला तो सतत करावा लागत असेल किंवा डिक्शनरी जर खूप मोठी झाली तर प्रोग्रामची कामगिरी खालावेल.

११.५ डिक्शनरीझ आणि लिस्ट्स (Dictionaries and lists)

लिस्ट एका डिक्शनरीमध्ये व्हॅल्यू म्हणून राहू शकते. उदा., समजा तुमच्याकडे एक डिक्शनरी आहे जी अक्षरे त्यांच्या वारंवारतेला (frequency) मॅप करते. आणि समजा तुम्हाला त्या डिक्शनरीला उलटे (invert) करायचे आहे, म्हणजेच अशी डिक्शनरी बनवायची आहे जी वारंवारता अक्षरांना मॅप करते. अनेक अक्षरांची सारखीच वारंवारता असू शकते, म्हणून उलट्या डिक्शनरीमध्ये प्रत्येक व्हॅल्यू ही अक्षरांची लिस्ट असायला हवी.

खालील फंक्शन उलटी डिक्शनरी देते:



आकृती ११.१: स्टेट डायग्राम (State diagram).

```
def invert_dict(d):
    inverse = dict()
    for key in d:
        val = d[key]
        if val not in inverse:
            inverse[val] = [key]
        else:
            inverse[val].append(key)
    return inverse
```

लूपमधून प्रत्येकवेळी जाताना `key` ला `d` मधून एक key मिळते आणि `val` ला संबंधित व्हॅल्यू. जर `val` `inverse` मध्ये नसेल तर त्याचा अर्थ असा की आपण ती व्हॅल्यू आधी पाहिलेली नाही, म्हणून आपण एक नवीन आयटम बनवून तो एका **सिंगलटन**ने (**singleton, एकघटक**, अशी लिस्ट ज्यात एकच एलेमेंट असतो तिने) इनिशलाइझ करतो. नाहीतर आपण ही व्हॅल्यू आधी पाहिलेली आहे, म्हणून आपण संबंधित key त्या लिस्टला जोडतो.

खाली एक उदाहरण आहे:

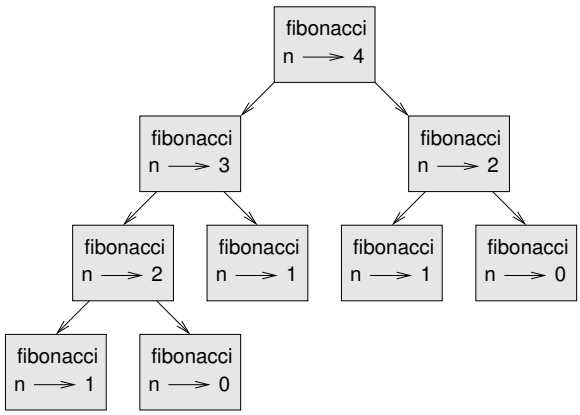
```
>>> hist = histogram('parrot')
>>> hist
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
>>> inverse = invert_dict(hist)
>>> inverse
{1: ['a', 'p', 't', 'o'], 2: ['r']}
```

आकृती ११.१ मध्ये `hist` आणि `inverse` दाखवणारी स्टॅक डायग्राम आहे. डिक्शनरी एका ठोकळ्याने दर्शवली आहे ज्याच्यावर `dict` असे लिहिलेले आहे आणि ज्याच्यामध्ये key-व्हॅल्यू जोड्या दाखवल्या आहेत. जर व्हॅल्यूझ इंटिजर्स, फ्लोट्स, किंवा स्ट्रिंग्स असतील तर त्या ठोकळ्यामध्ये दाखवल्या आहेत, पण आकृती समजायला सोपी जावी म्हणून सहसा लिस्ट्स ठोकळ्याबाहेर दाखवल्या आहेत.

लिस्ट्सना डिक्शनरीमध्ये व्हॅल्यूझ म्हणून ठेवू शकतो, जसे वरच्या उदाहरणात पाहिले, पण त्या keys म्हणून नाही ठेवू शकत. तसे करायचा प्रयत्न केल्यास हे होते:

```
>>> t = [1, 2, 3]
>>> d = dict()
>>> d[t] = 'oops'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

आधी सांगितल्याप्रमाणे डिक्शनरी इंप्लेमेंट (implement) करण्यासाठी हॅश-टेबल वापरतात, आणि म्हणूनच keys **हॅशेबल** (**hashable**, हॅश करता येण्यासारख्या) असणे गरजेचे आहे.



आकृती ११.२: कॉल ग्राफ (Call graph).

हॅश (**hash**) हे एक फंक्शन आहे जे (कोणत्याही प्रकारची) व्हॅल्यू घेऊन इंटिजर रिटर्न करते. ह्या इंटिजर्सना हॅश व्हॅल्यूझ म्हणतात ज्याचा वापर डिक्शनरी key-व्हॅल्यू जोड्या ठेवण्यासाठी आणि शोधण्यासाठी करते.

हा कार्यक्रम keys इम्युटबल असतील तर व्यवस्थित चालतो. पण जर keys म्युटबल असतील (उदा., लिस्ट्स) तर वाईट गोष्टी होऊ शकतात. उदा., जेव्हा तुम्ही एक key-व्हॅल्यू जोडी बनवता, पायथॉन ती key हॅश करून संबंधित स्थानी ती जोडी ठेवतो. जर तुम्ही key बदलली तर तिचा हॅश पण बदलतो, आणि तिचे ठेवण्याचे स्थानही बदलते. म्हणजेच, एकतर तुम्हाला एकाच key च्या दोन नोंदी मिळतील किंवा तुम्हाला ती key सापडणार नाही. कसेही, डिक्शनरी अपेक्षेप्रमाणे चालणार नाही.

म्हणून keys हॅशेबल असणे जरूरी आहे आणि लिस्ट्स सारखे म्युटबल टाइप्स हॅशेबल नसतात. ह्यावर सर्वांत सोपा उपाय म्हणजे टपल (tuple), जे आपण पुढच्या प्रकरणात बघणार आहोत.

डिक्शनरी म्युटबल असल्यामुळे तिला key म्हणून वापरता येत नाही, पण व्हॅल्यू म्हणून वापरता येते.

११.६ मेमोझ (Memos)

जर तुम्ही विभाग ६.७ मधील `fibonacci` फंक्शनवर प्रयोग करून बघितले असतील तर तुम्हाला असे आढळले असेल की जसजसे अर्ग्युमेंट मोठे होत जाते तसतसा फंक्शन रन व्हायला जास्ती वेळ लागतो. मुख्य म्हणजे, लागणारा वेळ झपाट्याने वाढतो.

हे का होते हे समजून घेण्यासाठी आकृती ११.२ बघा, जी `fibonacci` चा `n=4` घेऊन रन केल्यावरचा **कॉल ग्राफ** (**call graph**) दाखवते:

कॉल ग्राफ हा फंक्शन फ्रेम्स चा एक समूह दाखवतो. प्रत्येक फ्रेमकडून तिने जे फंक्शन कॉल केले आहेत त्यांच्या फ्रेमकडे बाण काढलेले आहेत. ग्राफमध्ये सर्वांत वर, `n=4` घेऊन `fibonacci` (परत) `fibonacci` `n=3` आणि `n=2` पाठवून कॉल करते. त्यानंतर `n=3` घेऊन `fibonacci` (परत) `fibonacci` `n=2` आणि `n=1` पाठवून कॉल करते. आणि ह्याचप्रकारे पुढे.

किती वेळा `fibonacci(0)` आणि `fibonacci(1)` कॉल होते ते मोजा. हे आपल्या प्रॉब्लेमचे अतिशय अकार्यक्षम उत्तर आहे, आणि जसजसे अर्ग्युमेंट वाढत जाते तसतसे हे अजून वाईट होत जाते.

ह्यावर एक उपाय हा आहे की जी उत्तरे (व्हॅल्यूझ) आधीच शोधलेली आहेत त्यांचा एका डिक्शनरीमध्ये साठा करणे. आधी शोधलेली व्हॅल्यू जिची भविष्यातील वापरासाठी नोंद करून (साठवून) ठेवलेली असते तिला **मेमो** (**memo**) म्हणतात. खाली `fibonacci` चे 'मेमो'करण केलेले ('memoized') रूप आहे:

```
known = {0:0, 1:1}

def fibonacci(n):
    if n in known:
        return known[n]

    res = fibonacci(n-1) + fibonacci(n-2)
    known[n] = res
    return res
```

आधीच शोधलेल्या फिबोनाची (Fibonacci) संख्यांची नोंद `known` ही डिक्शनरी ठेवते. ती दोन आयटम्सने सुरू होते: 0 ही 0ला मॅप होते, आणि 1 ही 1ला.

जेव्हाही `fibonacci` कॉल होते, ते `known` मध्ये तपासते की त्यात उत्तर आहे का. जर असेल तर ते ताबडतोब रिटर्न करू शकते. नाहीतर त्याला नवीन व्हॅल्यू काँप्युट करावी (शोधावी) लागते, ती डिक्शनरीमध्ये टाकावी लागते, आणि रिटर्न करावी लागते.

जर तुम्ही हे `fibonacci` रन करून त्याची तुलना आधीच्याशी केली तर तुम्हाला जाणवेल की हे खूप वेगाने चालते.

११.७ ग्लोबल व्हेरिएबल्स (Global variables)

मागील उदाहरणात `known` हे व्हेरिएबल फंक्शनच्या बाहेर बनवले होते, म्हणजेच ते `__main__` नावाच्या विशेष फ्रेममध्ये राहते; `__main__` मधील व्हेरिएबल्सना कधीकधी **ग्लोबल** (**global**, जागतिक) म्हणतात कारण ती कोणत्याही फंक्शनमध्ये वापरू शकतो. हे स्थानिक (local) व्हेरिएबल्सच्या विरुद्ध आहे, जी फंक्शन संपल्या संपल्या नष्ट होतात. पण ग्लोबल व्हेरिएबल्स एकापासून दुसऱ्या फंक्शन कॉलपर्यंत टिकून राहतात.

ग्लोबल व्हेरिएबल्सचा **फ्लॅग्स** (**flags**, खूणा) म्हणून वापर करणे कॉमन आहे; म्हणजे असे बूलियन व्हेरिएबल जे दर्शवते ('फ्लॅग' करते, खुणावते) की एखादी कंडिशन सत्य आहे का. उदा., काही प्रोग्राम्स `verbose` नावाचा फ्लॅग आउटपुट किती सविस्तरपणे द्यायचे आहे ह्यासाठी वापरतात:

```
verbose = True

def example1():
    if verbose:
        print('Running example1')
```

जर तुम्ही एक ग्लोबल व्हेरिएबल री-असाइन करायचा प्रयत्न केला तर तुम्हाला अनपेक्षित गोष्ट दिसेल. खालील उदाहरण फंक्शन कॉल झाले आहे की नाही ह्याची नोंद ठेवायचा प्रयत्न करते:

```
been_called = False

def example2():
    been_called = True          # WRONG
```

पण तुम्ही हे रन केले तर तुम्हाला दिसेल की `been_called` ची व्हॅल्यू बदलत नाही. गडबड ही आहे की `example2` मध्ये `been_called` नावाचे नवीन स्थानिक (local) व्हेरिएबल बनवले जाते. आणि ते स्थानिक व्हेरिएबल फंक्शन संपल्यावर नष्ट होते, आणि त्याचा ग्लोबल व्हेरिएबलवर काहीच परिणाम होत नाही.

ग्लोबल व्हेरिएबल एका फंक्शनमध्ये री-असाइन करण्यासाठी तुम्हाला ते ग्लोबल व्हेरिएबल वापरण्याआधी **डिक्लेअर**(**declare**, घोषित करावे) लागते:

```
been_called = False

def example2():
    global been_called
    been_called = True
```

ते **ग्लोबल स्टेटमेंट** (**global statement**) इंटरप्रिटरला हे सांगते की 'ह्या फंक्शनमध्ये जेव्हा `been_called` असे म्हटले की त्याचा अर्थ ग्लोबल व्हेरिएबल आहे, नवीन बनवू नये.'

खालील उदाहरण एक ग्लोबल व्हेरिएबल अपडेट करायचा प्रयत्न करते:

```
count = 0

def example3():
    count = count + 1          # WRONG
```

जर तुम्ही हे रन केले तर तुम्हाला मिळेल:

```
UnboundLocalError: local variable 'count' referenced before assignment
```

पायथॉन असे मानतो की `count` स्थानिक आहे आणि ते व्हेरिएबल तुम्ही लिहायच्या आधी वाचताहात. परत उपाय तोच, `count` ला ग्लोबल म्हणून डिक्लेअर करा.

```
def example3():
    global count
    count += 1
```

जर ग्लोबल व्हेरिएबल म्युटबल व्हॅल्यू दर्शवत असेल तर तुम्ही ते डिक्लेअर *न* करताही ती व्हॅल्यू बदलू शकता:

```
known = {0:0, 1:1}

def example4():
    known[2] = 1
```

म्हणजेच तुम्ही एका ग्लोबल लिस्टचे किंवा डिक्शनरीचे एलेमेंट्स काढू शकता, घालू शकता, आणि बदलू शकता, पण तुम्हाला जर ते व्हेरिएबल री-असाइन करायचे असेल तर तुम्हाला ते डिक्लेअर करावे लागते: So you

```
def example5():
    global known
    known = dict()
```

ग्लोबल व्हेरिएबल्स उपयोगी ठरू शकतात, पण तुम्ही खूप ग्लोबल व्हेरिएबल्स वापरली आणि सतत बदलली तर त्यामुळे प्रोग्राम डीबग करणे अवघड होऊन बसू शकते.

११.८ डीबगिंग (Debugging)

जसजसे तुम्ही मोठ्या डेटासेट्सवर (datasets) काम करता, तसतसे प्रिंट करून हातांनी आउटपुट तपासणे बोजड होते. मोठे डेटासेट्स डीबग करण्यासाठी खाली काही सूचना आहेत:

इनपुट लहान करणे: शक्य असेल तर डेटासेट लहान करायचा प्रयत्न करा. उदा., जर प्रोग्राम एखादी टेक्स्ट फाइल वाचत असेल तर सुरुवातीला फक्त १० च ओळी वाचा, किंवा शक्य तितक्या लहान उदाहरणाने सुरुवात करा. तुम्ही ती फाइल स्वत: बदलू शकता किंवा (अधिक चांगले हे की) प्रोग्राममध्ये असा बदल करा जेणेकरून तो पहिल्या `n` ओळी वाचेल.

जर एरर मिळाला, तर तुम्ही `n` एरर देणारी सर्वांत लहान व्हॅल्यू बनवा. मग जसजसे तुम्ही एरर दुरूस्त करत जाल, तसतसे `n` वाढवा.

सारांश आणि टाइप्स: पूर्ण डेटासेट प्रिंट करून तपासण्याऐवजी, डेटा-चा सारांश प्रिंट करून बघा: उदा., डिक्शनरीमधील आयटम्सची संख्या, किंवा लिस्टमधील संख्यांची बेरीज.

रनटाइम एरर्सचे एक कॉमन कारण म्हणजे एक व्हॅल्यू जिचा टाइप बरोबर नाहीये. ह्या प्रकारचे एरर डीबग करण्यासाठी सहसा व्हॅल्यूचा टाइप प्रिंट करणे पुरेसे असते.

स्वचाचण्या लिहा: कधीकधी तुम्ही आपोआप एरर तपासणीसाठी कोड लिहू शकता. उदा., जर तुम्ही संख्यांच्या लिस्टची सरासरी काढत असाल तर तुम्ही उत्तर सर्वांत मोठ्या संख्येहून मोठे नाहीये हे तपासणारी चाचणी लिहू शकता. ह्याला 'सॅनिटी चेक' ('sanity check') असे म्हणतात कारण तो 'insane' (वेडसर) उत्तरे ओळखतो.

आणखी एका प्रकारची चाचणी दोन वेगळ्या प्रकारच्या कॉंप्युटेशन्सची उत्तरे सुसंगत आहेत का नाही हे तपासते. ह्याला 'consistency check' म्हणतात.

आउटपुट व्यवस्थित दाखवा: आउटपुट व्यवस्थितशीररित्या दाखवल्याने एरर हेरणे सोपे जाते. आपण विभाग ६.९ मध्ये एक उदाहरण पाहिले. आणखी एक साधन जे तुम्हाला उपयोगी पडू शकते ते म्हणजे `pprint` मोड्युल, जे `pprint` नावाचे फंक्शन पुरवते; हे फंक्शन वापरून बिल्ट-इन टाइप्स वाचायला सोप्या पद्धतीने प्रिंट करता येतात (`pprint` नाव 'pretty print' ह्या अर्थाने ठेवले आहे).

पुन्हा एकदा: स्कॅफोल्डिंग (scaffolding) बनवण्यात केलेली मेहनत डीबगिंगचा वेळ कमी करू शकते.

११.९ शब्दार्थ

मॅपिंग (mapping): एका संचातील (set) प्रत्येक घटकाचा (element) दुसऱ्या संचातील एका घटकाशी जुळवलेला संबंध.

डिक्शनरी (dictionary): keys संबंधित व्हॅल्यूझला जोडणारी मॅपिंग.

key-व्हॅल्यू जोडी (key-value pair): मॅपिंगद्वारे एका key आणि एका व्हॅल्यूचा जोडलेला संबंध व्यक्त करण्याची पद्धत.

आयटम (item): डिक्शनरी संदर्भात, key-व्हॅल्यू जोडी चे दुसरे नाव (आयटम म्हणजे key-व्हॅल्यू जोडी).

key: डिक्शनरीमध्ये जाणाऱ्या key-व्हॅल्यू जोडी मधील पहिला भाग जो एक ऑब्जेक्ट असतो.

व्हॅल्यू (value): डिक्शनरीमध्ये जाणाऱ्या key-व्हॅल्यू जोडी मधील दुसरा भाग जो एक ऑब्जेक्ट असतो. 'व्हॅल्यू'चा हा अर्थ डिक्शनरीच्या संदर्भात आहे. आधीचा अर्थ साधारणपणे पायथॉनच्या संदर्भात आहे.

इंप्लेमेंटेशन (implementation): एखादे कॉंप्युटेशन पार पाडण्याची पद्धत.

हॅश-टेबल (hashtable): पायथॉनमध्ये डिक्शनरी इंप्लेमेंट करण्यासाठी वापरलेला अल्गोरिदम.

हॅश फंक्शन (hash function): हॅश-टेबलमध्ये एका key चे स्थान शोधण्यासाठी वापरलेले फंक्शन.

हॅशेबल (hashable): असा टाइप ज्यासाठी हॅश फंक्शन आहे. इंटिजर्स, फ्लोट्स, आणि स्ट्रिंग्स सारखे इम्युटबल टाइप्स हे हॅशेबल आहेत; तर लिस्ट्स आणि डिक्शनरीझ सारखे म्युटबल टाइप्स हॅशेबल नाहीत.

लूक-अप (lookup): key घेऊन संबंधित व्हॅल्यू शोधणारे डिक्शनरी ऑपरेशन.

उलटा लूक-अप (reverse lookup): व्हॅल्यू घेऊन एक किंवा अधिक संबंधित keys शोधणारे डिक्शनरी ऑपरेशन.

रेय्झ स्टेटमेंट (raise statement): (मुद्दामहून) एक्सेप्शन रेय्झ करणारे स्टेटमेंट.

सिंगलटन (singleton, एकघटक): एकच एलेमेंट असणारी लिस्ट (किंवा इतर सीक्वेन्स).

कॉल ग्राफ (call graph): एका प्रोग्रामच्या एक्सेक्युशन दरम्यान बनलेली प्रत्येक फ्रेम दाखवणारी आकृती, ज्यात प्रत्येक कॉल करणाऱ्यापासून प्रत्येक कॉल झालेल्यापर्यंत एक बाण असतो.

मेमो (memo): भविष्यातील अनावश्यक कॉंप्युटेशन्स टाळण्यासाठी साठवून ठेवलेली व्हॅल्यू जी आधीच शोधलेली असते.

ग्लोबल व्हेरिएबल (global variable): फंक्शनच्या बाहेर बनवलेले व्हेरिएबल. ग्लोबल व्हेरिएबल्स कोणत्याही फंक्शनमध्ये वापरले जाऊ शकतात.

ग्लोबल स्टेटमेंट (global statement): असे स्टेटमेंट जे एक व्हेरिएबल ग्लोबल आहे असे दर्शवते.

फ्लॅग (flag, खूण): असे बूलियन व्हेरिएबल जे एखादी कंडिशन सत्य आहे का हे दर्शवते.

डेक्लेरेशन (declaration): असे स्टेटमेंट (उदा., `global`) जे इंटरप्रिटरला एखाद्या व्हेरिएबल विषयी काहीतरी सांगते.

११.१० प्रश्नसंच (Exercises)

प्रश्न ११.१. *एक फंक्शन लिहा जे* `words.txt` *मधील शब्द वाचून त्यांना एका डिक्शनरीमध्ये ठेवते. व्हॅल्यूझ काय आहेत हे महत्त्वाचे नाहीये. नंतर तुम्ही* `in` *ऑपरेटर वापरून शीघ्रपणे एक स्ट्रिंग डिक्शनरीमध्ये आहे का हे तपासू शकता.*

जर तुम्ही प्रश्न १०.१० सोडवला असेल, तर तुम्ही हे इंप्लेमेंटेशन, लिस्टवरील `in` *ऑपरेटर, आणि बायसेक्शन सर्च ह्या तिन्हींच्या गतींची तुलना करू शकता.*

प्रश्न ११.२. `setdefault` *ह्या डिक्शनरी मेथडचे डॉक्युमेंटेशन वाचा आणि ती वापरून* `invert_dict` *चे संक्षिप्त रूप लिहा. उत्तर: http://thinkpython2.com/code/invert_dict.py.*

प्रश्न ११.३. *प्रश्न ६.२ मधील Ackermann फंक्शनचे 'मेमो'करण करा आणि बघा तसे केल्याने मोठ्या अर्ग्युमेंट्ससाठी ते फंक्शन इव्हॅल्यूएट करता येते का. टीप: नाही. उत्तर: http://thinkpython2.com/code/ackermann_memo.py.*

प्रश्न ११.४. *जर तुम्ही प्रश्न १०.७ सोडवला असेल तर तुमच्याकडे* `has_duplicates` *नावाचे फंक्शन असेलच, जे एक लिस्ट घेऊन जर तिच्यात जर कोणता ऑब्जेक्ट एकाहून जास्ती वेळेस असेल तर* `True` *रिटर्न करते.*

डिक्शनरी वापरून `has_duplicates` *चे जास्ती वेगवान आणि सोपे रूप लिहा. उत्तर: http://thinkpython2.com/code/has_duplicates.py.*

`has_duplicates`. *Solution: http://thinkpython2.com/code/has_duplicates.py.*

प्रश्न ११.५. *दोन शब्द 'rotate pairs' असतात जेव्हा तुम्ही एकाला फिरवून (rotate करून) दुसरा मिळवू शकता (प्रश्न ८.५ मधील* `rotate_word` *बघा).*

एक शब्दयादी वाचून त्यातील सर्व 'rotate pairs' शोधणारा प्रोग्राम लिहा. उत्तर: http://thinkpython2.com/code/rotate_pairs.py.

प्रश्न ११.६. Car Talk *मधील अजून एक कोडे (http://www.cartalk.com/content/puzzlers):*

> *हे डॅन ओ'लिअरी (Dan O'Leary) नावाच्या मनुष्याने पाठवले होते. नुकताच त्यांनी एका सिलेबलचा (syllable, उच्चाराचे एकक) पाच-अक्षरी शब्द पाहिला ज्यात खालील अद्वितीय गुणधर्म आहे. जर तुम्ही पहिले अक्षर काढले तर उर्वरीत अक्षरे मूळ शब्दाचा होमोफोन (homophone) देतात, म्हणजे असा शब्द ज्याचा उच्चार सारखाच आहे. पहिले अक्षर परत घालून दुसरे काढले तरी मूळ शब्दाचा होमोफोन मिळतो. प्रश्न असा आहे की तो शब्द कोणता?*
>
> *आता मी तुम्हाला एक न चालणारे उदाहरण देणार आहे. आपण 'wrack' हा पाच-अक्षरी शब्द बघूया. W-R-A-C-K, तुम्हाला माहीत असले जसे 'wrack with pain.' जर मी पहिले अक्षर काढले तर माझ्याकडे चार-अक्षरी शब्द 'R-A-C-K' राहतो. जसे 'Holy cow, did you see the rack[५] on that buck! It must have been a nine-pointer!' ('आई शप्पथ, तू त्या हरणाची शिंगे पाहिलीस का!*

[५] हरणाच्या शिंगांना rack म्हणतात.

त्याला नऊ टोकं तरी असतील!') हा परिपूर्ण होमोफोन आहे. जर तुम्ही 'w' परत टाकून 'r' काढला तर 'wack' राहतो, जो खरा शब्द आहे, पण तो बाकी दोन शब्दांचा होमोफोन नाहीये.

पण असा कमीतकमी एक शब्द आहे जो डॅनला आणि आम्हाला माहीत आहे, जो दोन होमोफोन्स देईल जर तुम्ही पहिल्या दोन अक्षरांपैकी कोणतेही काढून दोन नवीन चार-अक्षरी शब्द बनवले. प्रश्न असा आहे की तो शब्द कोणता?

तुम्ही प्रश्न ११.१ मधील डिक्शनरी वापरून एक स्ट्रिंग शब्दयादीमध्ये आहे का हे तपासू शकता.

दोन शब्द होमोफोन आहेत का हे तपासण्यासाठी तुम्ही CMU Pronouncing Dictionary वापरू शकता. ती तुम्ही http://www.speech.cs.cmu.edu/cgi-bin/cmudict किंवा http://thinkpython2.com/code/c06d वरून डाऊनलोड करू शकता. आणि तुम्ही http://thinkpython2.com/code/pronounce.py ही स्क्रिप्टसुद्धा डाऊनलोड करून शकता, जी `read_dictionary` *नावाचे एक फंक्शन पुरवते जे pronouncing dictionary वाचून एक पायथॉन डिक्शनरी रिटर्न करते. ती पायथॉन डिक्शनरी प्रत्येक शब्द त्याच्या स्ट्रिंग स्वरुपातील प्राथमिक उच्चाराला मॅप करते.*

कोडे सोडवणारे सगळे शब्द दाखवणारा प्रोग्राम लिहा. उत्तर: http://thinkpython2.com/code/homophone.py.

प्रकरण १२

टपल (Tuple)

ह्या प्रकरणात आपण अजून एक बिल्ट-इन टाइप बघणार आहोत, तो म्हणजे टपल (tuple), मग आपण लिस्ट्स, डिक्शनरीझ, आणि टपल्स एकत्र कसे वापरता येतात हे बघू. आणि, वैशिष्ट्यपूर्ण चल-लांबी-अर्ग्युमेंट-यादी (variable-length argument lists), गॅदर (gather) आणि स्कॅटर (scatter) ऑपरेटर्स ह्यांबद्दलसुद्धा शिकणार आहोत.

एक नोंद: 'tuple' चा उच्चार कसा करावा ह्याबद्दल एकमत नाहीये. काही लोक 'टपल' म्हणतात, पण प्रोग्रामिंगच्या संदर्भात सहसा लोक 'टूपल' म्हणतात. अनुवादकाची टिप्पणी: मी 'टपल' वापरण्याचे कारण साधे आहे, तुम्हाला दोन वेगळे उच्चार लक्षात ठेवायची गरज नाही. हा एकच उच्चार तुम्ही प्रोग्रामिंगच्या संदर्भात आणि त्याबाहेरच्या संदर्भात वापरू शकता.

१२.१ टपल्स इम्युटबल असतात (Tuples are immutable)

टपल हे व्हॅल्यूझचा सीक्वेन्स असते. व्हॅल्यूझ कोणत्याही टाइपच्या असू शकतात, आणि त्यांना इंटिजर्सचे निर्देशले (indexed) जाते, म्हणजेच त्या बाबतीत टपल लिस्टसारखेच असते. महत्त्वाचा फरक हा की टपल्स इम्युटबल असतात.

सिंटॅक्सनुसार, टपल म्हणजे व्हॅल्यूझची स्वल्पविराम-विभाजित यादी:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

टपल्सना साध्या कंसामध्ये टाकण्याची पद्धत आहे (असे जरूरी नसले तरी):

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

एकाच एलेमेंटचे टपल बनवण्यासाठी तुम्हाला शेवटी स्वल्पविराम द्यावा लागतो:

```
>>> t1 = 'a',
>>> type(t1)
<class 'tuple'>
```

कंसातील व्हॅल्यू ही टपल नसते:

```
>>> t2 = ('a')
>>> type(t2)
<class 'str'>
```

टपल बनवण्याचा अजून एक मार्ग म्हणजे बिल्ट-इन फंक्शन `tuple`. अर्ग्युमेंट *न* देता कॉल केल्यास ते रिकामे टपल बनवते

```
>>> t = tuple()
>>> t
()
```

जर अर्ग्युमेंट एक सीक्वेन्स (स्ट्रिंग, लिस्ट, टपल) असेल तर उत्तर म्हणून त्या सीक्वेन्सचे एलेमेंट्स असलेले टपल मिळते:

```
>>> t = tuple('lupins')
>>> t
('l', 'u', 'p', 'i', 'n', 's')
```

`tuple` हे एका बिल्ट-इन फंक्शनचे नाव असल्यामुळे तुम्ही त्याचा व्हेरिएबलचे नाव म्हणून उपयोग टाळावा.

बहुतांश लिस्ट ऑपरेटर्स टपलवर पण चालतात. ब्रॅकेट (bracket) ऑपरेटर एक एलेमेंट निर्देशतो:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t[0]
'a'
```

आणि स्लाइस ऑपरेटर एलेमेंट्सचा पट्टा (range) निवडतो.

```
>>> t[1:3]
('b', 'c')
```

पण जर तुम्ही टपलचा एक एलेमेंट बदलायचा प्रयत्न केला तर तुम्हाला एरर मिळतो:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

टपल इम्युटबल असल्यामुळे तुम्ही एलेमेंट्स बदलू शकत नाही. पण तुम्ही एका टपलला (पूर्णपणे) दुसऱ्याने बदलू शकता:

```
>>> t = ('A',) + t[1:]
>>> t
('A', 'b', 'c', 'd', 'e')
```

हे स्टेटमेंट एक नवीन टपल बनवते आणि नंतर `t` ला त्या नवीन टपलला दर्शवायला लावते.

रिलेश्नल ऑपरेटर्स टपल्स आणि इतर सीक्वेन्सेसवरही चालतात; पायथॉन प्रथम दोन्ही सीक्वेन्सेसच्या पहिल्या एलेमेंटची तुलना करतो. जर ते सारखे असतील, तर पुढचे एलेमेंट्स बघतो, आणि असेच पुढे जोपर्यंत त्याला वेगळे एलेमेंट्स मिळत नाहीत, आणि त्यानंतरचे एलेमेंट्स बघितले जात नाहीत (कितीही मोठे असले तरी).

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

१२.२ टपल असाइनमेंट (Tuple assignment)

कधीकधी दोन व्हेरिएबल्स च्या व्हॅल्यूझची अदलाबदली (swap, स्वाप) करावी लागते. पारंपारिक पद्धतीने असाइनमेंट वापरून हे करायचे झाले तर एक टेंपररी व्हेरिएबल वापरावे लागते. उदा `a` आणि `b` स्वाप करण्यासाठी:

```
>>> temp = a
>>> a = b
>>> b = temp
```

हे थोडे बेडौल आहे; **टपल असाइनमेंट** (**tuple assignment**) जास्ती सुंदर आहे:

```
>>> a, b = b, a
```

डावी बाजू व्हेरिएबल्सचे टपल आहे; उजवी बाजू एक्स्प्रेशन्सचे टपल आहे. प्रत्येक व्हॅल्यू संबंधित व्हेरिएबलला असाइन केली जाते. उजव्या बाजूची सर्व एक्स्प्रेशन्स कोणतीही असाइनमेंट होण्याच्या आधी शोधली जातात.

डावीकडील व्हेरिएबल्सची संख्या आणि उजवीकडील व्हॅल्यूझची संख्या सारखीच असली पाहिजे:

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

साधारणपणे उजवी बाजू कोणत्याही प्रकारचा सीक्वेन्स असू शकते (स्ट्रिंग, लिस्ट, किंवा टपल). उदा., एका ईमेल अड्रेसचे (email address) युझर-नेम (user name) आणि डोमेन (domain) मध्ये विभाजन करण्यासाठी तुम्ही असे लिहू शकता:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

ह्याठिकाणी `split` ची रिटर्न व्हॅल्यू ही दोन एलेमेंट्स असलेली एक लिस्ट आहे; पहिला एलेमेंट `uname` ला असाइन होतो आणि दुसरा `domain` ला.

```
>>> uname
'monty'
>>> domain
'python.org'
```

१२.३ रिटर्न व्हॅल्यू म्हणून टपलचा वापर (Tuples as return values)

खरे तर एक फंक्शन एकच व्हॅल्यू रिटर्न करू शकते, पण जर ती व्हॅल्यू टपल असेल तर त्याचा परिणाम हा अनेक व्हॅल्यूझ रिटर्न केल्यासारखाच होतो. उदा., जर तुम्हाला एका इंटिजरला दुसऱ्याने भागून त्यांचा भागाकार आणि बाकी शोधायचे असेल तर आधी `x//y` आणि नंतर `x%y` शोधणे अकार्यक्षम आहे. दोन्ही एकाच वेळेला काँप्युट करणे जास्ती चांगले.

`divmod` हे बिल्ट-इन फंक्शन दोन अर्ग्युमेंट्स घेऊन दोन व्हॅल्यूझचा टपल रिटर्न करते, एक भागाकार आणि दुसरी बाकी. तुम्ही उत्तर टपल म्हणून ठेवू शकता:

```
>>> t = divmod(7, 3)
>>> t
(2, 1)
```

किंवा टपल असाइनमेंट वापरून एलेमेंट्स स्वतंत्रपणे ठेवू शकता:

```
>>> quot, rem = divmod(7, 3)
>>> quot
2
>>> rem
1
```

खालील उदाहरणातील फंक्शन टपल रिटर्न करते:

```
def min_max(t):
    return min(t), max(t)
```

`max` आणि `min` ही बिल्ट-इन फंक्शन्स आहेत जी सीक्वेन्सचा अनुक्रमे सर्वांत मोठा एलेमेंट आणि सर्वांत लहान एलेमेंट शोधून काढतात. `min_max` दोन्ही शोधून दोन व्हॅल्यूझचे टपल रिटर्न करते.

१२.४ चल-लांबी-अर्ग्युमेंट-यादी साठी टपलचा वापर (Variable-length argument tuples)

फंक्शन्स कितीही अर्ग्युमेंट्स घेऊ शकतात. नाव `*` ने सुरू होणारा परॅमीटर हा अर्ग्युमेंट्स एका टपलमध्ये **गॅदर** करतो (**gathers**). उदा., `printall` कितीही अर्ग्युमेंट्स घेऊन ते प्रिंट करते:

```
def printall(*args):
    print(args)
```

अनुवादकाची टिप्पणी: अशाप्रकारे लिहिलेल्या फंक्शनला १, २, ३, ... कितीही अर्ग्युमेंट्स देता येतात, म्हणूनच ह्याला चल-लांबी-अर्ग्युमेंट-यादी म्हटलेले आहे. तिची लांबी चल (म्हणजे स्थिरच्या विरुद्ध) आहे.

गॅदर परॅमीटरचे नाव काहीही असू शकते, पण `args` हे नाव वापरण्याची पद्धत आहे. ते फंक्शन असे चालते:

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

गॅदरच्या थोडे उलट असे **स्कॅटर** (**scatter**) आहे. जर तुमच्याकडे व्हॅल्यूझचा सीक्वेन्स असेल आणि तुम्हाला तो एका फंक्शनला अनेक अर्ग्युमेंट्सच्या स्वरुपात पाठवायचा असेल, तर तुम्ही `*` ऑपरेटर वापरू शकता. उदा., `divmod` हे दोन म्हणजे दोनच अर्ग्युमेंट्स घेते; त्याला टपल देऊन चालणार नाही:

```
>>> t = (7, 3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
```

पण तुम्ही जर ते टपल स्कॅटर केले तर ते चालेल:

```
>>> divmod(*t)
(2, 1)
```

अनेक बिल्ट-इन फंक्शन्स चल-लांबी-अर्ग्युमेंट-यादी घेण्यासाठी टपलचा वापर करतात. उदा., `max` आणि `min` कितीही अर्ग्युमेंट्स घेऊ शकतात:

```
>>> max(1, 2, 3)
3
```

पण `sum` हे नाही.

```
>>> sum(1, 2, 3)
TypeError: sum expected at most 2 arguments, got 3
```

सराव म्हणून, `sum_all` नावाचे फंक्शन लिहा, जे कितीही अर्ग्युमेंट्स घेऊन त्यांची बेरीज रिटर्न करते.

१२.५ लिस्ट आणि टपल (Lists and tuples)

`zip` हे एक बिल्ट-इन फंक्शन आहे जे दोन किंवा अधिक सीक्वेन्सेस घेऊन त्यांना गुंफते (interleave करते). हे नाव अमेरिकन इंग्रजीमधील zipper (भारतीय/ब्रिटिश इंग्रजीमध्ये चेन, chain, आपल्या दप्तराला, कपड्यांना असते ती) ह्या शब्दावरून ठेवले आहे, कारण zipper दोन भागांना एका गुंफणाऱ्या यंत्रणेने जोडते.

हे उदाहरण एका स्ट्रिंग आणि एका लिस्टला झिप करते:

```
>>> s = 'abc'
>>> t = [0, 1, 2]
>>> zip(s, t)
<zip object at 0x7f7d0a9e7c48>
```

उत्तर एक **झिप ऑब्जेक्ट** (**zip object**) असते जे वापरून जोड्यांवरून इटरेट (iterate) करता येते. `zip` चा सर्वांत कॉमन उपयोग हा `for` लूप-मध्ये होतो:

```
>>> for pair in zip(s, t):
...     print(pair)
...
('a', 0)
('b', 1)
('c', 2)
```

झिप ऑब्जेक्ट हा एक प्रकारचा **इटरेटर** (**iterator**) असतो, म्हणजे असा ऑब्जेक्ट जो एका सीक्वेन्सवरून इटरेट करतो. इटरेटर्स हे काही प्रकारे लिस्ट्स सारखे असतात, पण तुम्ही त्यांच्यावर इंडेक्स वापरून एलेमेंट नाही निवडू शकत.

जर तुम्हाला लिस्ट ऑपरेटर्स आणि मेथड्स वापरायच्या असतील तर तुम्ही तो झिप ऑब्जेक्ट वापरून एक लिस्ट बनवू शकता:

```
>>> list(zip(s, t))
[('a', 0), ('b', 1), ('c', 2)]
```

त्याचे उत्तर हे टपल्सची लिस्ट असते; ह्या उदाहरणात, प्रत्येक टपलमध्ये स्ट्रिंगमधील एक कॅरेक्टर आणि लिस्टमधील संबंधित एलेमेंट असतात.

जर सीक्वेन्सेसची लांबी सारखी नसेल तर उत्तराची लांबी लहान सीक्वेन्सच्या लांबीइतकी असते.

```
>>> list(zip('Anne', 'Elk'))
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

तुम्ही `for` लूपमध्ये टपल असाइनमेंट वापरून टपल्सची लिस्ट खालीलप्रमाणे ट्रव्हर्स करू शकता:

```
t = [('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
    print(number, letter)
```

लूपमधून प्रत्येकवेळी जाताना पायथॉन पुढचे टपल निवडते आणि त्यातील (दोन) एलेमेंट्स `letter` आणि `number` ह्यांना असाइन करते. ह्या लूप-चे आउटपुट असे आहे:

```
0 a
1 b
2 c
```

जर तुम्ही `zip`, `for`, आणि टपल असाइनमेंटचा एकत्र वापर केला तर तुम्हाला दोन (किंवा अधिक) सीक्वेन्सेस ट्रव्हर्स करण्याची एक छान पद्धत मिळते. उदा., `has_match` दोन सीक्वेन्सेस घेते, `t1` आणि `t2`, आणि जर त्यांमध्ये कोणत्यातरी इंडेक्स वर सारखेच एलेमेंट्स असतील तर `True` रिटर्न करते (म्हणजेच `t1[i] == t2[i]` असे असणारी कोणतीतरी इंडेक्स `i` असेल तर):

```
def has_match(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
    return False
```

जर तुम्हाला एखाद्या सीक्वेन्समधील एलेमेंट्स त्यांच्या इंडसीस-सहित इटरेट करायचे असतील तर तुम्ही `enumerate` हे बिल्ट-इन फंक्शन वापरू शकता:

```
for index, element in enumerate('abc'):
    print(index, element)
```

`enumerate` एक एन्युमरेट (enumerate) ऑब्जेक्ट रिटर्न करते, जो जोड्यांचा सीक्वेन्स इटरेट करतो (जोड्या/pairs म्हणजे दोन एलेमेंट्सचे टपल). प्रत्येक जोडीत एक इंडेक्स (शून्यापासून सुरुवात करून) आणि दिलेल्या सीक्वेन्समधील एक एलेमेंट हे असतात. ह्या उदाहरणात आउटपुट असे आहे:

```
0 a
1 b
2 c
```

परत.

१२.६ डिक्शनरी आणि टपल (Dictionaries and tuples)

डिक्शनरीमध्ये `items` नावाची एक मेथड असते जी टपल्सचा सीक्वेन्स रिटर्न करते, ज्यातील प्रत्येक टपल हे key-व्हॅल्यू जोडी असते.

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> t
dict_items([('c', 2), ('a', 0), ('b', 1)])
```

उत्तर `dict_items` ऑब्जेक्ट आहे, जो एक key-व्हॅल्यू जोड्या इटरेट करणारा इटरेटर (iterator) आहे. तुम्ही तो `for` लूपमध्ये असा वापरू शकता:

```
>>> for key, value in d.items():
...     print(key, value)
...
c 2
a 0
b 1
```

एका डिक्शनरीकडून आपण जशी अपेक्षा करू, आयटम्स कोणत्याही विशिष्ठ क्रमात नाहीयेत.

दुसऱ्या बाजूस, तुम्ही टपल्सची लिस्ट वापरून एक नवीन डिक्शनरी बनवू शकता:

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

`dict` आणि `zip` ह्यांचा एकत्र वापर करून डिक्शनरी चटकन बनवण्याचा एक मार्ग मिळतो:

```
>>> d = dict(zip('abc', range(3)))
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

`update` ही डिक्शनरी मेथडसुद्धा टपल्सची लिस्ट घेऊन त्यांना key-व्हॅल्यू जोड्या म्हणून त्या डिक्शनरीमध्ये घालते.

डिक्शनरीमध्ये टपल्सना keys म्हणून वापरणे हे कॉमन आहे (मुख्यत: कारण तुम्ही लिस्ट्स वापरू शकता नाही). उदा., टेलीफोन डिरेक्टरी आडनाव, नाव जोड्या फोन-नंबर्सना मॅप करू शकते. समजा आपण `last`, `first`, आणि `number` ही व्हेरिएबल्स आधीच बनवली आहेत, तर आपण असे लिहू शकतो:

```
directory[last, first] = number
```

चौकटी कंसातील (brackets मधील) एक्स्प्रेशन हे टपल आहे. आपण टपल असाइनमेंट वापरून ही डिक्शनरी ट्रव्हर्स करू शकतो.

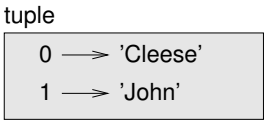
```
for last, first in directory:
    print(first, last, directory[last,first])
```

हा लूप `directory` मधील टपल असलेल्या keys ट्रव्हर्स करतो. तो प्रत्येक टपलमधील (दोन) एलेमेंट्स अनुक्रमे `last` आणि `first` ना असाइन करतो, आणि नंतर नाव आणि टेलीफोन नंबर प्रिंट करतो.

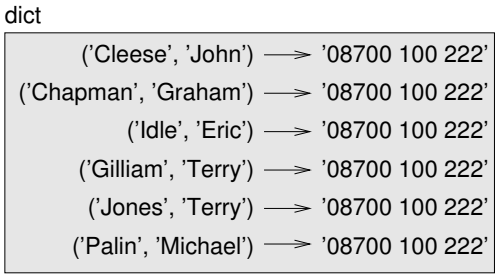
स्टेटमेंट डायग्राममध्ये टपल दाखवण्याचे दोन मार्ग आहेत. जास्ती सविस्तर पद्धतीत लिस्टसारखेच इंडसीस (indices) आणि एलेमेंट्स दाखवले जातात. उदा., `('Cleese', 'John')` हे टपल आकृती १२.१ मध्ये दाखवले आहे.

पण मोठ्या आकृतींमध्ये तुम्ही हा तपशील वगळलेला बरा. उदा., टेलीफोन डिरेक्टरीची आकृती १२.२ मध्ये दाखवलेली स्टेट डायग्राम.

आकृती १२.२ मध्ये, टपल्स दाखवण्यासाठी पायथॉनचाच सिंटॅक्स वापरला आहे. तिथे दाखवलेला टेलीफोन नंबर BBC च्या तक्रार-नोंदणी विभागाचा नंबर आहे, तर कृपया त्या नंबरला फोन करू नका.



आकृती १२.१: स्टेट डायग्राम (State diagram).



आकृती १२.२: स्टेट डायग्राम (State diagram).

१२.७ सीक्वेन्सेसचा सीक्वेन्स (Sequences of sequences)

आपण टपल्सच्या लिस्ट्स पाहिल्या, पण ह्या प्रकरणातील जवळजवळ सर्वच उदाहरणे लिस्ट्सच्या लिस्ट्स, टपल्सची टपल्स, आणि लिस्ट्सची टपल्स ह्यांवर पण चालतात. ह्या सर्व संचयांची गणती करण्यापेक्षा कधीकधी सीक्वेन्सेसच्या सीक्वेन्सेसबद्दल चर्चा करणे सोपे जाते.

अनेक ठिकाणी, वेगवेगळ्या प्रकारचे सीक्वेन्सेस (स्ट्रिंग, लिस्ट्स, आणि टपल्स) अदलाबदली करून वापरू शकतो. तर तुम्ही कोणता निवडाल?

सरळ आहे की स्ट्रिंगच्या इतर सीक्वेन्सेसपेक्षा जास्ती मर्यादा आहेत कारण एलेमेंट्स कॅरेक्टर्स असावे लागतात. आणि तेसुद्धा इम्युटबल अहेत. जर तुम्हाला कॅरेक्टर्स बदलायची क्षमता पाहिजे असेल, तर तुम्ही स्ट्रिंग ऐवजी कॅरेक्टर्सची लिस्ट वापरा.

टपलपेक्षा लिस्ट जास्ती कॉमन आहे मुख्यतः कारण लिस्ट म्युटबल असते. पण काही परिस्थितींमध्ये तुम्ही टपल वापरणे जास्ती सोयीस्कर ठरते:

१. काही ठिकाणी, उदा., रिटर्न (`return`) स्टेटमेंटमध्ये, टपल बनवण्याचा सिंटॅक्स लिस्टपेक्षा सोपा आहे.

२. जर तुम्हाला सीक्वेन्स एका डिक्शनरीमध्ये key म्हणून वापरायचा असेल तर तुम्हाला टपल किंवा स्ट्रिंगसारखा इम्युटबल टाइप वापरणे जरूरी आहे.

३. जर तुम्हाला सीक्वेन्स एका फंक्शनला अर्ग्युमेंट म्हणून पाठवायचा असेल तर टपल वापरल्याने एलिअसिंगमुळे (aliasing) होऊ शकणाऱ्या अनपेक्षित गोष्टी टाळता येऊ शकतात.

टपल इम्युटबल असल्यामुळे त्याच्याकडे `sort` आणि `reverse` असल्या लिस्ट्स बदलणाऱ्या मेथड्स नसतात. पण पायथॉनमध्ये `sorted` हे बिल्ट-इन फंक्शन आहे जे कोणताही सीक्वेन्स घेऊन तेच एलेमेंट्स सॉर्टेड (sorted) क्रमाने असलेली नवीन लिस्ट रिटर्न करते, आणि `reversed` एक सीक्वेन्स घेऊन त्या सीक्वेन्सला उलट्या क्रमाने ट्रव्हर्स करणारा इटरेटर (iterator) रिटर्न करते.

१२.८ डीबगिंग (Debugging)

लिस्ट्स, डिक्शनरीझ, आणि टपल्स ही **डेटा स्ट्रक्चर** (**data structure**) ची उदाहरणे आहेत; ह्या प्रकरणात आपण संयुक्त डेटा स्ट्रक्चर्स बघायला सुरुवात केली, जसे टपल्सची लिस्ट, किंवा टपल्स keys म्हणून आणि लिस्ट्स व्हॅल्यूझ

म्हणून असणारी डिक्शनरी. संयुक्त डेटा स्ट्रक्चर्स उपयोगी असतात पण ते वापरताना **स्वरुप एरर** (**shape error**) होण्याची शक्यता वाढते, म्हणजे डेटा स्ट्रक्चरच्या चुकीच्या टाइप, लांबी, किंवा रचनेमुळे झालेला एरर. उदा., जर तुम्ही एकच इंटिजर असलेल्या लिस्टची अपेक्षा करत असाल आणि मी तुम्हाला साधा इंटिजर दिला (लिस्टमध्ये नाही) तर ते चालणार नाही.

ह्या प्रकारचे एरर्स डीबग करण्यासाठी मी (मूळ लेखकाने) `structshape` नावाचे मोड्युल लिहिले आहे जे एक फंक्शन पुरवते, त्याचे नावसुद्धा `structshape` आहे, जे कोणत्याही प्रकारचे डेटा स्ट्रक्चर अर्ग्युमेंट म्हणून घेऊन त्याच्या स्वरुपाविषयी (shape) माहिती सांगणारी स्टेटमेंट रिटर्न करते. तुम्ही ते http://thinkpython2.com/code/structshape.py वरून डाऊनलोड करू शकता.

साध्या लिस्टवर ते असे चालते:

```
>>> from structshape import structshape
>>> t = [1, 2, 3]
>>> structshape(t)
'list of 3 int'
```

एखाद्या भपकेदार प्रोग्रामने 'list of 3 ints' असे लिहिले असते, पण अनेकवचनांची काळजी न करणे सोपे पडले. खाली एक लिस्ट्सची लिस्ट आहे:

```
>>> t2 = [[1,2], [3,4], [5,6]]
>>> structshape(t2)
'list of 3 list of 2 int'
```

जर लिस्टचे एलेमेंट्स एकाच टाइपचे नसतील तर `structshape` त्यांचे क्रमाने टाइपनुसार गट बनवते:

```
>>> t3 = [1, 2, 3, 4.0, '5', '6', [7], [8], 9]
>>> structshape(t3)
'list of (3 int, float, 2 str, 2 list of int, int)'
```

खाली टपल्सची लिस्ट आहे:

```
>>> s = 'abc'
>>> lt = list(zip(t, s))
>>> structshape(lt)
'list of 3 tuple of (int, str)'
```

आणि आता इंटिजर्स स्ट्रिंग्सना मॅप करणाऱ्या तीन आयटम्सची डिक्शनरी:

```
>>> d = dict(lt)
>>> structshape(d)
'dict of 3 int->str'
```

जर तुम्हाला तुमच्या डेटा स्ट्रक्चर्सची नोंद ठेवणे जिकीरीचे ठरत असेल तर `structshape` ची तुम्हाला मदत होईल.

१२.९ शब्दार्थ

टपल (tuple): एलेमेंट्सचा एक इम्युटबल सीक्वेन्स.

टपल असाइनमेंट (tuple assignment): अशी असाइनमेंट ज्यात उजव्या बाजूला एक सीक्वेन्स असतो आणि डाव्या बाजूला व्हेरिएबल्सचे टपल असते. आधी उजवी बाजू इव्हॅल्यूएट (evaluate) केली जाते आणि त्यातील एलेमेंट्स डाव्या बाजूस असणाऱ्या संबंधित व्हेरिएबल्सना असाइन केले जातात.

गॅदर (gather): असे ऑपरेशन जे अनेक अर्ग्युमेंट्स एका टपलमध्ये जमा करते.

स्कॅटर (scatter): असे ऑपरेशन जे एका सीक्वेन्सचे अनेक अर्ग्युमेंट्समध्ये रुपांतर करते.

झिप ऑब्जेक्ट (zip object): `zip` हे बिल्ट-इन फंक्शन कॉल केल्यावर मिळणारा ऑब्जेक्ट जो टपल्सच्या एका सीक्वेन्सवर इटरेट करतो.

इटरेटर (iterator): असा ऑब्जेक्ट जो एका सीक्वेन्सवर इटरेट करू शकतो पण लिस्ट ऑपरेटर्स आणि मेथड्स नाही पुरवत.

डेटा स्ट्रक्चर (data structure): एकमेकांशी संबंधित व्हॅल्यूझचा संच, सहसा ज्याची रचना लिस्ट्स, डिक्शनरीझ, टपल्स, इ. मध्ये केलेली असते.

स्वरूप एरर (shape error): व्हॅल्यूचे स्वरूप चुकीचे असेल तर होणारा एरर; म्हणजे चुकीचा टाइप किंवा लांबी.

१२.१० प्रश्नसंच (Exercises)

प्रश्न १२.१. *एक* `most_frequent` *नावाचे फंक्शन लिहा जे एक स्ट्रिंग घेऊन त्यातील अक्षरे वारंवारतेच्या उतरत्या क्रमाने दाखवते. वेगवेगळ्या भाषांमधील मजकूरांचे नमुने घेऊन अक्षरांची वारंवारता कशी दिसते ते बघा. (अनुवादकाची टिप्पणी: ह्या प्रश्नासाठी लॅटिन लिपी वापरणाऱ्या भाषाच बघा, उदा., इंग्रजी, जर्मन, फ्रेंच, स्पॅनिश, पोर्तुगीझ, इटालियन.) तुमच्या उत्तरांची तुलना पुढील लिंकवरील तक्त्यांशी करा: http://en.wikipedia.org/wiki/Letter_frequencies. उत्तर: http://thinkpython2.com/code/most_frequent.py*

प्रश्न १२.२. *अजून ॲनाग्राम्स!*

१. *एक प्रोग्राम लिहा जो एका फाइलमधून शब्दयादी वाचून (विभाग ९.१ बघा) सर्व ॲनाग्राम्सच्या गटांना प्रिंट करतो.*

   आउटपुट कसे दिसेल ह्याचे उदाहरण खाली आहे:

   ```
   ['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled']
   ['retainers', 'ternaries']
   ['generating', 'greatening']
   ['resmelts', 'smelters', 'termless']
   ```

   टीप: हे करण्याचा एक मार्ग म्हणजे अशी डिक्शनरी बनवणे जी अक्षरांचा समूह त्या अक्षरांपासून बनणाऱ्या शब्दांच्या लिस्टला मॅप करते. प्रश्न असा आहे की तुम्ही असा समूह डिक्शनरीमधील key *म्हणून वापरण्यासाठी त्याला कसे व्यक्त कराल?*

२. *मागचा प्रोग्राम सुधारून त्याला असे ॲनाग्राम्सच्या लिस्ट्स अशा प्रिंट करायला लावा की सर्वांत मोठी लिस्ट पहिले प्रिंट होईल, मग दुसरी सर्वांत मोठी लिस्ट, इ.*

३. *स्क्रॅबलमध्ये (Scrabble, एक बैठा खेळ) 'bingo' तेव्हा होतो जेव्हा तुम्ही तुमच्याकडे असलेल्या सर्व सात टाइल्स (tiles, प्रत्येक टाइलवर एक इंग्रजी अक्षर असते) आणि बोर्डवर असलेले एक अक्षर असे एकत्र वापरून एक आठ अक्षरी शब्द बनवता. अशी कोणती आठ अक्षरे आहेत जी वापरून सर्वांत जास्ती प्रकारे* bingos *होऊ शकेल?*

   उत्तर: http://thinkpython2.com/code/anagram_sets.py.

प्रश्न १२.३. *दोन शब्द 'metathesis' जोडी बनतात जर तुम्ही दोन अक्षरांची अदलाबदल (swap, स्वाप) करून एकाचे दुसऱ्यात रुपांतर करू शकता; उदा., 'converse' आणि 'conserve'. डिक्शनरीतील सर्व* metathesis *जोड्या शोधा. टीप: शब्दांच्या सर्व जोड्या तपासू नका, आणि शक्य असतील असे सर्व स्वाप्स तपासू नका. उत्तर: http://thinkpython2.com/code/metathesis.py. आभार: हा प्रश्न पुढील लिंक वरील एका उदाहरणावर आधारित आहे http://puzzlers.org.*

प्रश्न १२.४. Car Talk *मधील अजून एक कोडे (http://www.cartalk.com/content/puzzlers):*

> *असा सर्वांत मोठा इंग्रजी शब्द कोणता आहे ज्यातून तुम्ही एकेक करून त्याची अक्षरे काढली तरी प्रत्येक वेळेला तुम्हाला एक अर्थपूर्ण इंग्रजी शब्द मिळेल?*

अक्षरे कोणत्याही बाजूने किंवा मधून काढू शकतो, पण उरलेल्या अक्षरांची पुनर्रचना नाही करू शकत. प्रत्येक वेळी तुम्ही एक अक्षर काढले की तुम्हाला एक (अर्थपूर्ण) इंग्रजी शब्द मिळतो. जर तुम्ही हे केले तर अखेरीस तुम्हाला एक अक्षर मिळेल आणि ते अक्षरसुद्धा एक इंग्रजी शब्द असेल—जो एका शब्दकोशात मिळेल. मला हे जाणून घ्यायचे आहे की असा सर्वांत मोठा शब्द कोणता आहे आणि त्यात किती अक्षरे आहेत?

मी तुम्हाला एक लहान आणि साधे उदाहरण देतो: Sprite. ठीके? तुम्ही sprite पासून सुरुवात करता, मग तुम्ही एक अक्षर काढता, शब्दाच्या आतल्या भागातून, r काढून टाका, आणि आपल्याकडे spite शब्द उरतो, मग आपण शेवटून e काढतो, आपल्याकडे spit उरतो, आपण s काढतो, आपल्याकडे pit, it, आणि I राहते.

अशाप्रकारे संकुचित करता येऊ शकणारे सर्व शब्द शोधणारा प्रोग्राम लिहा आणि मग असा सर्वांत मोठा शब्द शोधा.

हा प्रश्न अन्य प्रश्नांपेक्षा अवघड आहे, तर ह्या काही टीपा:

१. *एक शब्द घेऊन त्यातील एक अक्षर काढून तयार करता येणाऱ्या सर्व शब्दांची लिस्ट शोधणारे फंक्शन फायदेशीर ठरू शकते. हे शब्द त्या शब्दाचे 'children' आहेत.*

२. *रिकर्सिव्हली (recursively), एक शब्द reducible तेव्हा असतो जेव्हा त्याचे कोणतेही 'child' हे reducible असते. बेस-केस (base case) म्हणून तुम्ही एम्प्टी स्ट्रिंग (empty string) reducible आहे असे मानू शकता.*

३. *दिलेल्या शब्दयादीमध्ये म्हणजे* `words.txt` *मध्ये एकअक्षरी शब्द नाहीयेत. तर तुम्ही 'I', 'a', आणि एम्प्टी स्ट्रिंग घालू शकता.*

४. *तुमच्या प्रोग्रामची कार्यक्षमता वाढवण्यासाठी (संथपणे न चालण्यासाठी) तुम्ही जे शब्द reducible आहेत त्यांचे 'मेमो'करण (memoize) केल्याचा फायदा होईल.*

उत्तर: http://thinkpython2.com/code/reducible.py

प्रकरण १३

केस स्टडी: डेटा स्ट्रक्चर निवडणे (Case study: data structure selection)

या टप्प्यावर तुम्ही पायथॉनच्या केंद्रस्थानी असलेली डेटा स्ट्रक्चर्स शिकली, आणि तुम्ही ती डेटा स्ट्रक्चर्स वापरणारे काही अल्गोरिदम्स पाहिले. जर तुम्हाला अल्गोरिदम्स विषयी जास्त जाणून घ्यायचे असेल तर प्रकरण २ वाचण्याची ही चांगली वेळ आहे. पण इथून पुढे जात राहण्यासाठी तुम्हाला ते प्रकरण वाचण्याची गरज नाही; तुम्ही ते तुमची इच्छा होईल तेव्हा वाचू शकता.

ह्या प्रकरणात आपण एक केस स्टडी बघणार आहोत ज्यात अनेक प्रश्न आहेत. ते सोडवताना तुम्हाला डेटा स्ट्रक्चर्स कसे निवडायचे ह्यावर विचार करावा लागेल आणि त्यांना वापरायचा सराव होईल.

१३.१ शब्दांच्या वारंवारतेचे विश्लेषण (Word frequency analysis)

नेहमीप्रमाणे, उत्तरे बघण्याआधी तुम्ही प्रश्न सोडवायचा निदान प्रयत्न तरी केला पाहिजे.

प्रश्न १३.१. *असा एक प्रोग्राम लिहा जो एक फाइल वाचून प्रत्येक ओळीचे शब्दांत तुकडे करतो, त्या शब्दांतून सर्व प्रकारची स्पेस (whitespace) आणि विरामचिन्हे काढून टाकतो, आणि त्याचे (त्यातील प्रत्येक अक्षराचे) स्मॉलमध्ये (lowercase) रुपांतर करतो.*

टीप: `string` *मोड्युल एक* `whitespace` *नावाची स्ट्रिंग पुरवते ज्यात सर्व प्रकारची स्पेस म्हणजे स्पेस (space), टॅब (tab), न्यूलाइन (newline), इ. असते, आणि* `punctuation` *नावाची स्ट्रिंग पुरवते ज्यात सर्व विरामचिन्हे असतात. चला पायथॉन शिव्या देतो का ते बघू:*

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

आणखी, तुम्हाला `strip`, `replace`, *आणि* `translate` *ह्या स्ट्रिंग मेथड्स फायदेशीर ठरू शकतात.*

प्रश्न १३.२. *गुटेनबर्ग प्रकल्पाच्या (Project Gutenberg) वेबसाइटवरून (http://gutenberg.org) तुमच्या आवडीचे कॉपीराइटच्या बाहेर असलेले (उदा., पब्लिक डोमेन, public domain, मध्ये असलेले) एखादे पुस्तक साध्या टेक्स्ट (plain text) स्वरुपात डाऊनलोड करा. (अनुवादकाचा प्रस्ताव: Alice's Adventures in Wonderland by Lewis Carroll https://www.gutenberg.org/ebooks/11.)*

मागच्या प्रश्नातील तुमचा प्रोग्राम बदलून त्यात तुम्ही डाऊनलोड केलेले पुस्तक वाचा, सुरुवातीची (header) माहिती वगळा आणि इतर शब्दांवर आधीसारखीच प्रक्रिया करा.

नंतर प्रोग्राम बदलून त्यात पुस्तकातील शब्दांची एकूण संख्या आणि प्रत्येक शब्द किती वेळा आला आहे ते शोधा.

त्या पुस्तकात वेगवेगळे असे किती शब्द वापरले आहेत? हे प्रिंट करा. वेगवेगळ्या कालखंडातील वेगवेगळ्या लेखकांची वेगवेगळी पुस्तके घेऊन त्यांची तुलना करा. कोणते लेखक विस्तृत शब्दसंग्रह वापरतात?

प्रश्न १३.३. *मागच्या प्रश्नातील प्रोग्राम बदलून त्यात २० सर्वांत जास्ती वापरलेले शब्द प्रिंट करा.*

प्रश्न १३.४. *मागचा प्रोग्राम बदलून त्यात एक शब्दयादी वाचा (विभाग ९.१ बघा), आणि त्या शब्दयादीत नसलेले पुस्तकातील सर्व शब्द प्रिंट करा. त्यांपैकी किती शब्द टाइपिंगच्या चुका आहेत? त्यांपैकी किती शब्द हे कॉमन असून ते शब्दयादीत असायला हवे होते? आणि त्यांपैकी किती शब्द अतिशय दुर्मिळ आहेत?*

१३.२ रँडम संख्या (Random numbers)

सारखेच इनपुट दिल्यावर बहुतांश प्रोग्राम्स प्रत्येकवेळी सारखेच आउटपुट देतात, म्हणून अशा प्रोग्राम्सना **डिटर्मिनिस्टिक** (**deterministic**) म्हणतात[१]. डिटर्मिनिझम (determinism) सहसा चांगले मानले जाते कारण आपली अशी अपेक्षा असते की सारख्याच गणनानंतर सारखेच उत्तर यायला हवे. पण कधीकधी कॉंप्युटरकडून अनिश्चितताही अपेक्षिली जाते. खेळ (games) हे त्याचे साधे उदाहरण आहे, पण अजून आहेत. (आपण अनिश्चिततेसाठी रँडम, random, हा शब्द वापरणार आहोत.)

एखाद्या प्रोग्रामला डिटर्मिनिस्टिकच्या पूर्ण विरुद्ध (अनिश्चित, रँडम) बनवणे खूप अवघड आहे, पण तसा भास निर्माण करणे शक्य आहे. तसे करण्याचा एक मार्ग म्हणजे **सूडो-रँडम** (**pseudorandom**) संख्यांची निर्मिती करणारा अल्गोरिदम. सूडो-रँडम संख्या ह्या पूर्णपणे रँडम नसतात कारण त्या डिटर्मिनिस्टिक पद्धतीनचे निर्माण केलेल्या असतात, पण आपल्याला किंवा बहुतांश प्रोग्राम्सना फक्त त्यांच्याकडे बघून त्या खरेच रँडम आहेत का सूडो-रँडम हे सांगणे अशक्य असते.

पायथॉनमध्ये, `random` मोड्युल सूडो-रँडम संख्यांची निर्माण करणारे फंक्शन पुरवते (इथून पुढे आपण त्यांना सूडो-रँडमच्या ऐवजी 'रँडम' असेच म्हणणार आहोत).

`random` फंक्शन 0.0 आणि 1.0 मधला (0.0 धरून आणि 1.0 सोडून) एक रँडम फ्लोट (float) रिटर्न करते. ह्या फंक्शनच्या प्रत्येक कॉलनंतर तुम्हाला एका मोठ्या मालिकेतील पुढची संख्या मिळते. एक नमुना बघण्यासाठी खालील लूप रन करा:

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

`randint` हे फंक्शन `low` आणि `high` हे दोन परॅमीटर्स घेऊन त्यांच्यामधला (दोन्ही `low` आणि `high` धरून) एक रँडम इंटिजर रिटर्न करते.

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

एका सीक्वेन्समधील रँडम एलेमेंट निवडण्यासाठी तुम्ही `choice` हे फंक्शन वापरू शकता:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
```

[१] अनुवादकाची टिप्पणी: मराठीत determinism म्हणजे निर्धारवाद. शब्द थोडा जड वाटतो, पण त्याचा अर्थ हा की भविष्यात होणाऱ्या सर्व घटना सध्याच्या परिस्थितीवरूनच पूर्णपणे ठरतात अशी विचारसरणी. प्रोग्रामिंगच्या संदर्भात हाच अर्थ आहे. जर फक्त इनपुटवरच प्रोग्रामचे आउटपुट अवलंबून असेल (आणि इतर कशावरही नाही), तर त्या प्रोग्रामला डिटर्मिनिस्टिक (deterministic) म्हणतात.

```
2
>>> random.choice(t)
3
```

हे `random` मोड्युल गाउसिअन (Gaussian), एक्स्पनन्शिअल (exponential), गॅमा (gamma) आणि इतर सतत संभाव्यता वितरणांतून (continuous probability distributions) रँडम संख्यांची निर्मिती करणारी फंक्शन्स पुरवते.

प्रश्न १३.५. *विभाग ११.२ मध्ये आपण हिस्टोग्राम (histogram) म्हणजे काय हे पाहिले होते. एक हिस्टोग्राम घेऊन त्याप्रमाणे एक रँडम व्हॅल्यू निवडून, म्हणजे त्या वारंवारतेच्या प्रमाणात असलेल्या संभाव्यतेने (probability in proportion to the frequency) निवडून, रिटर्न करणारे* `choose_from_hist` *नावाचे फंक्शन लिहा. उदा., खालील हिस्टोग्रामसाठी:*

```
>>> t = ['a', 'a', 'b']
>>> hist = histogram(t)
>>> hist
{'a': 2, 'b': 1}
```

तुमच्या फंक्शनने `'a'` *ही व्हॅल्यू* $2/3$ *संभाव्यतेने रिटर्न केली पाहिजे, आणि* `'b'` *ही* $1/3$ *संभाव्यतेने.*

१३.३ शब्दांचा हिस्टोग्राम (Word histogram)

पुढे जाण्याआधी तुम्ही आधीचे प्रश्न सोडवायचा प्रयत्न केलेला असला पाहिजे. तुम्ही उत्तर पुढील लिंकवरून डाऊनलोड करू शकता: http://thinkpython2.com/code/analyze_book1.py. तुम्हाला पुढील फाइलही लागेल: http://thinkpython2.com/code/emma.txt.

खालील प्रोग्राम एक फाइल वाचून त्यातील शब्दांचा हिस्टोग्राम बनवतो:

```
import string

def process_file(filename):
    hist = dict()
    fp = open(filename)
    for line in fp:
        process_line(line, hist)
    return hist

def process_line(line, hist):
    line = line.replace('-', ' ')

    for word in line.split():
        word = word.strip(string.punctuation + string.whitespace)
        word = word.lower()
        hist[word] = hist.get(word, 0) + 1

hist = process_file('emma.txt')
```

हा प्रोग्राम `emma.txt` वाचतो, ज्यात जेन ऑस्टेन (Jane Austen) चे *Emma* हे पुस्तक आहे.

`process_file` फाइलच्या ओळींवरून लूप करून त्यांना एकेक करून `process_line` ला पाठवते. `hist` हिस्टोग्रामचा वापर अक्युम्युलेटर (accumulator) म्हणून केला आहे.

`process_line` हे `split` ही स्ट्रिंग मेथड वापरून '-' (hyphen, dash, जोडचिन्ह) ला स्पेसने बदलते आणि `split` वापरून ओळीचे स्ट्रिंग्सच्या लिस्टमध्ये तुकडे करते. नंतर ते ती शब्दांची लिस्ट ट्रव्हर्स करून `strip` आणि

`lower` वापरून त्यांतील विरामचिन्हे काढते आणि त्यांतील अक्षरांचे स्मॉल (lowercase) मध्ये रुपांतर करते. ('त्यांचे ''रुपांतर'' करते' ही म्हणण्याची पद्धत झाली; लक्षात ठेवा की स्ट्रिंग इम्युटबल असते, तर `strip` आणि `lower` सारख्या मेथड्स नवीन स्ट्रिंग्स रिटर्न करतात.)

अखेरीस, `process_line` हिस्टोग्रामला नवीन आयटम बनवून किंवा त्यात असलेला वाढवून योग्यपणे अपडेट करते.

फाइलमधील एकूण शब्दांची संख्या शोधण्यासाठी आपण हिस्टोग्राममधील वारंवारतांची (frequencies) बेरीज करू शकतो:

```
def total_words(hist):
    return sum(hist.values())
```

वेगवेगळे असे किती शब्द वापरले आहेत ते शोधण्यासाठी तुम्हाला डिक्शनरीमध्ये किती आयटम्स आहेत ते बघावे लागेल:

```
def different_words(hist):
    return len(hist)
```

खालील कोड उत्तरे प्रिंट करतो:

```
print('Total number of words:', total_words(hist))
print('Number of different words:', different_words(hist))
```

आणि आता उत्तरे:

```
Total number of words: 161080
Number of different words: 7214
```

१३.४ सर्वांत कॉमन शब्द (Most common words)

सर्वांत कॉमन शब्द शोधण्यासाठी आपण टपल्सची एक लिस्ट बनवू शकतो, ज्यात प्रत्येक टपलमध्ये एक शब्द आणि त्याची वारंवारता असेल, आणि तिला सॉर्ट करू शकतो.

खालील फंक्शन एक हिस्टोग्राम घेऊन शब्द-वारंवारता टपल्सची लिस्ट रिटर्न करते:

```
def most_common(hist):
    t = []
    for key, value in hist.items():
        t.append((value, key))

    t.sort(reverse=True)
    return t
```

प्रत्येक टपलमध्ये वारंवारता पहिले येते, म्हणजे नंतरची लिस्ट वारंवारतेने सॉर्ट केली जाईल. खालील लूप १० सर्वांत कॉमन शब्द प्रिंट करतो:

```
t = most_common(hist)
print('The most common words are:')
for freq, word in t[:10]:
    print(word, freq, sep='\t')
```

कीवर्ड अर्ग्युमेंट (keyword argument) `sep` वापरून आपल्याला `print` ला सांगता येते की स्पेसऐवजी टॅब कॅरेक्टर 'हद्द' ('separator') म्हणून वापर, ज्यामुळे दुसरा रकान्यात शब्द एकाखाली एक असे दिसतील. *Emma* वरील उत्तरे खालीलप्रमाणे:

```
The most common words are:
to      5242
the     5205
```

```
and     4897
of      4295
i       3191
a       3130
it      2529
her     2483
was     2400
she     2364
```

हा कोड `sort` फंक्शनचा `key` परॅमीटर वापरून अजून सोप्या पद्धतीने लिहिता येऊ शकतो. कसे ते जर तुम्हाला जाणून घ्यायचे असेल तर तुम्ही पुढील लिंकवर त्याबद्दल वाचू शकता: https://wiki.python.org/moin/HowTo/Sorting.

१३.५ ऑप्शनल परॅमीटर (Optional parameters, पर्यायी परॅमीटर्स)

आपण ऑप्शनल अर्ग्युमेंट्स घेणारी बिल्ट-इन फंक्शन्स बघितलीत. प्रोग्रामरसुद्धा ऑप्शनल अर्ग्युमेंट्स घेणारी फंक्शन्स लिहू शकते. उदा., खालील फंक्शन हिस्टोग्राममधील सर्वांत कॉमन शब्द प्रिंट करते:

```
def print_most_common(hist, num=10):
    t = most_common(hist)
    print('The most common words are:')
    for freq, word in t[:num]:
        print(word, freq, sep='\t')
```

पहिला परॅमीटर अनिवार्य आहे, पण दुसरा पर्यायी आहे; `num` ची **डीफॉल्ट व्हॅल्यू** (**default value**[३]) 10 आहे.

जर तुम्ही एकच अर्ग्युमेंट पाठवले:

```
print_most_common(hist)
```

तर `num` ला डीफॉल्ट व्हॅल्यू मिळते. जर तुम्ही दोन अर्ग्युमेंट्स पाठवलीत:

```
print_most_common(hist, 20)
```

तर `num` ला दुसरे अर्ग्युमेंट व्हॅल्यू म्हणून मिळते. दुसऱ्या शब्दांत, ऑप्शनल अर्ग्युमेंट डीफॉल्ट व्हॅल्यूपेक्षा **वरचढ** ठरते (the optional argument **overrides** the default value) .

जर एका फंक्शनमध्ये अनिवार्य आणि पर्यायी अशी दोन्ही प्रकारची परॅमीटर्स असतील तर सर्व अनिवार्य परॅमीटर्स आधी आली पाहिजेत, त्यानंतर पर्यायी परॅमीटर्स.

१३.६ डिक्शनरी वजाबाकी (Dictionary subtraction)

पुस्तकात असलेले पण `words.txt` ह्या शब्दयादीत नसलेले शब्द शोधणे हा प्रॉब्लेम संच वजाबाकी (set subtraction) शोधणे हाच आहे हे तुम्ही कदाचित ओळखले असेल; म्हणजे, आपल्याला एका संचातील (पुस्तकातील) असे सर्व शब्द पाहिजे आहेत जे दुसऱ्या संचात (शब्दयादीत) नाहीयेत.

`subtract` हे `d1` आणि `d2` ह्या दोन डिक्शनरीझ घेऊन एक नवीन डिक्शनरी रिटर्न करते ज्यात `d1` मधील त्या सर्व keys असतील ज्या `d2` मध्ये नाहीत. आपल्याला त्यांतील व्हॅल्यूझशी काही देणेघेणे नसल्यामुळे आपण त्या `None` ठेवतो.

[३](अनुवादकाची टिप्पणी: आधी सांगितल्याप्रमाणे, ह्याठिकाणी 'default' चा अर्थ म्हणजे दुसरा पर्याय नसल्यामुळे निवडलेला प्रमाण (standard) पर्याय.)

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

पुस्तकात असलेले आणि `words.txt` मध्ये नसलेले शब्द शोधण्यासाठी आपण `process_file` ने `words.txt` चा हिस्टोग्राम बनवू शकतो, आणि नंतर वजाबाकी करू शकतो:

```
words = process_file('words.txt')
diff = subtract(hist, words)

print("Words in the book that aren't in the word list:")
for word in diff:
    print(word, end=' ')
```

खाली *Emma* हे पुस्तक वापरून शोधलेली काही उत्तरे आहेत:

```
Words in the book that aren't in the word list:
rencontre jane's blanche woodhouses disingenuousness
friend's venice apartment ...
```

ह्यांतली काही नावे आहेत आणि काही स्वामित्वदर्शक (possessive) शब्द. काही इतर शब्द, जसे 'rencontre', आजकाल वापरले जात नाहीत. पण काही हे कॉमन शब्द असून यादीमध्ये असायला हवे!

प्रश्न १३.६. *पायथॉनमध्ये* `set` *(सेट, संच) नावाचे एक डेटा स्ट्रक्चर आहे जे संचांवरील अनेक कॉमन प्रक्रिया पुरवते. तुम्ही त्याबद्दल विभाग १९.५ मध्ये वाचू शकता किंवा पुढील लिंकवर डॉक्युमेंटेशन वाचू शकता: http://docs.python.org/3/library/stdtypes.html#types-set.*

संचांची वजाबाकी वापरून पुस्तकात असलेले पण शब्दयादीत नसलेले शब्द शोधणारा प्रोग्राम लिहा. उत्तर: http://thinkpython2.com/code/analyze_book2.py.

१३.७ रँडम शब्द (Random words)

हिस्टोग्राममधून रँडम शब्द निवडण्यासाठी प्रत्येक शब्दाच्या त्याच्या वारंवारतेएवढ्या प्रती (copies) एका लिस्टमध्ये घालून त्यातून एक शब्द निवडणे हा सर्वांत सोपा अल्गोरिदम आहे:

```
def random_word(h):
    t = []
    for word, freq in h.items():
        t.extend([word] * freq)

    return random.choice(t)
```

`[word] * freq` हे एक्स्प्रेशन `word` च्या `freq` प्रती (copies) असणारी लिस्ट बनवते; `extend` मेथड `append` सारखी असून फरक हा की तिचे अर्ग्युमेंट एक सीक्वेन्स असते.

हा अल्गोरिदम चालतो पण तो खूप कार्यक्षम नाहीये; प्रत्येक वेळी जेव्हा तुम्ही एक रँडम शब्द निवडता तेव्हा तो पूर्ण लिस्ट परत बनवतो, जी मूळ पुस्तकाइतकीच मोठी आहे. सुधारण्याचा साधा मार्ग हा की लिस्ट एकदाच बनवून अनेकदा शब्द निवडणे, पण ती लिस्ट तरीही मोठीच असेल.

पर्याय आहे:

१. `keys` वापरून पुस्तकातील शब्दांची यादी बनवा.

२. शब्दांच्या वारंवारतांची क्युम्युलेटिव्ह बेरीज (cumulative sum) असणारी लिस्ट बनवा (प्रश्न १०.२ बघा). ह्या लिस्टमधील शेवटचा एलेमेंट म्हणजेच पुस्तकातील शब्दांची एकूण संख्या n.

३. 1 आणि n मधली एक रँडम संख्या निवडा. बायनरी सर्च (binary/bisection search, प्रश्न १०.१० बघा) वापरून क्युम्युलेटिव्ह बेरजांच्या लिस्टमध्ये ही रँडम संख्या कुठे जाईल ती इंडेक्स शोधा.

४. ती इंडेक्स वापरून संबंधित शब्द पायरी १ मध्ये बनवलेल्या लिस्टमध्ये शोधा.

प्रश्न १३.७. *वरील अल्गोरिदम वापरून पुस्तकातून एक रँडम शब्द निवडणारा प्रोग्राम लिहा. उत्तर: http://thinkpython2.com/code/analyze_book3.py.*

१३.८ मार्कोव्ह विश्लेषण (Markov analysis)

जर तुम्ही पुस्तकातून रँडम शब्द उचलले तर तुम्हाला त्यातल्या शब्दसंग्रहाचा अंदाज येईल, पण तुम्हाला वाक्य मिळणार नाही:

```
this the small regard harriet which knightley's it most things
```

अशा रँडम शब्दांच्या मालिकेचा क्वचितच अर्थ लागू शकेल कारण आजूबाजूच्या शब्दांमध्ये काहीच संबंध नसतो. उदा., एका खऱ्या वाक्यात तुम्हाला 'the' नंतर एक विशेषण किंवा नाम मिळेल, आणि शक्यतो क्रियापद किंवा क्रियाविशेषण नाही.

ह्याप्रकारच्या संबंधांचा वेध घेण्यासाठी मार्कोव्ह विश्लेषण वापरता येते, जे दिलेल्या शब्दांच्या मालिकेनंतर पुढचा शब्द कोणता असेल ह्याची संभाव्यता (probability) व्यक्त करते. उदा., *Eric, the Half a Bee* हे गीत असे सुरू होते:

> Half a bee, philosophically,
> Must, ipso facto, half not be.
> But half the bee has got to be
> Vis a vis, its entity. D'you see?
>
> But can a bee be said to be
> Or not to be an entire bee
> When half the bee is not a bee
> Due to some ancient injury?

ह्यात, 'half the' च्या नंतर नेहमी 'bee' हा शब्द येतो, पण 'the bee' नंतर 'has' किंवा 'is' येऊ शकतात.

मार्कोव्ह विश्लेषण आपल्याला प्रत्येक शब्दमालिका (उदा., 'half the' आणि 'the bee') त्यानंतर येऊ शकणाऱ्या सर्व शब्दांच्या समूहाला (उदा., 'has' आणि 'is') मॅप करणारी मॅपिंग (mapping) देते. आधी येणाऱ्या अशा शब्दमालिकेला आपण इंग्रजीत prefix म्हणणार आहोत आणि नंतर येऊ शकणाऱ्या एका शब्दाला suffix (आधी येणारे prefix, नंतर येणारा suffix).

ही मॅपिंग दिल्यावर, आपण कोणत्याही शब्दमालिकेपासून सुरुवात करून पुढे येणाऱ्या शब्दसमूहातून एक रँडम शब्द निवडून एक रँडम मजकूर बनवू शकतो. नंतर तुम्ही पूर्वीची शब्दमालिका आणि नवीन शब्द जोडून नवीन शब्दमालिका मिळवू शकता आणि मागच्या प्रक्रियेची पुनरावृत्ती करू शकता.

उदा., जर तुम्ही 'Half a' ने सुरुवात केली, तर पुढचा शब्द 'bee' असला पाहिजे, कारण ही शब्दमालिका फक्त एकदाच येते. पुढची शब्दमालिका 'a bee' आहे, तर त्यानंतरचा शब्द 'philosophically', 'be,' किंवा 'due' असू शकतो.

ह्या उदाहरणात शब्दमालिकेत नेहमी दोनच शब्द होते, पण तुम्ही कितीही शब्द घेऊन मार्कोव्ह विश्लेषण करू शकता.

प्रश्न १३.८. *मार्कोव्ह विश्लेषण:*

१. *एका फाइलमधील टेक्स्ट वाचून मार्कोव्ह विश्लेषण करणारा प्रोग्राम लिहा. ह्यात शब्दमालिका (नंतर येऊ शकणाऱ्या अशा) शब्दसमूहाला मॅप करणारी डिक्शनरी तयार करा. शब्दसमूह हा एक लिस्ट, टपल, किंवा डिक्शनरी असू शकतो; तुम्ही तुमच्या सोयीनुसार निवडा. तुम्ही तुमचा प्रोग्राम दोन शब्द असणाऱ्या शब्दमालिकांसाठी टेस्ट करू शकता, पण प्रोग्राम असा लिहा की त्याहून जास्त शब्द असलेल्या मालिकासुद्धा सहज तपासता येतील.*

२. *मागच्या प्रोग्राममध्ये मार्कोव्ह विश्लेषण वापरून रँडम मजकूर निर्माण करणारे फंक्शन लिहा.* Emma *आणि दोन शब्दांच्या मालिका वापरून तयार केलेले उदाहरण खाली आहे:*

> *He was very clever, be it sweetness or be angry, ashamed or only amused, at such a stroke. She had never thought of Hannah till you were never meant for me?" "I cannot make speeches, Emma:" he soon cut it all himself.*

ह्या उदाहरणात विरामचिन्हे शब्दांना जोडलेलीच राहू दिली आहेत. ह्याची संरचना जवळजवळ बरोबर आहे पण पूर्णपणे नाही. त्याचा अर्थ आपण जवळजवळ लावू शकतो, पण पूर्णपणे नाही.

जर तुम्ही दोनहून अधिक शब्दांच्या मालिका वापरल्या तर काय होईल? रँडम मजकूर जास्ती अर्थपूर्ण होईल का?

३. *तुमचा प्रोग्राम एकदाचा तयार झाला तर तो वापरून तुम्हाला एक प्रयोग करता येईल: दोन किंवा अधिक पुस्तकांतील टेक्स्ट तुम्ही एकत्र केले तर तुम्ही निर्माण केलेला रँडम मजकूर त्यांतील वाक्प्रचार आणि शब्दसंग्रह मजेशीरपणे मिसळेल.*

आभार: ही केस स्टडी पुढील पुस्तकातील एका उदाहरणावर आधारित आहे: Kernighan and Pike, The Practice of Programming, *Addison-Wesley, 1999.*

पुढे जाण्याआधी तुम्ही हा प्रश्न सोडवायचा प्रयत्न केलेला असला पाहिजे; नंतर तुम्ही पुढील लिंकवरून उत्तर डाऊनलोड करू शकता: http://thinkpython2.com/code/markov.py. तुम्हाला हे ही लागेल: http://thinkpython2.com/code/emma.txt.

१३.९ डेटा स्ट्रक्चर्स (Data structures)

मार्कोव्ह विश्लेषण वापरून रँडम मजकूर निर्माण करण्यात गंमत आहे, पण ह्या प्रश्नाचा एक हेतूदेखील आहे: डेटा स्ट्रक्चर निवडणे. मागच्या प्रश्नांच्या उत्तरांत तुम्हाला हे ठरवावे लागले:

- शब्दमालिका कशा ठेवायच्या.
- पुढे येऊ शकणाऱ्या शब्दांचा समूह कसा ठेवायचा.
- आणि प्रत्येक शब्दमालिका तिच्या पुढे येऊ शकणाऱ्या शब्दांच्या समूहाला मॅप करणारी मॅपिंग कशी दाखवायची.

शेवटच्यासाठी निर्णय सोपा आहे: keys संबंधित व्हॅल्यूझना मॅप करण्यासाठी डिक्शनरी ही सरळ निवड आहे.

शब्दमालिकांसाठी सरळ पर्याय आहेत: स्ट्रिंग्स, स्ट्रिंग्सची लिस्ट, किंवा स्ट्रिंग्सचे टपल.

पुढील शब्दसमूहांसाठी, एक पर्याय आहे लिस्ट आणि दुसरा हिस्टोग्राम (डिक्शनरी).

निर्णय कसे घ्यायचे? पहिले तुम्ही प्रत्येक डेटा स्ट्रक्चरवर कोणती ऑपरेशन्स करावी लागतील ह्यावर विचार करा. शब्दमालिकांसाठी, आपल्याला सुरुवातीचा शब्द काढून शेवटी नवीन शब्द लावता आला पाहिजे. उदा., जर सध्याची शब्दमालिका 'Half a' असेल आणि पुढचा शब्द 'bee' असेल तर तुम्हाला 'a bee' ही पुढची शब्दमालिका बनवता आली पाहिजे.

तुमचा पहिला विचार लिस्ट असू शकतो कारण लिस्टमधून एलेमेंट्स काढणे आणि घालणे सोपे आहे, पण आपल्याला शब्दमालिका एका डिक्शनरीमध्ये key म्हणून वापरता आली पाहिजे, म्हणजेच आपण लिस्ट नाही वापरू शकत. टपल्समधून एलेमेंट्स काढणे आणि घालणे शक्य नाही पण तुम्ही बेरीज ऑपरेटर वापरून नवीन टपल बनवू शकता:

```
def shift(prefix, word):
    return prefix[1:] + (word,)
```

`shift` हे फंक्शन `prefix` हे शब्दांचे टपल आणि `word` ही स्ट्रिंग घेऊन एक नवीन टपल तयार करते ज्यात `prefix` चे पहिला सोडून सर्व शब्द आणि शेवटी `word` आहे.

शब्दसमूहावर करायचे ऑपरेशन्स म्हणजे नवीन शब्द टाकणे (किंवा असलेल्याची वारंवारता वाढवणे) आणि एक रँडम शब्द निवडणे.

नवीन शब्द टाकणे लिस्ट आणि हिस्टोग्राम दोन्हींसाठी सोपे आहे. लिस्टमधून रँडम एलेमेंट निवडणे सोपे आहे, पण हिस्टोग्राममधून शीघ्रपणे निवडणे अवघड (प्रश्न १३.७ बघा).

आतापर्यंत आपण इंप्लेमेंटेशन करणे किती सोपे आहे ह्याबद्दलच बोललो, पण डेटा स्ट्रक्चर निवडताना बाकी काही बाबींचा विचार करणे देखील महत्त्वाचे आहे. एक म्हणजे **रनिंग टाइम** (**running time**). कधीकधी काही गणितीय सिद्धांतांवर आधारित कारणांमुळे एक डेटा स्ट्रक्चर दुसऱ्यापेक्षा जास्ती वेगवान असू शकते; उदा., आपण आधी बघितले होते की `in` ऑपरेटर डिक्शनरीवर लिस्टपेक्षा जास्ती वेगवान आहे, निदान जेव्हा एलेमेंट्सची संख्या जास्ती असते तेव्हा.

पण सहसा तुम्हाला कोणते डेटा स्ट्रक्चर जास्ती वेगाने चालेल ह्याची पूर्वकल्पना नसते. एक पर्याय असा आहे की दोन्ही इंप्लेमेंट करून कोणता जास्ती चांगला आहे ह्याची चाचणी करणे. ह्याला **बेंचमार्किंग** (**benchmarking**) असे म्हणतात. एक सोयीस्कर पर्याय हा की इंप्लेमेंट करायला सोपे असे डेटा स्ट्रक्चर निवडणे आणि तपासणे की ते आपल्या कामासाठी पुरेसे वेगवान आहे का. जर असेल, तर पुढे काही करायची गरज नाही. नाहीतर `profile` मोड्युल सारखी साधने वापरून तुम्ही प्रोग्राममधील सर्वांत संथ भाग हुडकू शकता.

दुसरी महत्त्वाची बाब म्हणजे जागा (space). उदा., शब्दसमूहासाठी हिस्टोग्राम वापरला तर कमी जागा लागेल कारण प्रत्येक शब्द एकदाच ठेवावा लागेल जरी तो कितीही वेळा टेक्स्टमध्ये असेल तरी. कधीकधी जागा वाचवण्याने सुद्धा प्रोग्राम वेगाने चालू शकतो; दुसऱ्या टोकाला, जर तुमच्या प्रोग्रामसाठी मेमरी (memory) उरली नसेल तर तो चालणारच नाही. पण बहुतांश कामांसाठी स्पेस ही रनिंग टाइमनंतरची दुय्यम बाब आहे.

एक शेवटचा मुद्दा: ह्या चर्चेत असे सुचवले गेले आहे की आपण विश्लेषण आणि निर्माण ह्या दोन्ही टप्प्यांसाठी एकच डेटा स्ट्रक्चर वापरले पाहिजे. पण हे वेगळे टप्पे असल्यामुळे विश्लेषणासाठी एक डेटा स्ट्रक्चर वापरून नंतर निर्माणासाठी दुसऱ्या डेटा स्ट्रक्चरमध्ये रुपांतर करणे असेही शक्य आहे. जर निर्माणासाठी झालेली वेळेची बचत रुपांतरासाठी लागलेल्या वेळेपेक्षा जास्त असेल तर ह्याचा आपल्याला निव्वळ फायदा होईल.

१३.१० डीबगिंग (Debugging)

जर तुम्ही एखादा प्रोग्राम डीबग करत असाल, आणि विशेषतः जर तुम्ही एखाद्या चिवट बग-वर काम करत असाल तर तुम्ही खालील पाच गोष्टी करू शकता:

वाचणे (Reading): कोड-चे परीक्षण करा, स्वतःशी पुन्हा वाचा, आणि हे तपासा की तो तेच म्हणतोय जे तुम्हाला म्हणायचे आहे.

रनिंग (Running): विविध प्रयोग करून रन करून बघा. सहसा जर तुम्ही प्रोग्राममध्ये बरोबर गोष्ट बरोबर जागी प्रिंट केली तर चूक समोर येते, पण कधीकधी तुम्हाला स्कॅफोल्डिंग बांधावी लागते.

मनन करणे (Ruminating): विचार करण्यासाठी थोडा वेळ काढा! एरर कोणत्या प्रकारचा आहे: सिंटॅक्स, रनटाइम, का सिमँटिक? एरर मेसेजवरून किंवा प्रोग्रामच्या आउटपुटवरून तुम्हाला काय माहिती मिळवता येऊ शकते? कोणत्या प्रकारच्या चुका तुम्हाला दिसणारा एरर घडवू शकतात? तुम्ही नुकताच काय बदल केला होता ज्यामुळे हा एरर आला आहे?

रबर-डकिंग (Rubberducking): जर तुम्ही कोणाला त्रासाचे वर्णन केले तर कधीकधी तुम्हाला प्रश्न विचारायच्या आधीच उत्तर मिळते. सहसा, तुम्हाला दुसऱ्या कोणाची गरज नसते; तुम्ही रबराच्या बदकाशीच बोलू शकता. आणि हीच **रबर-डक डीबगिंग** (**rubber duck debugging**) ह्या सुप्रसिद्ध तंत्राची गोष्ट आहे. खरेच, हा विनोद नाहीये; पुढील लिंक बघा: https://en.wikipedia.org/wiki/Rubber_duck_debugging.

माघार घेणे (Retreating): कधीतरी, माघार घेणे हेच सर्वोत्तम असते: नुकतेच केलेले बदल मागे घेणे, आणि हे तोपर्यंत करणे जोपर्यंत तुम्हाला समजणारा आणि चालणारा प्रोग्राम मिळत नाही. मग तुम्ही पुनर्बांधणी सुरू करू शकता.

नवशिके प्रोग्रामर्स कधीकधी वरीलपैकी एकाच तंत्रावर अडकून राहतात आणि अन्य पर्याय विसरतात. प्रत्येक तंत्राची स्वतःची अशी एक उणीव आहे.

उदा., जर लिहिण्यात काही चूक झाली असेल तर कोड वाचण्याचा काही उपयोग आहे, पण संकल्पनात्मक गैरसमज (conceptual misunderstanding) असेल तर ह्याचा उपयोग नाही. जर तुमचा प्रोग्राम काय करतो हेच तुम्हाला समजत नसेल, तर तुम्ही तो ४०० वेळा वाचला तरी तुम्हाला एरर दिसणार नाही कारण तो एरर तुमच्या डोक्यात आहे.

विविध प्रयोग रन करण्याचा फायदा विशेषतः लहान आणि साध्या टेस्ट्स रन करता आल्या तर होऊ शकतो. पण जर तुम्ही विचार *न* करता किंवा तुमचा कोड *न* वाचता प्रयोग करत बसलात तर तुम्ही 'रँडम वॉक प्रोग्रामिंग' ('random walk programming') ह्या साच्यात पडाल; म्हणजेच जोपर्यंत प्रोग्राम चालत नाही तोपर्यंत काहीतरी अहेतुक/असंबद्ध बदल करत बसायचे. हे सांगायची गरज नाही की रँडम वॉक प्रोग्रामिंगला प्रचंड वेळ लागू शकतो.

तुम्हाला विचार करायला वेळ दिला पाहिजे. डीबगिंग हे प्रायोगिक विज्ञानसारखे आहे. एरर काय असावा ह्याबद्दल तुमचे कमीतकमी एक तरी हायपॉथसिस (गृहीतक) असले पाहिजे. जर दोन किंवा अधिक शक्यता असतील तर त्यांच्यापैकी एखादे काढून टाकण्याच्या दिशेने काही टेस्ट्सचा विचार करा.

पण जर खूप एरर्स असतील किंवा तुम्ही दुरूस्त करत असणारा कोड मोठा आणि गुंतागुंतीचा असेल तर डीबगिंगची उत्तमोत्तम तंत्रेसुद्धा अपयशी ठरतील. कधीकधी सर्वोत्तम पर्याय हा असतो की माघार घेऊन, नुकतेच केलेले बदल मागे घेणे, आणि हे तोपर्यंत करणे जोपर्यंत तुम्हाला समजणारा आणि चालणारा प्रोग्राम मिळत नाही.

नवशिके प्रोग्रामर्स सहसा माघार घेण्यास अनिच्छुक असतात कारण त्यांना कोड-ची एक ओळसुद्धा काढणे सहन होत नाही (जरी ती चुकीची असली तरी). तुम्हाला जर इतकेच वाटत असेल तर काटछाट करण्याआधी तुम्ही तुमचा प्रोग्राम दुसऱ्या फाइलमध्ये कॉपी करू शकता. नंतर मग तुम्ही त्यातील तुकडे एकेक करून परत कॉपी करू शकता.

चिवट बग शोधायला वाचन, रनिंग, मनन, आणि कधीकधी माघार ह्यांची गरज पडते. "Finding a hard bug requires reading, running, ruminating, and sometimes retreating." जर तुम्ही एक तंत्र वापरताना अडकून पडले असाल तर अन्य तंत्रे वापरण्याचा प्रयत्न करा.

१३.११ शब्दार्थ

डिटर्मिनिस्टिक (deterministic): अशा प्रोग्रामशी संबंधित जो सारखेच इनपुट दिल्यावर प्रत्येकवेळा रन केल्यावर सारखीच गोष्ट करतो.

सूडो-रँडम (pseudorandom): संख्यांच्या अशा मालिकेशी संबंधित जी दिसते रँडम पण एका डिटर्मिनिस्टिक प्रोग्रामने निर्माण केलेली असते.

डीफॉल्ट व्हॅल्यू (default value): ऑप्शनल परॅमीटरला दिलेली व्हॅल्यू जेव्हा संबंधित अर्ग्युमेंट पाठवलेले नसते.

ओव्हर्राइड (override): डीफॉल्ट व्हॅल्यूस अर्ग्युमेंटने बदलणे.

बेंचमार्किंग (benchmarking): डेटा स्ट्रक्चर्सचे विविध पर्याय इंप्लेमेंट करून त्यांची इनपुटच्या काही नमुन्यांवर टेस्टिंग करूनएक डेटा स्ट्रक्चर निवडण्याची प्रक्रिया.

रबर डक डीबगिंग (rubber duck debugging): तुमच्यासमोरच्या अडथळ्याचे वर्णन एका निर्जीव वस्तूला, जसे रबराचे बदक, करून प्रोग्राम डीबग करणे. स्पष्ट शब्दांत तुमची समस्या व्यक्त केल्याचा फायदा तुम्हाला ती समस्या सोडवण्यास होऊ शकतो, जरी त्या रबरी बदकाला पायथॉन येत नसले तरी.

१३.१२ प्रश्नसंच (Exercises)

प्रश्न १३.९. *एका शब्दाचा 'क्रमांक' ('rank', रँक) म्हणजे वारंवारतेनुसार सॉर्ट केलेल्या यादीत त्याचे स्थान: सर्वांत कॉमन शब्दाचा क्रमांक १, नंतरच्या सर्वांत कॉमन शब्दाचा क्रमांक २, इ.*

झिफचा नियम (Zipf's law) नैसर्गिक भाषांसंदर्भात शब्दांचे क्रमांक आणि वारंवारता ह्यांमधला संबंध दर्शवतो (http://en.wikipedia.org/wiki/Zipf's_law). विशेषतः, तो असे वर्तवतो की r रँक असलेल्या शब्दाची वारंवारता (frequency) f ही

$$f = cr^{-s}$$

असते, ज्यात s आणि c ह्या संख्या भाषा आणि मजकूरावर अवलंबून असतात. दोन्ही बाजूचा लॉग (log) घेतला तर:

$$\log f = \log c - s \log r \,.$$

म्हणजेच जर तुम्ही $\log f$ विरुद्ध $\log r$ आलेखले तर तुम्हाला चढ (slope) $-s$ आणि आंतरछेद $\log c$ असणारी रेषा मिळेल.

एक प्रोग्राम लिहा जो एका फाइलमधून टेक्स्ट वाचून शब्दांच्या वारंवारता मोजतो आणि वारंवारतेच्या उतरत्या क्रमाने एका ओळीवर एक शब्द त्याच्या $\log f$ आणि $\log r$ ह्यांच्या किमतीसहित प्रिंट करतो. तुमच्या सोयीचा आलेखन प्रोग्राम वापरून ह्याचा आलेख काढा आणि तपासा की तो एक सरळ रेषा आहे की नाही. तुम्ही s ची किंमत शोधू शकता का?

उत्तर: http://thinkpython2.com/code/zipf.py. हे उत्तर रन करण्यासाठी तुम्हाला `matplotlib` *हे आलेखनाचे मोड्युल लागेल. जर तुम्ही Anaconda इन्स्टॉल केले असेल तर त्यात हे मोड्युल आधीपासूनच असते; नाहीतर तुम्हाला ते इन्स्टॉल करावे लागेल.*

प्रकरण १४

फाइल (File)

ह्या प्रकरणात आपण 'दीर्घस्थायी' ('persistent') प्रोग्राम्स बघणार आहोत जे त्यांचा डेटा कायमस्वरूपी संग्रहात ठेवतात, आणि फाइल्स आणि डेटाबेसेस (databases) सारखे वेगवेगळ्या प्रकारचे कायमस्वरूपी माध्यम (permanent storage) कसे वापरायचे हेही बघणार आहोत.

१४.१ दीर्घस्थायीता (Persistence)

आपण पाहिलेले बहुतांश प्रोग्राम्स अल्पजीवी होते, म्हणजे ते थोडा वेळ चालतात, काही आउटपुट निर्माण करतात, पण ते संपल्यावर त्यांचा डेटा निघून जातो. तुम्ही जर परत तो प्रोग्राम रन केलात, तर तो परत शून्यापासून सुरू होतो.

इतर प्रोग्राम्स **दीर्घस्थायी** (**persistent**) असतात: ते बराच वेळ (किंवा कायम) चालतात, ते त्यांच्या डेटा चा काही भाग कायमस्वरूपी माध्यमात (उदा., हार्ड डिस्कमध्ये) ठेवतात, आणि जर ते बंद होऊन परत सुरू झाले तर ज्या परिस्थितीत बंद झाले होते त्याच परिस्थितीतून त्यांचे काम चालू ठेवतात.

दीर्घस्थायी प्रोग्राम्सची काही उदाहरणे म्हणजे ऑपरेटिंग सिस्टम जी जेव्हाही कॉंप्युटर चालू असतो तेव्हा सुरू असते, आणि वेब-सर्व्हर (web server), जे नेटवर्कवरून माहितीच्या मागण्यांची वाट बघत कायम सुरू असते.

टेक्स्ट फाइल्स वाचून आणि लिहून डेटा टिकवून ठेवणे हा दीर्घस्थायीतेचा मार्ग प्रोग्रामसाठी सर्वांत सोपा आहे. आपण फाइल प्रोग्राम्स आधी पाहिलेच आहेत; ह्या प्रकरणात आपण फाइल्समध्ये लिहिणारे प्रोग्राम्स बघणार आहोत.

दीर्घस्थायीतेसाठी अजून एक पर्याय म्हणजे डेटाबेस (database) आहे. ह्या प्रकरणात आपण एक सोपे डेटाबेस आणि `pickle` नावाचे मोड्युल बघणार आहोत ज्याच्या मदतीने प्रोग्रामचा डेटा टिकवून ठेवणे सोपे पडते.

१४.२ वाचणे आणि लिहिणे (Reading and writing)

टेक्स्ट फाइल म्हणजे हार्ड-डिस्क, सीडी (CD), इ. सारख्या कायमस्वरूपी माध्यमात ठेवलेला कॅरेक्टर्सचा सीक्वेन्स होय. विभाग ९.१ मध्ये आपण पायथॉनमध्ये फाइल उघडून वाचायची कशी हे बघितले.

फाइलमध्ये लिहिण्यासाठी ती `'w'` mode[१] ('write' mode) मध्ये उघडावी लागते; हे दुसरे अर्ग्युमेंट असते:

```
>>> fout = open('output.txt', 'w')
```

[१] अनुवादकाची टिप्पणी: 'मोड' हा मराठीत अर्थपूर्ण शब्द असल्यामुळे आपण हा शब्द लॅटिन लिपीमध्येच लिहिणार आहोत.

जर फाइल आधीपासूनच असेल तर तिला write mode मध्ये उघडल्यावर तिच्यातला जुना डेटा निघून जातो आणि नवीन सुरुवात होते, तर काळजी घ्या! जर फाइल नसेल, तर नवीन बनवली जाते.

`open` एक फाइल ऑब्जेक्ट रिटर्न करते ज्यात फाइलवर काही करण्यासाठी मेथड्स असतात; `write` मेथड फाइलमध्ये डेटा लिहिते.

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
24
```

रिटर्न व्हॅल्यू किती कॅरेक्टर्स लिहिले ते दर्शवते. फाइल ऑब्जेक्ट तो सध्या फाइलमध्ये कुठे आहे ह्याची नोंद ठेवतो, म्हणजे जर तुम्ही `write` परत कॉल केले तर नवीन डेटा तो फाइलच्या शेवटी लिहितो.

```
>>> line2 = "the emblem of our land.\n"
>>> fout.write(line2)
24
```

तुमचे फाइलमध्ये लिहून झाल्यावर तुम्ही फाइल बंद केली पाहिजे.

```
>>> fout.close()
```

जर तुम्ही बंद नाही केली तर ती प्रोग्राम संपल्यावर बंद केली जाते.

१४.३ फॉरमॅट ऑपरेटर (Format operator)

`write` चे अर्ग्युमेंट स्ट्रिंग असणे गरजेचे आहे, म्हणून जर आपल्याला इतर व्हॅल्यूझ फाइलमध्ये टाकायच्या असतील तर आपल्याला त्यांचे स्ट्रिंगमध्ये रुपांतर करावे लागेल. ते करण्याचा सर्वांत सोपा मार्ग म्हणजे `str`:

```
>>> x = 52
>>> fout.write(str(x))
```

फॉरमॅट ऑपरेटर (**format operator**) % वापरणे हा एक पर्याय आहे. इंटिजरवर वापरल्यास % हा मोड्युलस (modulus) ऑपरेटर असतो आणि बाकी रिटर्न करतो. पण जर पहिला ऑपरँड (operand) स्ट्रिंग असेल तर % हा फॉरमॅट ऑपरेटर असतो.

पहिल्या ऑपरँडला **फॉरमॅट स्ट्रिंग** (**format string**) म्हणतात, ज्यात एक किंवा अधिक **फॉरमॅट सीक्वेन्सेस** (**format sequences**) असतात जे हे दर्शवतात की दुसऱ्या ऑपरँडचे स्वरूप कसे असले पाहिजे (तो कसा फॉरमॅट केला पाहिजे). उत्तर एक स्ट्रिंग असते.

उदा., फॉरमॅट सीक्वेन्स `'%d'` चा अर्थ हा की दुसरा ऑपरँड दशगुणोत्तरी पूर्ण संख्येच्या (decimal integer) स्वरुपात[२] असला पाहिजे:

```
>>> camels = 42
>>> '%d' % camels
'42'
```

उत्तर `'42'` ही स्ट्रिंग आहे, जी इंटिजर व्हॅल्यू `42` पेक्षा वेगळी आहे.

फॉरमॅट सीक्वेन्स स्ट्रिंगमध्ये कुठेही येऊ शकतो, तर तुम्ही एका वाक्यात एक व्हॅल्यू अशी टाकू शकता:

```
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

जर स्ट्रिंगमध्ये एकाहून अधिक फॉरमॅट सीक्वेन्सेस असतील तर दुसरे अर्ग्युमेंट टपल असले पाहिजे. प्रत्येक फॉरमॅट सीक्वेन्स क्रमाने टपलच्या संबंधित एलेमेंटशी जोडला जातो.

खालील उदाहरण `'%d'` ने इंटिजर, `'%g'` ने फ्लोटिंग-पॉइंट संख्या, आणि `'%s'` ने स्ट्रिंग फॉरमॅट करते:

[२] म्हणजेच ०, १, २, ३, ..., ९ हे आकडे वापरून.

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

टपलमध्ये तितकेच एलेमेंट्स असायला हवे जितके पहिल्या स्ट्रिंगमध्ये फॉरमॅट सीक्वेन्सेस आहेत. अजून, एलेमेंट्सचे टाइप्ससुद्धा फॉरमॅट सीक्वेन्सेसशी जुळले पाहिजेत:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

वर, पहिल्या उदाहरणात, पुरेसे एलेमेंट्स नाहीयेत, आणि दुसऱ्यात एलेमेंट चुकीच्या टाइपचा आहे.

फॉरमॅट ऑपरेटरविषयी अजून माहितीसाठी पुढील लिंक बघा: https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting.

अजून एक जास्ती प्रभावी पर्याय म्हणजे स्ट्रिंगची `format` मेथड जिच्याबद्दल तुम्ही पुढील लिंकवर वाचू शकता:https://docs.python.org/3/library/stdtypes.html#str.format.

१४.४ फाइलनेम आणि पाथ (Filenames and paths)

फाइल्सची रचना **डिरेक्टरीझ** (**directories**, फोल्डर्स, 'folders' सुद्धा म्हणतात) मध्ये केली जाते. चालणाऱ्या प्रत्येक प्रोग्रामची एक 'करंट डिरेक्टरी' ('current directory') असते ज्यात डिरेक्टरीमध्ये त्या प्रोग्रामच्या बहुतांश क्रिया होत असतात. उदा., तुम्ही जेव्हा एक फाइल वाचायला उघडता, तेव्हा पायथॉन ती फाइल करंट डिरेक्टरीमध्ये शोधतो.

`os` मोड्युल फाइल्स आणि डिरेक्टरीझसाठी फंक्शन्स पुरवते ('os' हे 'operating system' चे संक्षिप्त रूप आहे); `os.getcwd` करंट डिरेक्टरीचे नाव रिटर्न करते:

```
>>> import os
>>> cwd = os.getcwd()
>>> cwd
'/home/dinsdale'
```

`cwd` हे 'current working directory' चे संक्षिप्त रूप आहे. ह्या उदाहरणात उत्तर आहे `/home/dinsdale`, जी `dinsdale` नावाच्या युझरची होम (home) डिरेक्टरी आहे.

फाइल किंवा डिरेक्टरी दर्शवणाऱ्या `'/home/dinsdale'` सारख्या स्ट्रिंगला **पाथ** (**path**) म्हणतात.

साधे फाइलनेम जसे `memo.txt` सुद्धा पाथ मानले जाते, पण तो **सापेक्ष पाथ** (**relative path**) आहे, कारण तो करंट डिरेक्टरीच्या सापेक्ष असतो. जर करंट डिरेक्टरी `/home/dinsdale` असेल तर `memo.txt` हा पाथ `/home/dinsdale/memo.txt` दर्शवतो.

असा पाथ जो `/` ने सुरू होतो तो करंट डिरेक्टरीसापेक्ष नसतो; त्याला **निरपेक्ष पाथ** (**absolute path**) म्हणतात. एका फाइलचा निरपेक्ष पाथ शोधण्यासाठी तुम्ही `os.path.abspath` वापरू शकता:

```
>>> os.path.abspath('memo.txt')
'/home/dinsdale/memo.txt'
```

`os.path` मध्ये फाइलनेम्स आणि पाथ्सवर चालणारी अजूनही फंक्शन्स आहेत. उदा., `os.path.exists` फंक्शन दिलेली फाइल किंवा डिरेक्टरी अस्तित्वात आहे की नाही हे तपासते:

```
>>> os.path.exists('memo.txt')
True
```

जर अस्तित्वात असेल तर `os.path.isdir` हे तपासते की त्या नावाची डिरेक्टरी आहे का:

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('/home/dinsdale')
True
```

त्याचप्रमाणे `os.path.isfile` तपासते की ती फाइल आहे का.

`os.listdir` हे फंक्शन दिलेल्या डिरेक्टरीमधील फाइल्स आणि इतर डिरेक्टरीझ-ची लिस्ट रिटर्न करते:

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

ह्या फंक्शन्सचे प्रात्यक्षिक करून दाखण्याच्या हेतूने, खालील उदाहरण एका डिरेक्टरीमधून 'चालत' जाते ('walks' through), प्रत्येक फाइलचे नाव प्रिंट करते, आणि सर्व डिरेक्टरीझवर रिकर्सिव्ह (recursive) कॉल करते.

```
def walk(dirname):
    for name in os.listdir(dirname):
        path = os.path.join(dirname, name)

        if os.path.isfile(path):
            print(path)
        else:
            walk(path)
```

`os.path.join` हे फंक्शन एक डिरेक्टरी आणि फाइलनेम घेऊन त्यांना जोडून पूर्ण पाथ देते.

`os` मोड्युल `walk` नावाचे फंक्शन पुरवते जे वरील फंक्शनसारखे पण जास्ती वैशिष्ट्यपूर्ण आहे. सराव म्हणून त्याचे डॉक्युमेंटेशन वाचा आणि दिलेल्या डिरेक्टरीमधील आणि तिच्यातील सर्व डिरेक्टरीमधील फाइल्सची नावे दाखवा. उत्तर: http://thinkpython2.com/code/walk.py.

१४.५ एक्सेप्शन झेलणे (Catching exceptions)

फाइल लिहिताना आणि वाचताना अनेक गोष्टी बिघडू शकतात. जर तुम्ही अशी फाइल उघडायचा प्रयत्न केलात जी अस्तित्वातच नाहीये, तर तुम्हाला `FileNotFoundError` मिळेल:

```
>>> fin = open('bad_file')
FileNotFoundError: [Errno 2] No such file or directory: 'bad_file'
```

जर तुम्हाला ती फाइल बघायची परवानगी नसेल:

```
>>> fout = open('/etc/passwd', 'w')
PermissionError: [Errno 13] Permission denied: '/etc/passwd'
```

आणि जर तुम्ही एक डिरेक्टरी वाचायला उघडली तर तुम्हाला मिळते:

```
>>> fin = open('/home')
IsADirectoryError: [Errno 21] Is a directory: '/home'
```

ह्या प्रकारचे एरर्स टाळण्यासाठी तुम्ही `os.path.exists` आणि `os.path.isfile` सारखी फंक्शन्स वापरू शकता, पण ते तपासण्याचा कोड लिहिण्यासाठी खूप वेळ लागेल (कर 'Errno 21' ह्यावरून काही अंदाज लावता येत असेल तर तो म्हणजे कमीतकमी २१ प्रकारे बिघाड होऊ शकतो).

चांगला मार्ग हा आहे की काही एरर आल्यावरच तो हाताळणे, आणि `try` स्टेटमेंट तेच करते. सिंटॅक्स `if...else` सारखाच आहे:

```
try:
    fin = open('bad_file')
except:
    print('Something went wrong.')
```

पायथॉन `try` मधील स्टेटमेंट्स एक्सेक्युट करायला सुरुवात करतो. जर सर्व सुरळीत पार पडले तर तो `except` मधील स्टेटमेंट्स सोडून देतो आणि पुढे जातो. जर एक्सेप्शन आले तर तो `try` च्या बाहेर उडी मारून `except` मधील स्टेटमेंट्स रन करतो.

`try` वापरून एक्सेप्शन हाताळण्याला एक्सेप्शन **झेलणे** (**catching**) असे म्हणतात. ह्या उदाहरणात, `except` मध्ये एक मेसेज प्रिंट केला आहे जो जास्ती उपयोगी नाहीये. साधारणपणे एक्सेप्शन झेलल्यावर तुम्हाला चूक दुरूस्त करण्याची, पुन्हा प्रयत्न करण्याची किंवा निदान चांगल्या प्रकारे प्रोग्रामचा शेवट करण्याची संधी मिळते.

१४.६ डेटाबेस (Databases)

डेटाबेस (**database**) म्हणजे अशी फाइल जिची रचना डेटा जपून ठेवायला केली आहे. अनेक डेटाबेसेसची रचना ही डिक्शनरीसारखीच असते, म्हणजे ते keys व्हॅल्यूझ-ना मॅप करतात. डेटाबेस आणि डिक्शनरीमधला सर्वांत मोठा फरक म्हणजे डेटाबेस हा डिस्कमध्ये (किंवा इतर कायमस्वरूपी माध्यमात) ठेवलेला असतो, म्हणजे तो प्रोग्राम संपला तरी राहतो.

`dbm` हे मोड्युल डेटाबेस बनवण्यासाठी आणि त्यावर प्रक्रिया करण्यासाठी फायदेशीर आहे. उदाहरण म्हणून आपण एक डेटाबेस बनवूया ज्यात फोटो-फाइल्ससाठी शीर्षक/वर्णन ठेवले जाईल.

डेटाबेस उघडणे फाइल उघडण्यासारखे आहे:

```
>>> import dbm
>>> db = dbm.open('captions', 'c')
```

Mode `'c'` चा अर्थ असा की ते डेटाबेस आधीपासून नसेल तर बनवले पाहिजे (‘create’). ह्याचा परिणाम म्हणजे एक डेटाबेस ऑब्जेक्ट जे (बहुतांश ऑपरेशन्ससाठी) एका डिक्शनरीसारखे वापरता येते.

नवीन आयटम बनवल्यावर `dbm` डेटाबेस फाइल अपडेट करते.

```
>>> db['cleese.png'] = 'Photo of John Cleese.'
```

एखादा आयटम बघितलात तर `dbm` फाइल वाचते:

```
>>> db['cleese.png']
b'Photo of John Cleese.'
```

ह्याने एक **बाइट्स ऑब्जेक्ट** (**bytes object**) मिळतो म्हणून तो `b` ने सुरू होतो. बाइट्स ऑब्जेक्ट हा अनेकप्रकारे स्ट्रिंगसारखाच असतो. तुम्ही पायथॉनमध्ये पुढे गेलात की त्यांतला फरक महत्त्वाचा ठरतो, पण आता तुम्ही तो सोडून देऊ शकता.

जर तुम्ही आधीपासून असलेल्या key ला दुसरी असाइनमेंट केली तर `dbm` जुनी व्हॅल्यू बदलतो:

```
>>> db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'
>>> db['cleese.png']
b'Photo of John Cleese doing a silly walk.'
```

काही डिक्शनरी मेथड्स, जसे `keys` आणि `items` ह्या डेटाबेस ऑब्जेक्टवर चालत नाहीत. पण `for` वापरून इटरेशन चालते:

```
for key in db.keys():
    print(key, db[key])
```

फाइलसारखेच, तुमचे डेटाबेसवरील काम झाले असल्यास तुम्ही त्याला बंद केले पाहिजे:

```
>>> db.close()
```

१४.७ लोणचे घालणे (Pickling)

Keys आणि व्हॅल्यूझ स्ट्रिंग किंवा बाइट्स (bytes) असणे गरजेचे असणे ही `dbm` ची एक मर्यादा आहे. जर तुम्ही दुसरा टाइप वापरायचा प्रयत्न केला तर तुम्हाला एरर मिळेल.

ह्याठिकाणी `pickle` मोड्युल कामाला येऊ शकते. जवळजवळ सर्व टाइपच्या ऑब्जेक्टचे ते डेटाबेसमध्ये ठेवण्यासाठी स्ट्रिंगमध्ये रुपांतर करू शकते, आणि नंतर त्या स्ट्रिंग्सचे परत ऑब्जेक्ट्समध्ये रुपांतर करते.

`pickle.dumps` एक ऑब्जेक्ट घेऊन त्याचे स्ट्रिंगमधील स्वरूप रिटर्न करते (`pickle.dumps` हे 'dump string' चे संक्षिप्त रूप आहे).

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

हे रूप आपल्या वाचण्यासाठी नसून `pickle` ला समजणे सोपे जाण्यासाठी आहे; `pickle.loads` ('load string') परत ऑब्जेक्ट बनवते:

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> t2
[1, 2, 3]
```

जरी नवीन ऑब्जेक्टची व्हॅल्यू जुन्यासारखीच असली तरी (सामान्यपणे) तो आणि जुना हे एकच नसतात:

```
>>> t1 == t2
True
>>> t1 is t2
False
```

दुसऱ्या शब्दांत, pickling आणि नंतर unpickling चा परिणाम ऑब्जेक्ट कॉपी करण्यासारखाच असतो.

तुम्ही `pickle` वापरून स्ट्रिंगसोडून इतर व्हॅल्यूझ एका डेटाबेसमध्ये ठेवू शकता. खरे तर हे दोन्ही एकत्र वापरणे इतके कॉमन आहे की त्यांना `shelve` मोड्युलमध्ये एकत्र केले आहे.

१४.८ पाइप्स (Pipes)

बहुतांश ऑपरेटिंग सिस्टम्स तिला कमांड (आज्ञा) देण्यासाठी **शेल** (**shell**) नावाचा एक दुवा (interface) उपलब्ध करून देतात. शेलमध्ये साधारणपणे फाइल सिस्टममध्ये इकडून तिकडे जाण्यासाठी आणि सॉफ्टवेअर-प्रोग्राम सुरू करण्यासाठी कमांड्स असतात. उदा., युनिक्स (Unix) मध्ये तुम्ही `cd` वापरून डिरेक्टरी बदलू शकता, `ls` वापरून डिरेक्टरीत काय आहे ते बघू शकता, आणि वेब ब्राउझरचा प्रोग्राम सुरू करण्यासाठी, उदा., `firefox` ही कमांड देऊ शकता.

जो प्रोग्राम शेलमधून सुरू करता येऊ शकतो तो पायथॉनमधून **पाइप ऑब्जेक्ट** (**pipe object**) वापरून सुरू करता येऊ शकतो; हा ऑब्जेक्ट चालू असणाऱ्या प्रोग्रामची माहिती ठेवतो.

उदा., `ls -l` ही युनिक्स कमांड साधारणपणे करंट डिरेक्टरीमध्ये काय आहे ते सविस्तरपणे (long format) सांगते. तुम्ही `os.popen` वापरून `ls` रन करू शकता[3]:

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

[3] `popen` हे फंक्शन भविष्यात पायथॉनमधून काढले जाण्याची शक्यता आहे (popen is deprecated), म्हणजे ते न वापरता त्याऐवजी `subprocess` हे मोड्युल वापरावे अशी पायथॉनची अधिकृत घोषणा आहे. पण साध्या कामांसाठी `subprocess` हे मोड्युल गरजेपेक्षा जास्ती किचकट असल्यामुळे मूळ लेखकाचे म्हणणे आहे की ते जोपर्यंत चालते आहे तोपर्यंत `popen` च वापरतील.

शेल कमांड स्ट्रिंगरूपात अर्ग्युमेंट म्हणून पाठवली जाते. रिटर्न व्हॅल्यू ही उघडलेल्या फाइलसारखा ऑब्जेक्ट असते. आपण `ls` चे आउटपुट एकेक ओळ असे `readline` ने वाचू शकता, किंवा सर्व आउटपुट `read` ने मिळवू शकता:

```
>>> res = fp.read()
```

काम झाल्यावर तुम्ही फाइलसारखाच पाइप बंद केला पाहिजे:

```
>>> stat = fp.close()
>>> print(stat)
None
```

रिटर्न व्हॅल्यू ही `ls` कमांडची अंतिम स्थिती दर्शवते; `None` म्हणजे ती कमांड (काहीही एरर शिवाय) सामान्यपणे संपली.

उदा., बहुतांश युनिक्स सिस्टम्स `md5sum` ही कमांड पुरवतात जी एक फाइल वाचून तिची 'चेकसम' ('checksum') बनवते. तुम्ही MD5 विषयी पुढील लिंकवर वाचू शकता: http://en.wikipedia.org/wiki/Md5.

ही कमांड वापरून दोन फाइल्स सारख्याच आहेत की नाही (त्यांत एकसारखाच डेटा आहे की नाही) हे शीघ्रपणे तपासता येते. वेगळा डेटा असलेल्या दोन फाइल्सची सारखीच चेकसम येण्याची संभाव्यता (probability) खूप कमी आहे (म्हणजेच, ब्रह्मांड कोसळण्याआधी असे होणे अवघड आहे).

पाइप वापरून तुम्ही पायथॉनमधून `md5sum` चालवून उत्तर मिळवू शकता:

```
>>> filename = 'book.tex'
>>> cmd = 'md5sum ' + filename
>>> fp = os.popen(cmd)
>>> res = fp.read()
>>> stat = fp.close()
>>> print(res)
1e0033f0ed0656636de0d75144ba32e0  book.tex
>>> print(stat)
None
```

१४.९ मोड्युल लिखाण (Writing modules)

ज्या फाइलमध्ये पायथॉन कोड आहे ती मोड्युल म्हणून इंपोर्ट (import) केली जाऊ शकते. उदा., समजा तुमच्याकडे `wc.py` नावाची फाइल असेल ज्यात खालील कोड आहे:

```
def linecount(filename):
    count = 0
    for line in open(filename):
        count += 1
    return count

print(linecount('wc.py'))
```

जर तुम्ही हा प्रोग्राम रन केला तर तर तो स्वत:ला वाचून किती ओळी आहेत हे प्रिंट करेल, ज्याचे उत्तर ७ आहे. तुम्ही ही अशीही इंपोर्ट करू शकता:

```
>>> import wc
7
```

आता तुमच्याकडे `wc` हे मोड्युल ऑब्जेक्ट आहे:

```
>>> wc
<module 'wc' from 'wc.py'>
```

हे मोड्युल ऑब्जेक्ट `linecount` पुरवते:

```
>>> wc.linecount('wc.py')
7
```

तर अशाप्रकारे तुम्ही पायथॉनमध्ये मोड्युल लिहू शकता.

ह्याठिकाणी एकच गडबड अशी आहे की जेव्हा तुम्ही हे मोड्युल इंपोर्ट करता तेव्हा त्याच्या शेवटी असलेला टेस्ट कोड रन होतो. साधारणपणे तुम्ही जेव्हा एखादे मोड्युल इंपोर्ट करता तेव्हा ते नवीन फंक्शन्स बनवते पण रन नाही करत.

ज्या प्रोग्राम्सना मोड्युल म्हणून इंपोर्ट केले जाण्याची योजना असते त्यांत सहसा खालील कोड वापरतात:

```
if __name__ == '__main__':
    print(linecount('wc.py'))
```

`__name__` हे एक बिल्ट-इन व्हेरिएबल आहे ज्याला प्रोग्राम सुरू झाल्यावर व्हॅल्यू दिली जाते. जर तो प्रोग्राम स्क्रिप्ट म्हणून रन केला जात असेल तर `__name__` ची व्हॅल्यू `'__main__'` असते, म्हणजेच तेव्हा टेस्टिंग करणारा कोड रन होतो. नाहीतर जर मोड्युल इंपोर्ट होत असेल तर तो टेस्ट कोड रन नाही होत.

सराव म्हणून हे उदाहरण `wc.py` नावाच्या फाइलमध्ये लिहा आणि ती फाइल स्क्रिप्ट म्हणून रन करा. नंतर पायथॉन इंटरप्रिटर सुरू करून `import wc` रन करा. आता, मोड्युल इंपोर्ट होताना `__name__` ची व्हॅल्यू काय आहे?

इशारा: जर तुम्ही आधीच इंपोर्ट केलेले मोड्युल इंपोर्ट केले तर पायथॉन काहीच करत नाही. तो फाइल परत वाचत नाही, ती बदलली असेल तरी.

जर तुम्हाला मोड्युल परत री-लोड (reload) करायचे असेल तर तुम्ही `reload` हे बिल्ट-इन फंक्शन वापरू शकता, पण ते थोडे अडचणीचे होऊ शकते, म्हणून इंटरप्रिटर परत सुरू करून मोड्युल परत इंपोर्ट करणे हाच उत्तम मार्ग आहे.

१४.१० डीबगिंग (Debugging)

फाइल लिहिताना किंवा वाचताना स्पेस (whitespace) शी संबंधित समस्या उद्भवू शकतात. हे एरर्स डीबग करणे अवघड जाऊ शकते कारण स्पेस, टॅब, आणि न्यूलाइन साधारणपणे अदृश्य असतात:

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2  3
 4
```

`repr` ह्या बिल्ट-इन फंक्शनची मदत होऊ शकते. ते कोणतेही ऑब्जेक्ट घेऊन त्याचे स्ट्रिंग-रूप पाठवते. स्ट्रिंगमधील whitespace कॅरेक्टर्स ते बॅकस्लॅश (backslash) पद्धतीने व्यक्त करते:

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

ह्याचा डीबगिंगसाठी फायदा होऊ शकतो.

अजून एक अडथळा ह्यामुळे उभा राहू शकतो की वेगवेगळ्या सिस्टम्स ओळीचा शेवट दर्शवण्यासाठी वेगवेगळी कॅरेक्टर्स वापरतात. काही सिस्टम्स `\n` हे न्यूलाइन कॅरेक्टर वापरतात, तर काही `\r` हे 'रिटर्न' कॅरेक्टर वापरतात. काही दोन्ही वापरतात. जर तुम्ही फाइल एका सिस्टमवरून दुसरीवर हलवली तर ह्या विसंगतींमुळे अडचणी येऊ शकतात.

बहुतांश सिस्टम्ससाठी, एकाचे दुसऱ्यात रुपांतर करणारे प्रोग्राम्स उपलब्ध असतात. तुम्ही पुढील लिंकवर ते शोधू शकता आणि ह्या समस्येविषयी वाचू शकता: http://en.wikipedia.org/wiki/Newline.

किंवा तुम्ही स्वतः असा प्रोग्राम लिहू शकता.

१४.११ शब्दार्थ

दीर्घस्थायी (persistent): अशा प्रोग्रामशी संबंधित जो कायम चालू असतो आणि त्याचा काही डेटा कायमस्वरूपी माध्यमात (permanent storage मध्ये) जपून ठेवतो.

फॉरमॅट ऑपरेटर (format operator): % हा ऑपरेटर जो एक फॉरमॅट स्ट्रिंग आणि एक टपल घेऊन अशी स्ट्रिंग तयार करतो ज्यात फॉरमॅट स्ट्रिंगमध्ये सांगितलेल्या स्वरुपात टपलचे एलेमेंट्स असतात.

फॉरमॅट स्ट्रिंग (format string): फॉरमॅट ऑपरेटरमध्ये वापरलेली ती स्ट्रिंग ज्यात फॉरमॅट सीक्वेन्सेस असतात.

फॉरमॅट सीक्वेन्स (format sequence): फॉरमॅट स्ट्रिंगमधील एक कॅरेक्टर्सचा सीक्वेन्स, जसा %d, जो एका व्हॅल्यूचे स्वरूप (ती कशी फॉरमॅट करायची ते) दर्शवतो.

टेक्स्ट फाइल (text file): डिस्कसारख्या कायमस्वरूपी माध्यमात जपून ठेवलेला कॅरेक्टर्सचा एक सीक्वेन्स.

डिरेक्टरी (directory): नाव दिलेला फाइल्सचा संग्रह; फोल्डर (folder) सुद्धा म्हणतात.

पाथ (path): फाइल दर्शवणारी स्ट्रिंग.

सापेक्ष पाथ (relative path): करंट डिरेक्टरीपासून सुरू होणारा पाथ.

निरपेक्ष पाथ (absolute path): फाइल सिस्टममधील सर्वोच्च डिरेक्टरीपासून सुरू होणारा पाथ.

झेलणे (catch): एखाद्या एक्सेप्शनला `try` आणि `except` स्टेटमेंट्स वापरून प्रोग्राम संपवू न देणे.

डेटाबेस (database): अशी फाइल ज्यात डेटा डिक्शनरीसारखा, म्हणजे keys आणि संबंधित व्हॅल्यूझच्या स्वरुपात ठेवलेला असतो.

बाइट्स ऑब्जेक्ट (bytes object): स्ट्रिंगसमान असलेला एक ऑब्जेक्ट.

शेल (shell): असा प्रोग्राम ज्यात युझर कमांड्स लिहितात, आणि त्यानुसार तो प्रोग्राम इतर प्रोग्राम्स सुरू करतो.

पाइप ऑब्जेक्ट (pipe object): चालू असणारा एक प्रोग्राम दर्शवणारा ऑब्जेक्ट, जो वापरून एक पायथॉन प्रोग्राम कमांड्स रन करून त्यांची उत्तरे वाचू शकतो.

१४.१२ प्रश्नसंच (Exercises)

प्रश्न १४.१. *एक* `sed` *नावाचे फंक्शन लिहा जे पुढील अर्ग्युमेंट्स घेते: एक नमुना स्ट्रिंग, एक बदली स्ट्रिंग, आणि दोन फाइलनेम्स (filenames). त्यात तुम्ही पहिली फाइल वाचून दुसरीत लिहिणार आहात (जर दुसरी अस्तित्वात नसेल तर तिला बनवावे). जर नमुना स्ट्रिंग फाइलमध्ये कुठेही असेल तर तिला बदली स्ट्रिंगने बदलले पाहिजे.*

जर फाइल्स उघडताना, वाचताना, लिहिताना, किंवा बंद करताना काही एरर आला तर तुमच्या प्रोग्रामने ते एक्सेप्शन झेलून एक एरर मेसेज प्रिंट करून नंतर प्रोग्राम संपवला पाहिजे.

प्रश्न १४.२. *जर तुम्ही प्रश्न १२.२ चे उत्तर पुढील लिंकवरून डाऊनलोड केले तर तुम्हाला दिसेल की त्यात कॅरेक्टर्सची सॉर्टेड स्ट्रिंग त्या कॅरेक्टर्स पासून बनवता येणाऱ्या शब्दांच्या लिस्टला मॅप करणारी डिक्शनरी बनवली आहे: http://thinkpython2.com/code/anagram_sets.py.*

उदा., `'opst'` *ही* `['opts', 'post', 'pots', 'spot', 'stop', 'tops']` *ला मॅप होते.*

एक मोड्युल लिहा जे `anagram_sets` *इंपोर्ट करते आणि दोन नवीन फंक्शन्स पुरवते: ही ॲनाग्राम डिक्शनरी एका 'shelf' मध्ये जपून ठेवणारे* `store_anagrams`*; आणि दिलेला शब्द शोधून त्याच्या ॲनाग्राम्सची लिस्ट रिटर्न करणारे* `read_anagrams`*. उत्तर: http://thinkpython2.com/code/anagram_db.py.*

प्रश्न १४.३. *MP3 फाइल्सच्या एका मोठ्या संग्रहात एकाच गाण्याच्या वेगळ्या डिरेक्टरीमध्ये किंवा वेगळ्या नावाच्या अनेक कॉपीझ असू शकतात. ह्या प्रश्नाचा उद्देश नकला शोधणे हा आहे.*

१. *दिलेली डिरेक्टरी आणि तिच्यातल्या सर्व डिरेक्टरीझ रिकर्सिव्हली (recursively) बघून, दिलेल्या प्रत्ययाच्या (suffix, उदा., .mp3) सर्व फाइल्सचे पूर्ण पाथ्स (paths) रिटर्न करणारा प्रोग्राम लिहा. टीप:* `os.path` *फाइल आणि पाथ वर प्रक्रिया करणारी अनेक उपयोगी फंक्शन्स पुरवते.*

२. *नकला शोधण्यासाठी तुम्ही* `md5sum` *वापरून प्रत्येक फाइलची 'checksum' काढू शकता. जर दोन फाइल्सची चेकसम सारखीच असेल तर त्या सारख्याच आहेत असे आपण समजू शकतो.*

३. *पुनर्तपासणीसाठी तुम्ही* `diff` *ही युनिक्स कमांड वापरू शकता.*

उत्तर: http://thinkpython2.com/code/find_duplicates.py.

प्रकरण १५

क्लास आणि ऑब्जेक्ट (Class and object)

आता, तुम्हाला हे माहीत आहे की फंक्शन्स वापरून कोड-ची रचना कशी करायची आणि बिल्ट-इन टाइप्स वापरून डेटा-ची रचना कशी करायची. पुढची पायरी 'ऑब्जेक्ट-ओरिएंटेड प्रोग्रामिंग' ('object-oriented programming') आहे, ज्यात प्रोग्रामर नवीन टाइप बनवून कोड आणि डेटा दोन्हींची रचना करू शकतो (programmer-defined types, प्रोग्रामर-परिभाषित टाइप्स). ऑब्जेक्ट-ओरिएंटेड प्रोग्रामिंग हा एक मोठा विषय असून तिथपर्यंत पोहोचायला आपल्याला काही प्रकरणे लागतील.

ह्या प्रकरणातील कोड-ची उदाहरणे पुढील लिंकवर आहेत: http://thinkpython2.com/code/Point1.py.

प्रश्नांची उत्तरे पुढील लिंकवर आहेत: http://thinkpython2.com/code/Point1_soln.py.

१५.१ प्रोग्रामर-परिभाषित टाइप्स (Programmer-defined types)

आपण पायथॉनचे अने बिल्ट-इन टाइप्स वापरले आहेत; आता आपण नवीन टाइपची व्याख्या देणार आहोत. उदाहरण म्हणून आपण `Point` नावाचा टाइप बनवणार आहोत ज्याने आपण XY-प्रतलातील बिंदू (a point in XY-plane) दर्शवू शकतो.

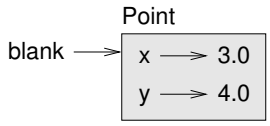
गणितातील भाषेत, बिंदू हे सहसा साध्या कंसात निर्देशकांच्यामध्ये स्वल्पविराम टाकून लिहिले जातात. उदा., $(0, 0)$ उगम-बिंदू (origin) दर्शवतो, आणि (x, y) हा उगम-बिंदूच्या x एकक उजवीकडे आणि y एकक वर असणारा बिंदू दर्शवतो.

पायथॉनमध्ये बिंदू दर्शवण्यासाठी आपण अनेक मार्ग पत्करू शकतो:

- आपण दोन्ही निर्देशक दोन वेगळ्या व्हेरिएबल्समध्ये ठेवू शकतो: `x` आणि `y`.
- आपण दोन्ही निर्देशक लिस्ट किंवा टपलमध्ये एलेमेंट्स म्हणून ठेवू शकतो.
- आपण एक नवीन टाइप बनवून बिंदू त्याच्या ऑब्जेक्टमध्ये ठेवू शकतो.

नवीन टाइप बनवणे अन्य पर्यायांपेक्षा थोडे अवघड जरी असले तरी त्याचे काही फायदे आहेत जे आपल्याला लवकरच दिसतील.

प्रोग्रामर-परिभाषित टाइपला **क्लास** (**class**) म्हणतात. क्लासची डेफनिशन अशी दिसते:



आकृती १५.१: ऑब्जेक्ट डायग्राम (Object diagram).

```
class Point:
    """Represents a point in 2-D space."""
```

हेडर सांगते की नवीन क्लासचे नाव `Point` आहे. बॉडी एक डॉकस्ट्रिंग (docstring) आहे जी क्लास कशासाठी आहे ते सांगते. क्लास डेफनिशनमध्ये आपण व्हेरिएबल्स आणि मेथड्ससुद्धा देऊ शकतो, पण ते आपण नंतर पाहू.

`Point` नावाचा क्लास डिफाइन (define) केल्यावर **क्लास ऑब्जेक्ट** (**class object**) बनतो.

```
>>> Point
<class '__main__.Point'>
```

`Point` सर्वोच्च पातळीवर डिफाइन केल्यामुळे त्याचे 'पूर्ण नाव' ('full name') `__main__.Point` असे आहे.

क्लास ऑब्जेक्ट हा एका ऑब्जेक्ट बनवण्याच्या कारखान्यासारखा आहे. एक Point बनवण्यासाठी तुम्ही `Point` असे कॉल करता जसे काही तो एक फंक्शन आहे.

```
>>> blank = Point()
>>> blank
<__main__.Point object at 0xb7e9d3ac>
```

रिटर्न व्हॅल्यू ही एका Point ऑब्जेक्टचा रेफ्रन्स (reference) आहे, जी आपण `blank` ला असाइन करतो.

नवीन ऑब्जेक्ट बनवण्याच्या क्रियेला **इन्स्टॅन्शियेशन** (**instantiation**) म्हणतात, आणि असे म्हणतात की तो ऑब्जेक्ट त्या क्लासचा **इन्स्टन्स** (**instance**) आहे.

जेव्हा तुम्ही एक इन्स्टन्स प्रिंट करता, तो कोणत्या क्लासचा आहे आणि मेमरीमध्ये कुठे ठेवला आहे हे पायथॉन तुम्हाला सांगतो (उपसर्ग `0x` म्हणजे पुढची संख्या ही हेक्सा-डेसिमल, hexadecimal, मध्ये आहे).

प्रत्येक ऑब्जेक्ट हा कोणत्यातरी क्लासचा इन्स्टन्स असतो, म्हणजेच 'ऑब्जेक्ट' आणि 'इन्स्टन्स' हे शब्द अदलून-बदलून वापरू शकतो. पण ह्या प्रकरणात आपण 'इन्स्टन्स'चा वापर प्रोग्रामर-परिभाषित टाइप्स दर्शवण्यासाठी करू.

१५.२ ॲट्रिब्युट (Attributes)

एका इन्स्टन्सला व्हॅल्यूझ असाइन करण्यासाठी तुम्ही डॉट नोटेशन (dot notation) वापरू शकता:

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

हा सिंटॅक्स एका मोड्युलमधून व्हेरिएबल निवडताना वापरतात त्या सिंटॅक्ससारखा आहे, उदा., `math.pi` आणि `string.whitespace`. पण ह्याठिकाणी आपण, दिलेल्या नावांनी संबोधल्या जाणाऱ्या एलेमेंट्सना व्हॅल्यूझ असाइन करत आहोत. अशा एलेमेंटला **ॲट्रिब्युट** (**attribute**[१]) असे म्हणतात.

आकृती १५.१ मधील स्टेट डायग्राम ह्या असाइनमेंट्सचा परिणाम दाखवतात. एक ऑब्जेक्ट आणि त्याचे ॲट्रिब्युट्स दाखवणाऱ्या स्टेट डायग्रामला **ऑब्जेक्ट डायग्राम** (**object diagram**) म्हणतात.

[१] अनुवादकाची टिप्पणी: मूळ पुस्तकात ह्या शब्दाचा उच्चार कसा करायचा ह्यावर एक वाक्य आहे. मी तो शब्द देवनागरी लिपीतच लिहिल्याने ते लागू होत नाही, पण तुमच्या माहितीसाठी इथे तळटीपेत देत आहे. As a noun, 'AT-trib-ute' is pronounced with emphasis on the first syllable, as opposed to 'a-TRIB-ute', which is a verb.

blank व्हेरिएबल एक Point ऑब्जेक्ट दर्शवते ज्यात दोन ॲट्रिब्युट्स आहेत. प्रत्येक ॲट्रिब्युट एक फ्लोटिंग-पॉइंट संख्या दर्शवतो.

तोच सिंटॅक्स वापरून तुम्ही एका व्हेरिएबलची व्हॅल्यू बघू शकता:

```
>>> blank.y
4.0
>>> x = blank.x
>>> x
3.0
```

इथे, blank.x ह्या एक्स्प्रेशनचा अर्थ असा होतो की 'blank जे ऑब्जेक्ट दर्शवते त्याच्याकडे जा आणि x ची व्हॅल्यू आणा.' ह्या उदाहरणात, आपण ती व्हॅल्यू x नावाच्या व्हेरिएबलल असाइन करतो. व्हेरिएबल x आणि ॲट्रिब्युट x ह्यांमध्ये काहीच बेबनाव (conflict) नाही, इमानेइतबारे ते स्वतंत्रपणे राहतात.

डॉट नोटेशन कोणत्या एक्स्प्रेशनमध्ये वापरू शकतो. उदा.:

```
>>> '(%g, %g)' % (blank.x, blank.y)
'(3.0, 4.0)'
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> distance
5.0
```

इन्स्टन्सला तुम्ही अर्ग्युमेंट म्हणून नेहमीसारखे पाठवू शकता. उदा.:

```
def print_point(p):
    print('(%g, %g)' % (p.x, p.y))
```

print_point एक Point अर्ग्युमेंट म्हणून घेते आणि त्याला गणितीय भाषेत दाखवते. ते कॉल करण्यासाठी तुम्ही blank अर्ग्युमेंट म्हणून पाठवू शकता:

```
>>> print_point(blank)
(3.0, 4.0)
```

ह्या फंक्शनमध्ये, p हे blank चे एलिअस (alias) आहे, म्हणजे जर फंक्शनने p मध्ये काही बदल केला तर blank सुद्धा बदलते.

सराव म्हणून distance_between_points नावाचे फंक्शन लिहा जे दोन Points घेते आणि त्यांतले अंतर रिटर्न करते.

१५.३ आयत (Rectangles)

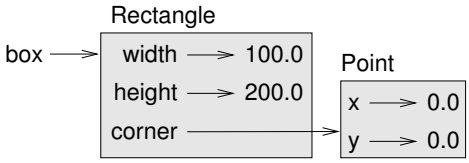
कधीकधी कोणते ॲट्रिब्युट्स निवडायचे हे स्पष्ट असते, पण कधीकधी तुम्हाला काही निर्णय घ्यावे लागतात. उदा., समजा तुम्हाला (XY-प्रतलातील) आयतांसाठी एक क्लास डिझाइन करायचा आहे. त्याचा आकार आणि स्थान दर्शवण्यासाठी तुम्ही कोणते ॲट्रिब्युट्स वापराल? तुम्ही सध्या कोन (angle) विसरून जा, असे समजा की तो आयत एक तर उभा किंवा आडवा आहे.

कमीतकमी दोन शक्यता आहेत:

- तुम्ही आयताचा एक शिरोबिंदू (corner) किंवा केंद्र आणि रुंदी व लांबी ठेवू शकता.
- तुम्ही दोन विरुद्ध शिरोबिंदू ठेवू शकता.

ह्यावेळेला कोणती पद्धत जास्ती चांगली आहे हे सांगणे कठीण आहे, तर आपण उदाहरण म्हणून पहिली इंप्लेमेंट करूया.

ही क्लास डेफनिशन:



आकृती १५.२: ऑब्जेक्ट डायग्राम (Object diagram).

```
class Rectangle:
    """Represents a rectangle.

    attributes: width, height, corner.
    """
```

डॉकस्ट्रिंग ॲट्रिब्युट्सची यादी देते: `width` आणि `height` ह्या संख्या आहेत; `corner` एक Point ऑब्जेक्ट आहे जो खालचा-डावा शिरोबिंदू देतो.

एक आयत दर्शवण्यासाठी तुम्हाला एक Rectangle तयार (instantiate, इन्स्टॅन्शियेट) करावा लागेल आणि ॲट्रिब्युट्सना व्हॅल्यूझ असाइन कराव्या लागतील:

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

इथे, `box.corner.x` ह्या एक्स्प्रेशनचा अर्थ असा होतो की, 'box जे ऑब्जेक्ट दर्शवते त्याच्याकडे जा आणि `corner` नावाचा ॲट्रिब्युट निवडा; मग त्या ऑब्जेक्टकडे जा आणि `x` नावाचा ॲट्रिब्युट निवडा.'

आकृती १५.२ ह्या ऑब्जेक्टची स्थिती दाखवते. जो ऑब्जेक्ट दुसऱ्या ऑब्जेक्टचा ॲट्रिब्युट असतो त्याला **एम्बेडेड** (**embedded**) म्हणतात.

१५.४ इन्स्टन्सला रिटर्न व्हॅल्यू म्हणून पाठवणे (Instances as return values)

फंक्शन इन्स्टन्स रिटर्न करू शकते. उदा., `find_center` एक `Rectangle` घेऊन त्याचे केंद्र असलेला `Point` रिटर्न करते:

```
def find_center(rect):
    p = Point()
    p.x = rect.corner.x + rect.width/2
    p.y = rect.corner.y + rect.height/2
    return p
```

खालील उदाहरण `box` अर्ग्युमेंट म्हणून पाठवते आणि मिळालेले उत्तर (Point) `center` ला असाइन करते:

```
>>> center = find_center(box)
>>> print_point(center)
(50, 100)
```

१५.५ ऑब्जेक्ट म्युटबल असतो (Objects are mutable)

तुम्ही ऑब्जेक्टची स्टेट बदलू शकता; त्यासाठी कोणत्याही ॲट्रिब्युटला एक असाइनमेंट करा. उदा., एका आयताचे स्थान *न* बदलता आकार बदलण्यासाठी तुम्ही `width` आणि `height` च्या व्हॅल्यूझ बदलू शकता:

```
box.width = box.width + 50
box.height = box.height + 100
```

तुम्ही ऑब्जेक्ट बदलणारे फंक्शनही लिहू शकता. उदा., `grow_rectangle` एक Rectangle ऑब्जेक्ट आणि `dwidth` आणि `dheight` ह्या दोन संख्या घेऊन त्या अनुक्रमे आयताच्या रुंदी आणि लांबीमध्ये मिळवते:

`grow_rectangle` takes a Rectangle object and two numbers, `dwidth` and `dheight`, and adds the numbers to the width and height of the rectangle:

```
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

खालील उदाहरण त्याचा परिणाम दाखवते:

```
>>> box.width, box.height
(150.0, 300.0)
>>> grow_rectangle(box, 50, 100)
>>> box.width, box.height
(200.0, 400.0)
```

त्या फंक्शनमध्ये, `rect` हे `box` चा एलिअस आहे, म्हणून जेव्हा फंक्शन `rect` ला बदलते, तेव्हा `box` मध्ये सुद्धा बदल घडून येतो

सराव म्हणून `move_rectangle` नावाचे फंक्शन लिहा जे एक Rectangle आणि `dx` व `dy` ह्या दोन संख्या घेते. नंतर ते `corner` च्या `x`-निर्देशकात `dx` मिळवते आणि `corner` च्या `y`-निर्देशकात `dy` मिळवते. अशाप्रकारे ते आयताचे स्थान बदलते.

१५.६ कॉपी (Copying)

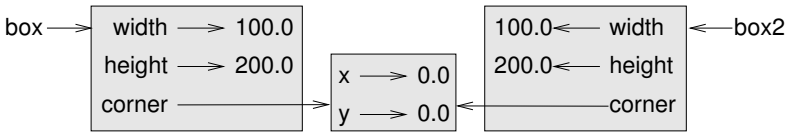
एलिअसच्या वापरामुळे प्रोग्राम समजून घेणे अवघड जाऊ शकते कारण एका ठिकाणचे बदल दुसऱ्या ठिकाणी अनपेक्षितपणे परिणाम करू शकतात. एखाद्या ऑब्जेक्टला दर्शवणाऱ्या सर्व व्हेरिएबल्सची नोंद ठेवणे अवघड होऊन बसते.

ऑब्जेक्ट कॉपी करणे हा कधीकधी एलिअस वापरण्याला पर्याय असतो. `copy` मोड्युलमध्ये `copy` नावाचे फंक्शन आहे जे कोणत्याही ऑब्जेक्टची प्रत (copy) बनवू शकते:

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0

>>> import copy
>>> p2 = copy.copy(p1)
```

ह्याठिकाणी `p1` आणि `p2` मध्ये सारखीच माहिती आहे पण ते एकच Point नाहीयेत.

```
>>> print_point(p1)
(3, 4)
>>> print_point(p2)
(3, 4)
>>> p1 is p2
```



आकृती १५.३: ऑब्जेक्ट डायग्राम (Object diagram).

```
False
>>> p1 == p2
False
```

इथे `is` ऑपरेटर दाखवतो की `p1` आणि `p2` एकच ऑब्जेक्ट नाहीयेत, जे आपण अपेक्षिले होते. पण आपण `==` ऑपरेटर `True` सांगेल अशी अपेक्षा केली होती कारण त्या बिंदूंमध्ये सारखीच माहिती आहे. ऐकून तुम्हाला खेद होईल की इन्स्टन्सेससाठी `==` ची डीफॉल्ट (प्रमाण, standard) प्रक्रिया `is` सारखीच आहे; तो ऑब्जेक्ट्स एकच आहेत का हे तपासतो, सारखे/समान आहेत का ते नाही. त्याचे कारण म्हणजे प्रोग्रामर-परिभाषित टाइपसाठी पायथॉन हे नाही सांगू शकत की सारखेपणा काय आहे. निदान अजूनपर्यंत तरी नाही.

जर तुम्ही `copy.copy` वापरून Rectangle ची प्रत बनवली तर तुम्हाला दिसेल की ते त्या Rectangle ऑब्जेक्टची प्रत बनवते पण आतील एम्बेडेड Point ची नाही.

```
>>> box2 = copy.copy(box)
>>> box2 is box
False
>>> box2.corner is box.corner
True
```

आकृती १५.३ त्याची ऑब्जेक्ट डायग्राम दाखवते.

ह्या प्रक्रियेला **शॅलो कॉपी** (**shallow copy**, उथळ प्रत) म्हणतात कारण ती ऑब्जेक्ट आणि त्यातील रेफ्रन्सेस कॉपी करते पण आतील एम्बेडेड ऑब्जेक्ट्स नाही.

बहुतांश कामांसाठी तुम्हाला हे नसते पाहिजे. ह्या उदाहरणात, दोन्हीपैकी एक Rectangle पाठवून `grow_rectangle` कॉल केले तर दुसऱ्यावर काही परिणाम होणार नाही, पण एकाने `move_rectangle` कॉल केले तर दोन्हींवर परिणाम होईल! हे वर्तन संभ्रमात टाकणारे आणि एरर घडवण्याची शक्यता वाढवणारे आहे.

सुदैवाने `copy` मोड्युल `deepcopy` नावाचे फंक्शन पुरवते जे फक्त तोच ऑब्जेक्ट कॉपी करत नाही तर त्या ऑब्जेक्टमधील एम्बेडेड ऑब्जेक्ट्स, आणि *त्या* ऑब्जेक्ट्स मधील एम्बेडेड ऑब्जेक्ट्स, आणि असेच पुढे...कॉपी करते. ह्या ऑपरेशनला **डीप कॉपी** (**deep copy**, खोल प्रत) म्हणतात हे ऐकून तुम्हाला आश्चर्य तर नक्कीच वाटणार नाही.

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
>>> box3.corner is box.corner
False
```

`box3` आणि `box` हे पूर्णपणे वेगळे ऑब्जेक्ट्स आहेत.

सराव म्हणून `move_rectangle` चे सुधारित रूप लिहा जे दिलेला जुना Rectangle बदलण्याऐवजी एक नवीन Rectangle बनवून रिटर्न करते.

१५.७ डीबगिंग (Debugging)

ऑब्जेक्ट्सवर काम करताना तुम्हाला काही नवीन एक्सेप्शन मिळण्याची शक्यता आहे. जर तुम्ही अस्तित्वात नसणारा ॲट्रिब्युट बघायचा प्रयत्न केला तर तुम्हाला `AttributeError` मिळेल:

```
>>> p = Point()
>>> p.x = 3
>>> p.y = 4
>>> p.z
AttributeError: Point instance has no attribute 'z'
```

जर तुम्हाला ऑब्जेक्टचा टाइप माहीत नसेल तर तुम्ही विचारू शकता:

```
>>> type(p)
<class '__main__.Point'>
```

ऑब्जेक्ट एका क्लासचा इन्स्टन्स आहे का हे तपासण्यासाठी तुम्ही `isinstance` सुद्धा वापरू शकता:

```
>>> isinstance(p, Point)
True
```

जर तुम्हाला शंका असेल की ऑब्जेक्टला एक विशिष्ठ ॲट्रिब्युट आहे की नाही तर तुम्ही `hasattr` हे बिल्ट-इन फंक्शन वापरू शकता:

```
>>> hasattr(p, 'x')
True
>>> hasattr(p, 'z')
False
```

इथे पहिले अर्ग्युमेंट कोणताही ऑब्जेक्ट असू शकते; दुसरे अर्ग्युमेंट ॲट्रिब्युट चे नाव सांगणारी *स्ट्रिंग* असते.

तुम्ही `try` स्टेटमेंट वापरूनसुद्धा ऑब्जेक्टमध्ये तुम्हाला पाहिजे तो ॲट्रिब्युट आहे का हे तपासू शकता:

```
try:
    x = p.x
except AttributeError:
    x = 0
```

ही पद्धत वापरून वेगवेगळ्या टाइप्सवर चालणारी फंक्शन्स लिहिणे सोपे जात; ह्यावर पुढील चर्चा आपण विभाग १७.९ मध्ये करूच.

१५.८ शब्दार्थ

क्लास (class): एक प्रोग्रामर-परिभाषित टाइप. क्लास डेफनिशन एक नवीन क्लास ऑब्जेक्ट बनवते.

क्लास ऑब्जेक्ट (class object): प्रोग्रामर-परिभाषित टाइप विषयी माहिती ठेवणारा ऑब्जेक्ट. क्लास ऑब्जेक्ट वापरून त्या टाइपचे इन्स्टन्सेस बनवता येतात.

इन्स्टन्स (instance): एका क्लासचा ऑब्जेक्ट.

इन्स्टॅन्शियेट (instantiate): नवीन ऑब्जेक्ट बनवणे.

ॲट्रिब्युट (attribute): एका ऑब्जेक्टशी संलग्न (त्यामधील) नाव असलेली व्हॅल्यू.

एम्बेडेड ऑब्जेक्ट (embedded object): असा ऑब्जेक्ट जो दुसऱ्या ऑब्जेक्टचा ॲट्रिब्युट असतो.

शॅलो कॉपी (shallow copy, उथळ प्रत): ऑब्जेक्टमधील डेटा त्यातील एम्बेडेड ऑब्जेक्ट्स दर्शवणाऱ्या रेफ्रन्सेससहित कॉपी करणे; `copy` मोड्युलमधील `copy` फंक्शन हे इंप्लेमेंट करते.

डीप कॉपी (deep copy, खोल प्रत): ऑब्जेक्टमधील डेटा, त्यातील एम्बेडेड ऑब्जेक्ट्स, त्यातील एम्बेडेड ऑब्जेक्ट्स, आणि असेच पुढे...कॉपी करणे; `copy` मोड्युलमधील `deepcopy` फंक्शन हे इंप्लेमेंट करते.

ऑब्जेक्ट डायग्राम (object diagram): ऑब्जेक्ट्स, त्यांचे ॲट्रिब्युट्स, आणि त्यांच्या ॲट्रिब्युट्सच्या व्हॅल्यूझ दाखवणारी आकृती.

१५.९ प्रश्नसंच (Exercises)

प्रश्न १५.१. `Circle` *नावाच्या क्लासची डेफनिशन लिहा. ह्या क्लासचे* `center` *आणि* `radius` *हे ॲट्रिब्युट असतील;* `center` *हा* *Point* *ऑब्जेक्ट असेल आणि* `radius` *हा संख्या.*

केंद्र $(150, 100)$ *असलेले आणि त्रिज्या* 75 *असलेले एक वर्तुळ दर्शवणारा एक* *Circle* *ऑब्जेक्ट इन्स्टॅन्शियेट करा.*

एक *Circle* *आणि एक* *Point* *घेऊन जर तो बिंदू त्या वर्तुळाच्या आत किंवा त्या वर्तुळावर (परीघावर) असेल तर* `True` *रिटर्न करणारे* `point_in_circle` *नावाचे फंक्शन लिहा.*

एक *Circle* *आणि एक* *Rectangle* *घेऊन तो आयत जर पूर्णपणे त्या वर्तुळामध्ये (परीघासह) असेल तर* `True` *रिटर्न करणारे* `rect_in_circle` *नावाचे फंक्शन लिहा.*

एक *Circle* *आणि एक* *Rectangle* *घेऊन जर त्या आयताचा कोणताही शिरोबिंदू त्या वर्तुळामध्ये असेल तर* `True` *रिटर्न करणारे* `rect_circle_overlap` *नावाचे फंक्शन लिहा. किंवा, थोडा अजून आव्हानात्मक प्रश्न: आयताचा कोणताही भाग वर्तुळामध्ये असेल तर* `True` *रिटर्न करा.*

उत्तर: *http://thinkpython2.com/code/Circle.py*

प्रश्न १५.२. *एक* *Turtle* *ऑब्जेक्ट आणि एक* *Rectangle* *घेऊन, तो* *Turtle* *वापरून तो* *Rectangle* *काढणारे* `draw_rect` *नावाचे फंक्शन लिहा. तुम्ही* *Turtle* *ऑब्जेक्ट्स वापरणारी उदाहरणे प्रक्रण ४ मध्ये बघू शकता.*

एक *Turtle* *आणि एक* *Circle* *घेऊन ते वर्तुळ काढणारे* `draw_circle` *नावाचे फंक्शन लिहा.*

उत्तर: *http://thinkpython2.com/code/draw.py*.

प्रकरण १६

क्लास आणि फंक्शन (Class and function)

नवीन टाइप कसा बनवायचा हे आपण शिकलो; प्रोग्रामर-परिभाषित ऑब्जेक्ट्स घेणारी आणि रिटर्न करणारी फंक्शन्स लिहिणे ही पुढची पायरी आहे. ह्या प्रकरणात आपण 'फंक्शनल प्रोग्रामिंग शैली' ('functional programming style') आणि दोन नवीन प्रोग्राम डव्हेलपमेंट प्लान्स शिकणार आहोत.

ह्या प्रकरणातील कोड-ची उदाहरणे पुढील लिंकवर उपलब्ध आहेत: http://thinkpython2.com/code/Time1.py.

प्रश्नांची उत्तरे पुढील लिंकवर उपलब्ध आहेत: http://thinkpython2.com/code/Time1_soln.py.

१६.१ Time

प्रोग्रामर-परिभाषित टाइपचे अजून एक उदाहरण म्हणून आपण `Time` नावाचा दिवसाची वेळ ठेवणारा क्लास बनवूया. क्लास डेफनिशन अशी दिसते:

```
class Time:
    """Represents the time of day.

    attributes: hour, minute, second
    """
```

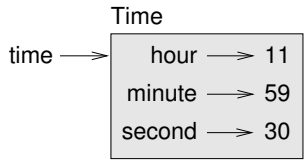
आपण एक नवीन `Time` ऑब्जेक्ट तयार करून तास, मिनिटे, आणि सेकंद ह्यांच्यासाठी ॲट्रिब्युट्स टाकू शकतो:

```
time = Time()
time.hour = 11
time.minute = 59
time.second = 30
```

`Time` च्या ऑब्जेक्टची स्टेट डायग्राम आकृती १६.१ मध्ये दाखवली आहे.

सराव म्हणून `print_time` नावाचे फंक्शन लिहा जे एक Time ऑब्जेक्ट घेऊन `hour:minute:second` ह्या स्वरुपात दाखवते. टीप: `'%.2d'` हा फॉरमॅट सीक्वेन्स वापरून तुम्ही एक इंटिजर कमीतकमी २ आकडे वापरून प्रिंट करू शकता, गरज पडली तर आधी शून्य लावून.

दोन Time ऑब्जेक्ट्स `t1` आणि `t2` घेऊन `t1` ही वेळ जर `t2` नंतर असेल तर `True` नाही तर `False` रिटर्न करणारे `is_after` नावाचे फंक्शन लिहा. आव्हान: हे `if` न वापरता करा.



आकृती १६.१: Object diagram.

१६.२ शुद्ध फंक्शन (Pure functions)

पुढील काही विभागांमध्ये आपण दोन फंक्शन्स लिहिणार आहोत जी Time व्हॅल्यूझची बेरीज करतात. ती दोन प्रकारची फंक्शन्स दाखवतात: शुद्ध फंक्शन आणि बदल-करणारे (modifier). त्यांद्वारे आपण **प्रोटोटाइप आणि पॅच** (**prototype and patch**, नमुना-बनवा-दुरुस्ती-करा) हा प्रोग्राम डव्हेलपमेंट प्लानसुद्धा बघणार आहोत, ज्यात क्लिष्ट प्रॉब्लेम सोडवण्यासाठी साध्या नमुन्याने सुरुवात करून त्यात हळूहळू भर घालून गुंतागुंत हाताळण्याचा मार्ग सुचवला आहे.

खाली `add_time` चा साधा नमुना आहे:

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second
    return sum
```

हे फंक्शन एक नवीन Time ऑब्जेक्ट बनवून आणि त्याचे ॲट्रिब्युट्स इनिशलाइझ करून त्याचा रेफ्रन्स रिटर्न करते. ह्याला **शुद्ध फंक्शन** (**pure function**) म्हणतात कारण ते त्याला पाठवलेला कोणताही ऑब्जेक्ट बदलत नाही आणि त्याचा एक व्हॅल्यू रिटर्न करण्याशिवाय इतर कोणताही परिणाम होत नाही, जसे व्हॅल्यू दाखवणे किंवा युझरकडून इनपुट घेणे.

हे फंक्शन टेस्ट करण्यासाठी आपण दोन Time ऑब्जेक्ट्स बनवूया: `start` मध्ये चित्रपट, उदा., *Monty Python and the Holy Grail*, सुरू होण्याची वेळ, आणि `duration` मध्ये चित्रपटाची लांबी, जी १ तास आणि ३५ मिनिटे आहे.

चित्रपट कधी संपेल हे `add_time` शोधते.

`add_time` figures out when the movie will be done.

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second =  0

>>> duration = Time()
>>> duration.hour = 1
>>> duration.minute = 35
>>> duration.second = 0

>>> done = add_time(start, duration)
>>> print_time(done)
10:80:00
```

`10:80:00` हे उत्तर मिळण्याची तुमची अपेक्षा नसेल. गडबड अशी आहे की हे फंक्शन सेकंदांची किंवा मिनिटांची बेरीज ६० किंवा जास्त होणाऱ्या परिस्थितींना हाताळत नाही. तसे जेव्हा होते तेव्हा आपल्याला 'हातचा' घेऊन अतिरिक्त सेकंद मिनिटांमध्ये किंवा अतिरिक्त मिनिटे तासांमध्ये न्यावी ('carry' करावी) लागतात.

सुधारित रूप असे आहे:

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second

    if sum.second >= 60:
        sum.second -= 60
        sum.minute += 1

    if sum.minute >= 60:
        sum.minute -= 60
        sum.hour += 1

    return sum
```

हे फंक्शन जरी बरोबर असले तरी ते मोठे होत चालले आहे. लहान पर्याय आपण पुढे पाहूच.

१६.३ बदल-करणारे फंक्शन (Modifiers)

कधीकधी परॅमीटर्समध्ये आलेले ऑब्जेक्ट्स बदलणे फंक्शनसाठी उपयोगी असते. त्याठिकाणी, केलेले बदल हे कॉलरला दिसतात. अशा प्रकारे चालणाऱ्या फंक्शनला **बदल-करणारे** फंक्शन (**modifier**, मॉडिफायर) म्हणतात.

`increment`, जे पाठवले तितके सेकंद Time ऑब्जेक्टमध्ये मिळवते हे सरळपणे एक मॉडिफायर (modifier) म्हणून लिहिले जाऊ शकते. खाली त्याची एक कच्ची आवृत्ती आहे:

```
def increment(time, seconds):
    time.second += seconds

    if time.second >= 60:
        time.second -= 60
        time.minute += 1

    if time.minute >= 60:
        time.minute -= 60
        time.hour += 1
```

पहिली ओळ साधे ऑपरेशन करते; बाकीचे आपण आधी पाहिलेली विशेष परिस्थिती हाताळते.

हे फंक्शन बरोबर आहे का? जर `seconds` ६० पेक्षा खूप जास्ती असेल तर?

तेव्हा हातचा एकच घेणे पुरेसे नाही; आपल्याला ते `time.second` ६० पेक्षा कमी होईपर्यंत करत रहावे लागेल. एक मार्ग असा आहे की `if` स्टेटमेंट्सना `while` स्टेटमेंट्सनी बदलणे. त्याने फंक्शन बरोबर होईल पण अप्रभावी (संथ). सराव म्हणून बरोबर असलेले पण कोणतेही लूप *न* वापरणारे `increment` फंक्शन लिहा.

मॉडिफायरने जे करता येते ते शुद्ध फंक्शनने पण करता येते. खरे तर काही प्रोग्रामिंग लँग्वेजेसमध्ये फक्त शुद्ध फंक्शनचीच सोय आहे. काही पुरावे हे दाखवतात की शुद्ध फंक्शन्स वापरणारे प्रोग्राम्स लवकर लिहून होतात आणि त्यांच्यात एरर येण्याची शक्यता मॉडिफायर वापरणाऱ्या प्रोग्राम्सपेक्षा कमी असते. पण मॉडिफायर कधीकधी उपयोगी पडते, आणि फंक्शनल प्रोग्राम साधारणपणे कमी वेगाने चालतात.

साधारणपणे सांगणे हे आहे की तुम्ही जेव्हा जेव्हा योग्य आहे तेव्हा शुद्ध फंक्शन लिहा, आणि मॉडिफायर तेव्हाच वापरा जेव्हा त्याचा ठोस फायदा दिसेल. ह्या पद्धतीला **फंक्शनल प्रोग्रामिंग शैली** (**functional programming style**) म्हणतात.

सराव म्हणून, `increment` चे एक 'शुद्ध' स्वरूप लिहा जे परॅमीटर बदलण्याऐवजी एक नवीन Time ऑब्जेक्ट बनवून रिटर्न करते.

१६.४ नमुना-बनवणे की योजना-आखणे (Prototyping versus planning)

आपण जो डेव्हेलपमेंट प्लान बघणार आहोत त्याला 'प्रोटोटाइप आणि पॅच' ('prototype and patch', नमुना-बनवा-दुरुस्ती-करा) असे म्हणतात. प्रत्येक फंक्शनसाठी आपण एक साधे कॅल्क्युलेशन करणारा नमुना लिहिला आणि टेस्ट केला, आणि मार्गात आलेले एरर्स दुरूस्त केले.

हा मार्ग प्रभावी ठरू शकतो, विशेषत: जेव्हा तुम्हाला प्रॉब्लेमची पूर्ण समज नसेल तेव्हा. पण अशा वाढीव दुरुस्त्यांमुळे असा कोड तयार होऊ शकतो जो असतो १) अनावश्यक आणि गुंतागुंतीचा—कारण त्यात अनेक विशेष केसेस (cases) असतात, आणि २) अविश्वसनीय—कारण तुम्हाला बहुतांश महत्त्वाचे एरर्स मिळाले आहेत की नाही ह्याची खात्री देता येत नाही.

एक पर्याय आहे **डिझाइन्ड डेव्हेलपमेंट** (**designed development**), ज्यात प्रॉब्लेमचा वरील स्तरावरून (high level) विचार करून त्याचे अंतरंग ओळखले जाते आणि त्या अंतर्दृष्टीने उत्तर मिळवले जाते, ज्याने प्रोग्रामिंग सोपे होते. ह्याठिकाणी अंतर्दृष्टी अशी आहे की Time ऑब्जेक्ट हा खरे तर पाया ६० मधील तीन-अंकी संख्या आहे (पुढील लिंक बघा: http://en.wikipedia.org/wiki/Sexagesimal).

`second` ॲट्रिब्युट हा 'एकाचा रकाना ('ones column') आहे, `minute` ॲट्रिब्युट हा 'साठाचा रकाना ('sixties column') आहे, आणि `hour` ॲट्रिब्युट हा 'छत्तीसशेचा रकाना ('thirty-six hundreds column') आहे.

जेव्हा आपण `add_time` आणि carrying, addition with लिहिले, तेव्हा आपण खरे तर ६० पाया मध्ये बेरीज करत होतो, म्हणूनच आपल्याला एका ठिकाणाहून दुसऱ्या ठिकाणी हातचा न्यावा लागत होता.

हे निरीक्षण ह्या पूर्ण प्रॉब्लेमसाठीच अजून एक उपाय सुचवते—आपण Time ऑब्जेक्टचे इंटिजरमध्ये रुपांतर करून ह्या गोष्टीचा फायदा घेऊ शकतो की कॉंप्युटरला इंटिजरचे अंकगणित कसे करायचे हे माहीत आहे.

खालील फंक्शन Time चे इंटिजरमध्ये रुपांतर करते:

```
def time_to_int(time):
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

आणि खालील फंक्शन एका इंटिजरचे Time मध्ये रुपांतर करते (हे आठवा की `divmod` पहिल्या अर्ग्युमेंटला दुसऱ्याने भागून भागाकार आणि बाकी टपल स्वरुपात रिटर्न करते).

```
def int_to_time(seconds):
    time = Time()
    minutes, time.second = divmod(seconds, 60)
    time.hour, time.minute = divmod(minutes, 60)
    return time
```

ही फंक्शन्स बरोबर आहेत हे तपासण्यासाठी तुम्हाला थोडा विचार करावा लागेल आणि त्यांवर काही टेस्ट्स चालवाव्या लागतील. त्यांना टेस्ट करण्याचा एक मार्ग म्हणजे `time_to_int(int_to_time(x)) == x` हे `x` च्या शक्य तितक्या व्हॅल्यूझसाठी तपासणे. हे एक सुसंगतता तपासणी (consistency check) चे उदाहरण आहे.

ती बरोबर असल्याची तुमची खात्री पटली की तुम्ही ती वापरून `add_time` परत लिहू शकता:

```
def add_time(t1, t2):
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

ही आवृत्ती आधीपेक्षा संक्षिप्त आणि तपासायला सोपी आहे. सराव म्हणून `time_to_int` आणि `int_to_time` वापरून `increment` परत लिहा.

काही अंशी, पाया ६० ते पाया १० आणि परत रुपांतर हे प्रत्यक्ष वेळांवर क्रिया करण्यापेक्षा अवघड आहे. पाया रुपांतर हे जास्ती संकल्पनात्मक (abstract) असून आपली वेळेविषयीची कल्पना जास्ती ठोस आहे (साहजिक आहे, वेळेनुसार जग चालते पण पाया रुपांतर ही बहुतांश वाचकांसाठी काहीशी गूढ कल्पना आहे).

पण जर आपल्याला वेळेला ६० पायातल्या संख्यासारखे बघण्याची कल्पना सुचली आणि आपण रुपांतर करणारी फंक्शन्स लिहिण्याचे काम केले तर आपल्याला असा प्रोग्राम मिळतो जो संक्षिप्त, समजायला आणि डीबग करायला सोपा, आणि जास्ती विश्वसनीय असतो.

त्यात नंतर नवीन वैशिष्ट्यांची भर घालणे देखील सोपे पडते. उदा., समजा दोन Time ऑब्जेक्ट्सची वजाबाकी करून त्यांच्यातले अंतर शोधायचे आहे. साधा मार्ग म्हणजे वजाबाकी (हातचा धरून) इंप्लेमेंट करणे. पण रुपांतर फंक्शन्स वापरणे सोपे ठरेल आणि ते बरोबर असण्याची शक्यतादेखील जास्ती असेल.

उपरोधात्मक असले तरी कधीकधी एखादा प्रॉब्लेम अवघड (किंवा जास्ती व्यापक) केल्याने तो सोपा होतो (कारण त्यात कमी विशेष केसेस आणि त्यामुळे एरर यायला कमी शक्यता असते).

१६.५ डीबगिंग (Debugging)

Time ऑब्जेक्ट उचित-स्वरुपात असते जेव्हा `minute` आणि `second` च्या व्हॅल्यूझ 0 आणि 60 च्या मध्ये (0 सहित आणि 60 सोडून) असतात, आणि जेव्हा `hour` ऋण नसते. `hour` आणि `minute` इंटिजर असले पाहिजेत पण `second` अपूर्णांक असेल तर चालेल.

अशा मागण्यांना **इनव्हेरिअंट** (**invariant**, निश्चल) म्हणतात कारण त्या नेहमी सत्य असल्या पाहिजेत. दुसऱ्या शब्दांत सांगायचे तर: जर त्या सत्य नसल्या तर काही तरी चूक झाली आहे.

इनव्हेरिअंट्स तपासणारा कोड लिहिल्याने एरर आणि त्याचे कारण शोधायला मदत होते. उदा., तुम्ही `valid_time` सारखे एक फंक्शन लिहू शकता जे एक Time ऑब्जेक्ट घेऊन तो कोणत्याही इनव्हेरिअंटचे उल्लंघन करत असेल तर `False` रिटर्न करते:

```
def valid_time(time):
    if time.hour < 0 or time.minute < 0 or time.second < 0:
        return False
    if time.minute >= 60 or time.second >= 60:
        return False
    return True
```

मग, प्रत्येक फंक्शनच्या सुरुवातीला तुम्ही अर्ग्युमेंट्स वैध आहेत की नाही ते तपासू शकता:

```
def add_time(t1, t2):
    if not valid_time(t1) or not valid_time(t2):
        raise ValueError('invalid Time object in add_time')
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

किंवा तुम्ही एक **ॲसर्ट स्टेटमेंट** (**assert statement**, खात्री करणारे स्टेटमेंट) वापरू शकता जे दिलेले इनव्हेरिअंट तपासून त्याचे उल्लंघन झाल्यास एक्सेप्शन रेय्झ (raise) करते:

```
def add_time(t1, t2):
    assert valid_time(t1) and valid_time(t2)
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

`assert` स्टेटमेंट फायदेशीर असतात कारण ते सामान्य परिस्थितीत चालणारा कोड आणि एरर तपासणारा कोड ह्यांच्यातला फरक दर्शवतात.

१६.६ शब्दार्थ

प्रोटोटाइप आणि पॅच (prototype and patch, नमुना-बनवा-दुरुस्ती-करा): एक डेव्हेलपमेंट प्लान ज्यात प्रोग्रामचा एक कच्चा मसूदा लिहिला जातो, टेस्टिंग केली जाते, आणि जसे एरर्स मिळतील तसे ते दुरूस्त केले जातात.

डिझाइन्ड डेव्हेलपमेंट (designed development): एक डेव्हेलपमेंट प्लान ज्यात हळूहळू वाढीव डेव्हेलपमेंट किंवा नमुने बनवण्याऐवजी प्रॉब्लेमचा वरील स्तरावरून (high level) विचार करून त्याचे अंतरंग ओळखले जाते आणि ज्यात जास्ती योजना आखल्या जातात

शुद्ध फंक्शन (pure function): असे फंक्शन जे त्याला अर्ग्युमेंट्स म्हणून मिळालेल्या कोणत्याही ऑब्जेक्ट्सना बदलत नाही. बहुतांश शुद्ध फंक्शन्स फलदायी (fruitful) असतात.

मॉडिफायर (modifier, बदल करणारे फंक्शन): असे फंक्शन जे त्याला अर्ग्युमेंट्स म्हणून मिळालेल्या काही ऑब्जेक्ट्समध्ये बदल घडवून आणते. बहुतांश शुद्ध फंक्शन्स व्हॉय्ड (void) असतात, म्हणजे ते `None` रिटर्न करतात .

फंक्शनल प्रोग्रामिंग शैली (functional programming style): प्रोग्राम डिझाइनची अशी शैली ज्यात बहुसंख्य फंक्शन्स शुद्ध असतात.

इनव्हेरिअंट (invariant): अशी अट जी प्रोग्रामच्या एक्सेक्युशनच्या दरम्यान नेहमी सत्य असली पाहिजे.

ॲसर्ट स्टेटमेंट (assert statement): कंडिशन तपासून ती असत्य असेल तर एक्सेप्शन रेय्झ (raise) करणारे स्टेटमेंट.

१६.७ प्रश्नसंच (Exercises)

ह्या प्रकरणातील कोड-ची उदाहरणे पुढील लिंकवर उपलब्ध आहेत: http://thinkpython2.com/code/Time1.py.

प्रश्नांची उत्तरे पुढील लिंकवर उपलब्ध आहेत: http://thinkpython2.com/code/Time1_soln.py.

प्रश्न १६.१. *एक Time ऑब्जेक्ट आणि एक संख्या घेऊन त्यांचा गुणाकार असणारा एक नवीन Time ऑब्जेक्ट बनवून रिटर्न करणारे* `mul_time` *नावाचे फंक्शन लिहा.*

नंतर, `mul_time` *वापरून, शर्यत पूर्ण करण्याची वेळ व्यक्त करणारा एक Time ऑब्जेक्ट आणि शर्यतीचे अंतर मैलात दर्शवणारी एक संख्या, हे दोन अर्ग्युमेंट्स घेणारे एक फंक्शन लिहा जे सरासरी वेग (वेळ प्रति मैल एककात) दर्शवणारा Time ऑब्जेक्ट रिटर्न करते.*

प्रश्न १६.२. `datetime` *मोड्युल* `time` *ऑब्जेक्ट्स पुरवते जे थोडेफार ह्या प्रकरणातील Time ऑब्जेक्ट्स सारखे असतात, पण ते मेथड्स आणि ऑपरेटर्स चा संपन्न संग्रह पुरवतात. पुढील लिंकवर त्याचे डॉक्युमेंटेशन वाचा: http://docs.python.org/3/library/datetime.html.*

१. `datetime` *मोड्युल वापरून सध्याची तारीख घेऊन वार दाखवणारा एक प्रोग्राम लिहा.*

२. *वाढदिवस इनपुट म्हणून घेऊन युझरचे वय आणि पुढच्या वाढदिवसाला किती दिवस, तास, मिनिटे, आणि सेकंद बाकी आहेत हे प्रिंट करणारा प्रोग्राम लिहा.*

३. *दोन वेगळ्या दिवशी जन्मलेल्या लोकांसाठी एक दिवस असा असतो की त्यांपैकी एका व्यक्तीचे वय दुसऱ्याच्या वयाच्या दुप्पट असते. तो त्यांचा Double Day असतो. दोन वाढदिवस घेऊन त्यांचा Double Day प्रिंट करणारा प्रोग्राम लिहा.*

४. *अजून चॅलेंज (challenge, आव्हान) पाहिजे असेल तर ह्याचीच अजून व्यापक आवृत्ती लिहा: तो दिवस शोधा जेव्हा एक व्यक्ती दुसरीच्या n पट मोठी असेल .*

उत्तर: http://thinkpython2.com/code/double.py.

प्रकरण १७

क्लास आणि मेथड (Class and method)

आपण जरी पायथॉनची काही ऑब्जेक्ट-ओरिएंटेड वैशिष्ट्ये वापरत असलो तरी मागच्या दोन प्रकरणातील प्रोग्राम्स वास्तविक पाहता ऑब्जेक्ट-ओरिएंटेड नव्हते कारण ते प्रोग्रामर-परिभाषित टाइप्स आणि त्यांवर चालणारी फंक्शन्स ह्यांतील संबंध प्रस्थापित करत नाहीत. त्या फंक्शन्सचे मेथड्समध्ये रुपांतर करुन ते संबंध सुस्पष्ट करणे ही पुढची पायरी आहे.

ह्या प्रकरणातील कोड-ची उदाहरणे पुढील लिंकवर उपलब्ध आहेत: http://thinkpython2.com/code/Time2.py.

आणि प्रश्नांची उत्तरे पुढील लिंकवर उपलब्ध आहेत: http://thinkpython2.com/code/Point2_soln.py

१७.१ ऑब्जेक्ट-ओरिएंटेड वैशिष्ट्ये (Object-oriented features)

पायथॉन ही प्रोग्रामिंग लँग्वेज एक **ऑब्जेक्ट-ओरिएंटेड प्रोग्रामिंग लँग्वेज** (**object-oriented programming language**) आहे, म्हणजेच त्याला चालना देणारी वैशिष्ट्ये पायथॉनमध्ये आहेत. ऑब्जेक्ट-ओरिएंटेड प्रोग्रामिंग म्हणजे साधारणपणे:

- प्रोग्राममध्ये क्लास आणि मेथड डेफनिशन्स वापरणे.
- बहुतांश कॉंप्युटेशन हे ऑब्जेक्ट्सवरील प्रक्रियांनी व्यक्त करणे.
- ऑब्जेक्ट्सद्वारे ठोस गोष्टी आणि मेथड्सद्वारे त्या गोष्टींमधील संबंध व्यक्त करणे.

उदा., प्रकरण १६ मध्ये दिलेला `Time` क्लास ज्याप्रकारे लोक वेळेची नोंद ठेवतात त्याच्याशी, आणि आपण दिलेली फंक्शन्स ज्याप्रकारे लोक वेळांवर प्रक्रिया करतात त्याच्याशी प्रत्यक्षरित्या संबंधित आहे. त्याचप्रमाणे प्रकरण १५ मधील `Point` आणि `Rectangle` क्लासेस बिंदू आणि आयत ह्या गणितीय संकल्पनांशी प्रत्यक्षरित्या संबंधित आहेत.

आतापर्यंत, आपण पायथॉनच्या ऑब्जेक्ट-ओरिएंटेड प्रोग्रामिंगला चालना देणाऱ्या वैशिष्ट्यांचा फायदा करून घेतला नाहीये. ह्या वैशिष्ट्यांवाचून खरे तर आपले काही अडले नाहीये; सर्वसाधारणपणे त्यात आपण आधीच केलेल्या गोष्टींसाठी पर्यायी सिंटॅक्स आहे. पण अनेक ठिकाणी, हा पर्याय जास्ती संक्षिप्त असतो आणि प्रोग्रामची संरचना जास्ती अचूकपणे व्यक्त करतो.

उदा., `Time1.py` मध्ये क्लास डेफनिशन आणि त्यानंतरच्या फंक्शन डेफनिशन्समध्ये स्पष्ट नाते नाहीये. थोड्या तपासानंतर असे लक्षात येते की प्रत्येक फंक्शन कमीतकमी एक Time ऑब्जेक्ट अर्ग्युमेंट म्हणून घेते.

हे निरीक्षणच **मेथड** (**method**) मागचा प्रेरणास्रोत आहे; मेथड म्हणजे एका विशिष्ठ क्लासशी संलग्न फंक्शन. आपण स्ट्रिंग, लिस्ट, डिक्शनरी, आणि टपल ह्यांच्या मेथड्स बघितल्यात. ह्या प्रकरणात आपण प्रोग्रामर-परिभाषित टाइपसाठी मेथड डिफाइन करणार आहोत.

मेथड ही फंक्शनसारखीच असते पण तिचा सिंटॅक्स दोन प्रकारे वेगळा असतो:

- मेथड डेफनिशन एका क्लास डेफनिशनमध्ये दिली जाते जेणेकरून तो क्लास आणि त्या मेथडमधील संबंध सुस्पष्ट होईल.
- मेथड इन्व्होक करण्याचा सिंटॅक्स फंक्शन कॉलच्या सिंटॅक्सहून वेगळा आहे.

पुढील काही विभागांमध्ये आपण मागच्या दोन प्रकरणांतील फंक्शन्स घेऊन त्यांचे मेथड्समध्ये रुपांतर करणार आहोत. हे रुपांतर पूर्णपणे यंत्रवत[१] असेल; तुम्ही हे काही सरळ सूचनांच्या अनुसरणाने करू शकता. जर तुम्हाला एकातून दुसऱ्यात रुपांतर करणे सोपे जात असेल तर तुम्ही तुमचे काम पार पाडण्यासाठी सर्वोत्तम मार्ग निवडू शकता.

१७.२ ऑब्जेक्ट प्रिंट करणे (Printing objects)

प्रकरण १६ मध्ये आपण `Time` नावाचा क्लास बघितला आणि विभाग १६.१ मध्ये तुम्ही `print_time` नावाचे फंक्शन लिहिले:

```
class Time:
    """Represents the time of day."""

def print_time(time):
    print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

हे फंक्शन कॉल करण्यासाठी तुम्हाला एक Time ऑब्जेक्ट अर्ग्युमेंट म्हणून पाठवावा लागतो:

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 00
>>> print_time(start)
09:45:00
```

`print_time` ला मेथड बनवण्यासाठी आपल्याला फक्त ती फंक्शन डेफनिशन क्लास डेफनिशनच्या आतमध्ये हलवायची आहे. इंडेंटेशन (म्हणजेच प्रत्येक ओळीच्या सुरुवातीला सोडलेल्या जागे) मधील बदल लक्षात घ्या.

```
class Time:
    def print_time(time):
        print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

आता `print_time` कॉल करण्याचे दोन मार्ग आहेत. पहिला (कमी वापरला जाणारा) मार्ग म्हणजे फंक्शन सिंटॅक्स वापरणे:

```
>>> Time.print_time(start)
09:45:00
```

डॉट नोटेशनच्या ह्या वापरात `Time` हे क्लासचे नाव आहे आणि `print_time` हे मेथडचे नाव आहे. `start` अर्ग्युमेंट म्हणून पाठवले जाते.

दुसरा (जास्ती संक्षिप्त) मार्ग म्हणजे मेथड सिंटॅक्स वापरणे:

```
>>> start.print_time()
09:45:00
```

[१] (अनुवादकाची टिप्पणी: आधी सांगितल्याप्रमाणे, ह्याठिकाणी यंत्रवत म्हणजे तंतोतंतपणे पार पाडणे, यंत्रांशी संबंधित नाही.)

डॉट नोटेशनच्या ह्या वापरात print_time हे (पुन्हा) मेथडचे नाव आहे आणि ज्या ऑब्जेक्टवर ही मेथड इन्व्होक केली आहे तो start ने दर्शवला जातो, आणि त्याला **सब्जेक्ट** (**subject**) असे म्हणतात. जसे इंग्रजीमध्ये एका वाक्याचा subject म्हणजे ते वाक्य ज्यावर आहे ते, तसेच मेथड इन्व्होक करतानाचा subject म्हणजे ती मेथड ज्यावर आहे ते.

मेथडच्या आतमध्ये, सब्जेक्ट पहिल्या परॅमीटरला असाइन केला जातो, म्हणजे ह्याठिकाणी start time ला असाइन होते.

मेथडच्या पहिल्या परॅमीटरचे नाव self असे ठेवण्याची पद्धत आहे, तर print_time अशी लिहिणे रास्त ठरेल:

```
class Time:
    def print_time(self):
        print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))
```

ह्या पद्धतीमागचे कारण हे एक अलिखित रूपक (implicit metaphor) आहे:

- फंक्शन कॉलचा सिंटॅक्स print_time(start) सुचवतो की फंक्शन हे कर्ता आहे. हा सिंटॅक्स म्हणतो, "अरे print_time(start)! हा तुझ्यासाठी प्रिंट करायला ऑब्जेक्ट आहे, घे."
- ऑब्जेक्ट-ओरिएंटेड प्रोग्रामिंगमध्ये ऑब्जेक्ट कर्ता असतो. आणि start.print_time() असे मेथड इन्व्होक करणे म्हणजे म्हणणे "अरे start! कृपया स्वतःला प्रिंट कर."

दृष्टीकोनातील हा बदल कदाचित जास्ती शिष्टाचारयुक्त असेल, पण हे स्पष्ट नाही की तो फायदेशीर आहे. आपण बघितलेल्या उदाहरणात तो नसेलही. पण कधीकधी फंक्शनवरील जबाबदारी ऑब्जेक्टवर हलवण्यामुळे जास्ती चांगली फंक्शन्स (किंवा मेथड्स) लिहिणे शक्य होते आणि कोड-चा पुनर्वापर करणे आणि कोड मेन्टेन (maintain) करणे सोपे जाते.

सराव म्हणून (विभाग १६.४ मधील) time_to_int पुन्हा मेथडस्वरुपात लिहा. तुम्हाला int_to_time ची पण मेथड लिहिण्याचा मोह होईल पण ते काहीसे अर्थहीन आहे कारण ती इन्व्होक करायला ऑब्जेक्टच नसेल.

१७.३ अजून एक उदाहरण (Another example)

खाली (विभाग १६.३ मधील) increment हे मेथडस्वरुपात लिहिले आहे:

```
# inside class Time:

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

ह्यात असे गृहीत धरले आहे की time_to_int ही मेथड आहे. आणि ह्याची नोंद घ्या की हे शुद्ध फंक्शन (pure function) आहे, मॉडिफायर (modifier) नाही.

तुम्ही increment असे इन्व्होक करू शकता:

```
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
>>> end.print_time()
10:07:17
```

start हा सब्जेक्ट पहिल्या परॅमीटरला म्हणजे self ला असाइन होतो; 1337 हे अर्ग्युमेंट दुसऱ्या परॅमीटरला म्हणजे seconds ला असाइन होते.

ह्या कार्यतंत्रामुळे गोंधळ होऊ शकतो, विशेषतः तुम्हाला एरर मिळाला तर. उदा., जर तुम्ही increment दोन अर्ग्युमेंट्सने इन्व्होक केले तर तुम्हाला मिळेल:

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes 2 positional arguments but 3 were given
```

हा एरर मेसेज सुरुवातीला संभ्रमात टाकू शकतो कारण कंसात फक्त दोनच अर्ग्युमेंट्स आहेत. पण सब्जेक्टलासुद्धा अर्ग्युमेंटच गृहीत धरले जाते, म्हणजे एकूण तीन आहेत.

आणि हो, **पोझिशनल अर्ग्युमेंट** (**positional argument**) म्हणजे असे अर्ग्युमेंट ज्याला (संबंधित) परॅमीटरचे नाव जोडलेले नसते; म्हणजे, ते कीवर्ड अर्ग्युमेंट (keyword argument) नसते. खालील फंक्शन कॉलमध्ये:

```
sketch(parrot, cage, dead=True)
```

`parrot` आणि `cage` ही पोझिशनल आहेत आणि `dead` हे कीवर्ड अर्ग्युमेंट आहे.

१७.४ थोडे अवघड उदाहरण (A more complicated example)

विभाग १६.१ मधील `is_after` चे मेथडस्वरूप लिहिणे थोडे अवघड आहे कारण ते दोन Time ऑब्जेक्ट्स घेते. ह्याठिकाणी पहिल्या परॅमीटरचे नाव `self` आणि दुसऱ्याचे नाव `other` ठेवण्याची पद्धत आहे:

```
# inside class Time:

    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

ही मेथड वापरण्यासाठी तुम्हाला ती एका ऑब्जेक्टवर इन्व्होक करावी लागेल आणि दुसरा ऑब्जेक्ट अर्ग्युमेंट म्हणून पाठवावा लागेल:

```
>>> end.is_after(start)
True
```

ह्या सिंटॅक्सबद्दल एक छान गोष्ट म्हणजे ते जवळजवळ इंग्रजीसारखेच आहे: 'end is after start?'

१७.५ init मेथड (The init method)

init मेथड ('initialization' चे संक्षिप्त रूप) ही एक विशेष मेथड असते जी कोणताही ऑब्जेक्ट इन्स्टॅन्शियेट (instantiate) होताना इन्व्होक होते. तिचे पूर्ण नाव `__init__` आहे—दोन अंडरस्कोर (underscore) कॅरेक्टर्स नंतर `init`, आणि नंतर अजून दोन अंडरस्कोर कॅरेक्टर्स. `Time` क्लासची init अशी असू शकते:

```
# inside class Time:

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
```

`__init__` च्या परॅमीटर्सची नावे आणि ॲट्रिब्युट्सची नावे सारखी ठेवणे कॉमन आहे. खालील स्टेटमेंट

```
        self.hour = hour
```

`hour` ह्या परॅमीटरची व्हॅल्यू `self` च्या ॲट्रिब्युटमध्ये ठेवते.

ह्याठिकाणी परॅमीटर्स पर्यायी आहेत, म्हणजे तुम्ही अर्ग्युमेंट्स *न* पाठवता `Time` कॉल केले तर तुम्हाला डीफॉल्ट व्हॅल्यूझ मिळतील.

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

जर तुम्ही एक अर्ग्युमेंट दिले तर ते hour ला ओव्हर्राइड (override) करते:

```
>>> time = Time (9)
>>> time.print_time()
09:00:00
```

जर तुम्ही दोन अर्ग्युमेंट्स दिली तर ती hour आणि minute ह्यांना ओव्हर्राइड करतात.

```
>>> time = Time(9, 45)
>>> time.print_time()
09:45:00
```

आणि जर तुम्ही तीन अर्ग्युमेंट्स दिली तर ती तिन्ही डीफॉल्ट व्हॅल्यूझ-ना ओव्हर्राइड करतात.

सराव म्हणून, Point साठी init मेथड लिहा जी x आणि y ही ऑप्शनल परॅमीटर्स घेते आणि त्यांना संबंधित ॲट्रिब्युट्सना असाइन करते.

१७.६ __str__ मेथड (The __str__ method)

__str__ ही __init__ सारखीच एक विशेष मेथड आहे; ती ऑब्जेक्टला स्ट्रिंगस्वरुपात व्यक्त करते.

उदा., Time ऑब्जेक्टसाठी str मेथड खाली दिली आहे:

```
# inside class Time:

    def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

जेव्हा तुम्ही एखादा ऑब्जेक्ट print करता, तेव्हा पायथॉन str मेथड इन्व्होक करतो:

```
>>> time = Time(9, 45)
>>> print(time)
09:45:00
```

नवीन क्लास लिहिताना पहिले ह्या दोन, म्हणजेच __init__ आणि __str__ मेथड्स लिहिणे ही एक चांगली सवय आहे; __init__ मुळे नवीन ऑब्जेक्ट बनवणे सोपे जाते, आणि __str__ चा डीबगिंगसाठी फायदा होतो.

सराव म्हणून Point साठी str मेथड लिहा. एक Point ऑब्जेक्ट बनवून प्रिंट करा.

१७.७ ऑपरेटर ओव्हरलोडिंग (Operator overloading)

इतर विशेष मेथड्स लिहून तुम्ही प्रोग्रामर-परिभाषित टाइपवे ऑपरेटरची क्रिया व्यक्त करू शकता. उदा., जर तुम्ही Time क्लास साठी __add__ नावाची मेथड लिहिली तर तुम्ही Time ऑब्जेक्ट्सवर + ऑपरेटर वापरू शकता.

डेफनिशन अशी असू शकते:

```
# inside class Time:

    def __add__(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)
```

आणि तुम्ही असा वापर करू शकता:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
```

जेव्हा तुम्ही Time ऑब्जेक्ट्सवर + ऑपरेटर वापरता, तेव्हा पायथॉन `__add__` इन्व्होक करतो. जेव्हा तुम्ही उत्तर प्रिंट करता, तेव्हा पायथॉन `__str__` इन्व्होक करतो. म्हणजे पडद्यामागे खूप काही होते!

एखाद्या ऑपरेटरची वर्तणूक बदलून त्याला प्रोग्रामर-परिभाषित टाइपवर चालवण्यायोग्य करणे ह्याला **ऑपरेटर ओव्हरलोडिंग (operator overloading)** असे म्हणतात. पायथॉनमधील प्रत्येक ऑपरेटरशी संलग्न एक विशेष मेथड आहे (जशी `__add__`). सविस्तर माहितीसाठी पुढील लिंक बघा: http://docs.python.org/3/reference/datamodel.html#specialnames.

सराव म्हणून Point क्लासमध्ये `add` मेथड लिहा.

१७.८ टाइप-वर आधारित हस्तांतरण (Type-based dispatch)

मागच्या विभागात आपण दोन Time ऑब्जेक्ट्सची बेरीज केली, पण तुम्हाला Time ऑब्जेक्टमध्ये इंटिजर मिळवण्याची सोय पाहिजे असू शकते. `__add__` च्या खालील सुधारित रुपात `other` चा टाइप तपासून `add_time` किंवा `increment` इन्व्होक केली जाते:

```
# inside class Time:

    def __add__(self, other):
        if isinstance(other, Time):
            return self.add_time(other)
        else:
            return self.increment(other)

    def add_time(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

`isinstance` हे बिल्ट-इन फंक्शन एक व्हॅल्यू आणि एक क्लास ऑब्जेक्ट घेऊन ती व्हॅल्यू जर त्या क्लासचा इन्स्टन्स (instance) असेल तर `True` रिटर्न करते.

जर `other` Time ऑब्जेक्ट असेल तर `__add__` ही `add_time` ला इन्व्होक करते. नाहीतर ती असे मानून चालते की परॅमीटर इंटिजर आहे आणि `increment` इन्व्होक करते. ह्या प्रक्रियेला **टाइप-वर आधारित हस्तांतरण (type-based dispatch)** असे म्हणतात, कारण कोणत्या मेथडला हस्तांतरण करायचे हे अर्ग्युमेंट्सच्या टाइप-वर ठरते.

खालील उदाहरणे + ऑपरेटर वेगळ्या टाइप्सवर वापरतात:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
>>> print(start + 1337)
10:07:17
```

दुर्दैवाने बेरजेचे हे इंप्लेमेंटेशन क्रमनिरपेक्ष (commutative) नाही. जर पहिला ऑपरँड इंटिजर असेल तर तुम्हाला मिळेल:

```
>>> print(1337 + start)
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

गडबड ही आहे की Time ऑब्जेक्टला इंटिजर मिळवण्यास सांगण्याऐवजी पायथॉन इंटिजरला सांगतोय की Time ऑब्जेक्ट मिळव, पण त्याला ते कसे करायचे ते माहीत नाही. पण ह्याचे एक कल्पक उत्तर आहे: `__radd__` ही विशेष मेथड, जी 'right-side add' चे संक्षिप्त रूप आहे (उजव्या बाजूने मिळवणे). जर Time ऑब्जेक्ट + ऑपरेटरच्या उजव्या बाजूला आला तर ही मेथड इन्व्होक केली जाते. डेफनिशन अशी आहे:

```
# inside class Time:

    def __radd__(self, other):
        return self.__add__(other)
```

आणि वापर असा होतो:

```
>>> print(1337 + start)
10:07:17
```

सराव म्हणून Point साठी एक `add` मेथड लिहा जी Point ऑब्जेक्टवर किंवा टपल-वर चालेल:

- जर दुसरा ऑपरँड Point असेल तर त्या मेथडने एक नवीन Point रिटर्न केला पाहिजे ज्याचा x-निर्देशक दोन्ही ऑपरँड्सच्या x-निर्देशकांची बेरीज असेल, आणि ह्याचप्रमाणे y-निर्देशक शोधला जाईल.
- जर दुसरा ऑपरँड टपल असेल तर त्या मेथडने टपलचा पहिला एलेमेंट x-निर्देशकात मिळवून आणि दुसरा एलेमेंट y-निर्देशकात मिळवून हे अनुक्रमे x आणि y निर्देशक असलेला एक नवीन Point रिटर्न केला पाहिजे.

१७.९ पॉलिमॉर्फिझम (Polymorphism)

टाइप-वर आधारित हस्तांतरण गरज पडल्यावर उपयोगी पडते, पण (सुदैवाने) त्याची नेहमी गरज पडत नाही. अनेकदा वेगळ्या टाइपच्या अर्ग्युमेंट्सवर बरोबर चालणारे फंक्शन लिहून तुम्ही ते टाळू शकता.

आपण स्ट्रिंगसाठी लिहिलेली अनेक फंक्शन्स इतर सीक्वेन्स टाइप्स साठी पण चालतात. उदा., विभाग ११.२ मध्ये आपण `histogram` वापरून दिलेल्या शब्दात प्रत्येक अक्षर किती वेळा येते हे मोजले.

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] = d[c]+1
    return d
```

हेच फंक्शन लिस्ट, टपल, आणि डिक्शनरीवरसुद्धा चालते, जर `s` चे एलेमेंट्स हॅशेबल (hashable) असतील तर, जेणेकरून ते `d` मध्ये key म्हणून वापरता येतील.

```
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}
```

अनेक टाइप्सवर चालणाऱ्या फंक्शन्सना **पॉलिमॉर्फिक** (**polymorphic**, बहुरूपी) म्हणतात. त्यामुळे कोड-चा पुनर्वापर सोयीस्कर होतो. उदा., `sum` हे बिल्ट-इन फंक्शन दिलेल्या सीक्वेन्सच्या एलेमेंट्सची बेरीज शोधते, आणि त्याची इतकीच अपेक्षा आहे की त्या सीक्वेन्सच्या एलेमेंट्सवर बेरीज (addition) करता आली पाहिजे.

आणि Time ऑब्जेक्ट्समध्ये `add` मेथड असल्यामुळे त्यांवर `sum` वापरता येते:

```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 41)
>>> t3 = Time(7, 37)
```

```
>>> total = sum([t1, t2, t3])
>>> print(total)
23:01:00
```

साधारणपणे, जर एखाद्या फंक्शनमधील सर्व ऑपरेशन्स दिलेल्या टाइप-वर चालत असतील, तर ते फंक्शन त्या टाइप-वर चालते.

नकळत घडलेले पॉलिमॉर्फिझम हे सर्वोत्तम असते, जेव्हा तुम्हाला हा शोध लागतो की तुम्ही आधी लिहिलेले एक फंक्शन तुम्ही ज्या टाइपबद्दल विचार पण नव्हता केला त्यावरही चालते.

१७.१० डीबगिंग (Debugging)

कोणत्याही ऑब्जेक्टमध्ये कधीही नवीन ॲट्रिब्युट्स टाकता येतात, पण जर तुमच्याकडे एकाच टाइपचे पण वेगवेगळे ॲट्रिब्युट्स असणारे दोन ऑब्जेक्ट्स असतील तर चुका होण्याची शक्यता वाढते. ऑब्जेक्टचे सर्व ॲट्रिब्युट्स एकदमच init मेथडमध्ये इनिशलाइझ करणे व्यवस्थितपणाचे लक्षण मानले जाते.

(आधी विभाग १५.७ मध्ये सांगितल्याप्रमाणे) जर तुम्हाला शंका असेल की ऑब्जेक्टला एक विशिष्ठ ॲट्रिब्युट आहे की नाही तर तुम्ही `hasattr` हे बिल्ट-इन फंक्शन वापरू शकता.

ॲट्रिब्युट्स बघण्याचा अजून एक मार्ग म्हणजे बिल्ट-इन फंक्शन `vars`, जे एक ऑब्जेक्ट घेऊन ॲट्रिब्युट्सची (स्ट्रिंगस्वरुपातील) नावे त्यांच्या व्हॅल्यूझ-ना मॅप करणारी डिक्शनरी रिटर्न करते:

```
>>> p = Point(3, 4)
>>> vars(p)
{'y': 4, 'x': 3}
```

डीबगिंगसाठी तुम्हाला खालील फंक्शन उपयोगी पडू शकते:

```
def print_attributes(obj):
    for attr in vars(obj):
        print(attr, getattr(obj, attr))
```

`print_attributes` डिक्शनरीमधील प्रत्येक ॲट्रिब्युटचे नाव आणि त्याची व्हॅल्यू प्रिंट करते.

आणि `getattr` हे बिल्ट-इन फंक्शन एक ऑब्जेक्ट आणि एका ॲट्रिब्युटचे स्ट्रिंगस्वरुपातील नाव घेऊन त्या ॲट्रिब्युटची व्हॅल्यू रिटर्न करते.

१७.११ इंटरफेस आणि इंप्लेमेंटेशन (Interface and implementation)

ऑब्जेक्ट-ओरिएंटेड डिझाइन चा एक मुख्य उद्देश सॉफ्टवेअर मेन्टेनेबल (maintainable) ठेवणे हा आहे, म्हणजेच अवतीभवतीची सिस्टम जसजशी बदलेल तसतसे तुम्ही जुळवून घेण्यासाठी प्रोग्राममध्ये (सोयीस्करपणे) बदल/सुधार करू शकता.

इंटरफेस आणि इंप्लेमेंटेशन वेगळे ठेवणे हे तत्त्व तो उद्देश साध्य करायला महत्त्वाचे ठरते. ऑब्जेक्टच्या दृष्टीने ह्याचा अर्थ असा की क्लास पुरवत असलेल्या मेथड्स ॲट्रिब्युट्स कसे व्यक्त केलेत ह्यावर अवलंबून नाही पाहिजे.

उदा., ह्या प्रकरणात आपण वेळेसाठी एक क्लास बनवला. ह्या क्लासमध्ये `time_to_int`, `is_after`, आणि `add_time` ह्या मेथड्स आहेत.

आपण त्या मेथड्स अनेकप्रकारे इंप्लेमेंट करू शकतो. इंप्लेमेंटेशनचा तपशील आपण वेळ कशी व्यक्त करतो ह्यावर अवलंबून आहे. ह्या प्रकरणात, `Time` ऑब्जेक्टचे ॲट्रिब्युट्स `hour`, `minute`, आणि `second` आहेत.

पर्याय म्हणून, आपण हे ॲट्रिब्युट्स एकूण सेकंद दर्शवणाऱ्या एका इंटिजरने बदलू शकतो. असे केल्याने `is_after` सारख्या मेथड्सचे इंप्लेमेंटेशन लिहायला सोपे होईल पण अन्य मेथड्स अवघड होतील

नवीन क्लास बनवून झाल्यावर तुम्हाला अशी जाणीव होऊ शकते हा क्लास अजून चांगल्याप्रकारे इंप्लेमेंट करता येईल. पण जर प्रोग्रामचे इतर भाग जर तुमचा क्लास वापरत असतील तर त्याचा इंटरफेस बदलणे वेळखाऊ आणि एरर्स देणारे ठरू शकते

पण जर तुम्ही आधी इंटरफेस व्यवस्थितपणे डिझाइन केला असेल तर तुम्ही तो *न* बदलता इंप्लेमेंटेशन बदलू शकता; म्हणजेच तुम्हाला प्रोग्रामचे अन्य भाग बदलायची गरज नाही.

१७.१२ शब्दार्थ

ऑब्जेक्ट-ओरिएंटेड लँग्वेज (object-oriented language): अशी प्रोग्रामिंग लँग्वेज ज्यात प्रोग्रामर-परिभाषित टाइप्स आणि मेथड्स अशी ऑब्जेक्ट-ओरिएंटेड प्रोग्रामिंगला चालना देणारी वैशिष्ट्ये असतात.

ऑब्जेक्ट-ओरिएंटेड प्रोग्रामिंग (object-oriented programming): प्रोग्रामिंगची अशी शैली ज्यात डेटा आणि त्यावर चालणारी ऑपरेशन्स ह्यांची मांडणी क्लासेस आणि मेथड्समध्ये केलेली असते.

मेथड (method): एका क्लासच्या डेफनिशनमध्ये डिफाइन केलेले फंक्शन जे त्या क्लासच्या इन्स्टन्सवर (instance) इन्व्होक केले जाते.

सब्जेक्ट (subject): ज्या ऑब्जेक्टवर मेथड इन्व्होक केली आहे तो.

पोझिशनल अर्ग्युमेंट (positional argument): असे अर्ग्युमेंट ज्यात संलग्न परॅमीटरचे नाव टाकलेले नसते, म्हणजे ते एक कीवर्ड अर्ग्युमेंट (keyword argument) नसते.

ऑपरेटर ओव्हरलोडिंग (operator overloading): प्रोग्रामर-परिभाषित टाइपवर चालावा म्हणून (+ सारख्या) ऑपरेटरची वर्तणूक बदलणे.

टाइप-वर आधारित हस्तांतरण (type-based dispatch): एक प्रोग्रामिंग पॅटर्न ज्यात अर्ग्युमेंटचा टाइप बघितला जातो आणि त्यानुसार कोणते फंक्शन कॉल करायचे हे ठरवले जाते.

पॉलिमॉर्फिक (polymorphic): एकाधिक टाइप्सवर चालणाऱ्या फंक्शनशी संबंधित.

१७.१३ प्रश्नसंच (Exercises)

प्रश्न १७.१. *ह्या प्रकरणातील कोड पुढील लिंकवरून डाऊनलोड करा: http://thinkpython2.com/code/Time2.py*

`Time` *चे ॲट्रिब्युट्स बदलून त्यांऐवजी एकच इंटिजर वापरा जो (मध्यरात्रीपासूनचे) एकूण सेकंद दर्शवेल. नंतर मेथड्स आणि* `int_to_time` *फंक्शन बदलून ह्या नवीन इंप्लेमेंटेशनवर चालतील असे बनवा. तुम्हाला* `main` *मधील टेस्ट कोड बदलायची गरज नाही पडली पाहिजे आणि प्रोग्रामचे आउटपुट आधीसारखेच असले पाहिजे. उत्तर: http://thinkpython2.com/code/Time2_soln.py.*

प्रश्न १७.२. *हा प्रश्न पायथॉनमधील सर्वांत कॉमन आणि शोधायला अवघड अशा एररविषयीची एक cautionary tale आहे, म्हणजे अशी गोष्ट जी धोक्याची सूचना देते.* `Kangaroo` *नावाच्या क्लासची डेफनिशन खालील मेथड्ससहित लिहा:*

१. `pouch_contents` *नावाचा ॲट्रिब्युट एका रिकाम्या लिस्टने इनिशलाइझ करणारी* `__init__` *मेथड.*

२. `put_in_pouch` *नावाची मेथड जी एक ऑब्जेक्ट कोणत्याही टाइपचा ऑब्जेक्ट घेऊन* `pouch_contents` *मध्ये टाकते.*

३. *Kangaroo ऑब्जेक्टचे आणि पिशवी (pouch) मधील सामानाचे स्ट्रिंगरूप पाठवणारी* `__str__` *मेथड.*

दोन Kangaroo ऑब्जेक्ट्स बनवून त्यांना `kanga` *आणि* `roo` *नावाच्या व्हेरिएबल्सना असाइन करा; नंतर* `roo` *ला* `kanga` *च्या पिशवी (pouch) मध्ये टाका.*

पुढील कोड डाऊनलोड करा: http://thinkpython2.com/code/BadKangaroo.py.

ह्या कोडमध्ये वरील प्रश्नाचे उत्तर आहे पण त्यात एक मोठा आणि फार वाईट बग आहे. त्याला शोधून दुरूस्त करा.

जर तुम्ही अडकलात तर तुम्ही पुढील लिंकवरून उत्तर डाऊनलोड करू शकता ज्यात बग-चे वर्णन केले आहे आणि त्याला दुरूस्तही केले आहे: http://thinkpython2.com/code/GoodKangaroo.py.

प्रकरण १८

इनहेरिटन्स (Inheritance)

ऑब्जेक्ट-ओरिएंटेड प्रोग्रामिंगचे एक मुख्य वैशिष्ट्य म्हणजे **इनहेरिटन्स** (**inheritance**, अनुहरण). इनहेरिटन्स म्हणजे एका क्लासपासून दुसरा नवीन क्लास बनवण्याची क्षमता. ह्या प्रकरणात आपण पत्ते (cards) आणि पोकर (Poker) ह्या पत्त्यांच्या खेळातील हात ह्यांचे क्लासेस वापरून इनहेरिटन्स शिकणार आहोत.

जर तुम्ही पोकर खेळत नसाल तर तुम्ही पुढील लिंकवर त्याविषयी वाचू शकता: http://en.wikipedia.org/wiki/Poker.

पण तुम्हाला हे वाचायची किंवा खेळ शिकायची गरज नाही; शिकण्यासाठी आणि प्रश्न सोडवण्यासाठी लागणाऱ्या गोष्टी आपण लागतील तशा पाहूच.

ह्या प्रकरणातील कोड-ची उदाहरणे पुढील लिंकवर उपलब्ध आहेत: http://thinkpython2.com/code/Card.py.

१८.१ Card ऑब्जेक्ट्स (Card objects)

पत्त्यांच्या एका कॅटमध्ये बावन्न पत्ते असतात; प्रत्येक पत्ता चारपैकी एका गटा (suit) चा आणि तेरापैकी एका रँक (rank) चा असतो. (ब्रिजमधील उतरत्या क्रमाने) गट हे पुढीलप्रमाणे: इस्पिक (Spades ♠), बदाम (Hearts ♥), चौकट (Diamonds ♦), किल्वर (Clubs ♣). रँक्स हे पुढीलप्रमाणे: एक्का (ace), दुर्री (2), तिर्री (3), चौकी (4), पंजी (5), छक्की (6), सत्ती (7), अठ्ठी (8), नव्वी (9) , दश्शी (10), गुलाम (Jack), राणी (Queen), आणि राजा (King). तुम्ही खेळत असलेल्या खेळानुसार एक्का हा राजाहून मोठा किंवा दुर्रीच्या खाली असू शकतो.

जर आपल्याला एका पत्त्यासाठी एक ऑब्जेक्ट बनवायचा असेल तर हे स्पष्ट आहे की ॲट्रिब्युट्स `rank` आणि `suit` असले पाहिजेत. पण त्यांचे टाइप काय असले पाहिजे हे स्पष्ट नाहीये. स्ट्रिंग वापरणे ही एक शक्यता आहे: गटांसाठी `'Spade'` अशाप्रकारचे शब्द आणि रँक्स-साठी `'Queen'` अशा प्रकारचे शब्द. ह्या इंप्लेमेंटेशनची एक कमतरता अशी आहे की पत्त्यांची तुलना करून कोणत्या पत्त्याचा मोठा रँक किंवा गट आहे हे शोधणे शक्य नाही.

दुसरा पर्याय म्हणजे रँक्स आणि गट हे इंटिजर्स वापरून **एन्कोड** (**encode** [१]) ह्यासंदर्भात, 'encode' म्हणजे आपण गट ते इंटिजर्स आणि रँक्स ते इंटिजर्स अशा दोन मॅपिंग्स (mappings) देणार आहोत. गुप्तता हा हेतू नाहीये (तसे असते तर आपण ह्याला 'encryption' म्हटले असते).

उदा., खालील तक्ता गट आणि संलग्न इंटिजर संकेत (code) दाखवतो:

इस्पिक (Spades)	↦	3
बदाम (Hearts)	↦	2
चौकट (Diamonds)	↦	1
किल्वर (Clubs)	↦	0

[१] म्हणजे सांकेतिक चिन्हात रुपांतर करणे.

हे संकेत वापरून पत्त्यांची तुलना करणे सोपे होते; मोठे गट मोठ्या इंटिजर्सना मॅप होत असल्यामुळे आपण गटांची तुलना त्यांच्या संकेतांच्या तुलनेद्वारे करू शकतो.

रँक्स-ची मॅपिंग ही स्पष्टच आहे; एक्का ते दश्शी अनुक्रमे 1 ते 10 क्रमांकांना मॅप करणार आहोत आणि चित्रांच्या पत्त्यांसाठी मॅपिंग खालीलप्रमाणे:

गुलाम (Jack) $\mapsto$ 11
राणी (Queen) $\mapsto$ 12
राजा (King) $\mapsto$ 13

ह्याठिकाणी $\mapsto$ चिन्ह वापरण्याचे कारण म्हणजे हे दर्शवणे की ह्या मॅपिंग्स पायथॉन प्रोग्रामचा भाग नाहीत. त्या प्रोग्राम डिझाइनचा भाग आहेत, पण त्या कोड-मध्ये दिसणार नाहीत.

`Card` ची क्लास डेफनिशन ही अशी:

```
class Card:
    """Represents a standard playing card."""

    def __init__(self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank
```

नेहमीप्रमाणे init मेथड प्रत्येक ॲट्रिब्युटसाठी एक ऑप्शनल परॅमीटर घेते. किल्वरची दुर्री हा डीफॉल्ट पत्ता आहे.

एक Card बनवण्यासाठी तुम्हाला `Card` कॉल करून त्याला पाहिजे असलेला गट आणि रँक पाठवावा लागेल:

```
queen_of_diamonds = Card(1, 12)
```

१८.२ क्लास ॲट्रिब्युट्स (Class attributes)

Card ऑब्जेक्ट वाचता येण्यासारख्या स्वरुपात प्रिंट करण्यासाठी आपल्याला इंटिजर संकेतांपासून संलग्न रँक्स आणि गटांपर्यंत मॅपिंग लागेल.हे करण्याचा एक सरळ मार्ग म्हणजे स्ट्रिंग्सची लिस्ट. ह्या लिस्ट्स आपण **क्लास ॲट्रिब्युट्स** (**class attributes**) ना असाइन करणार आहोत:

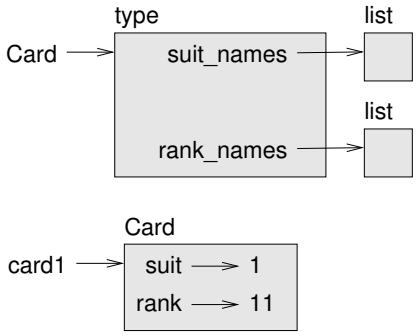
```
# inside class Card:

    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
              '8', '9', '10', 'Jack', 'Queen', 'King']

    def __str__(self):
        return '%s of %s' % (Card.rank_names[self.rank],
                             Card.suit_names[self.suit])
```

वरील `suit_names` आणि `rank_names` सारखी व्हेरिएबल्स जी क्लासच्यामध्ये पण कोणत्याही मेथडच्या बाहेर असतात त्यांना क्लास ॲट्रिब्युट म्हणतात कारण ती `Card` ह्या क्लास ऑब्जेक्टशी संबंधित असतात.

हे व्याख्या ती व्हेरिएबल्स `suit` आणि `rank` सारख्या व्हेरिएबल्सपासून वेगळी आहेत हे दर्शवते ज्यांना **इन्स्टन्स ॲट्रिब्युट** (**instance attributes**) म्हणतात कारण ती विशिष्ठ इन्स्टन्सशी संबंधित असतात.

दोन्हीही प्रकारचे ॲट्रिब्युट्स डॉट नोटेशन (dot notation) ने वापरता येतात. उदा., `__str__` मध्ये `self` हे Card ऑब्जेक्ट दर्शवते आणि `self.rank` त्याचा रँक. त्याचप्रमाणे `Card` हे एक क्लास ऑब्जेक्ट दर्शवते आणि `Card.rank_names` हे स्ट्रिंग्सची लिस्ट दर्शवते व ते त्या क्लासशी संबंधित आहे.

प्रत्येक पत्त्याचा स्वतंत्र `suit` आणि `rank` आहे, पण `suit_names` आणि `rank_names` ह्यांची एकच प्रत (copy) आहे.



आकृती १८.१: ऑब्जेक्ट डायग्राम (Object diagram).

सारांश म्हणजे `Card.rank_names[self.rank]` चा अर्थ असा की 'self ऑब्जेक्टचा `rank` ॲट्रिब्युट हा `Card` क्लासच्या `rank_names` लिस्टमध्ये इंडेक्स म्हणून वापरून योग्य ती स्ट्रिंग निवडा.'

`rank_names` चा पहिला एलेमेंट `None` आहे कारण शून्य रँकचा कोणताच पत्ता नाही. `None` सारखी फक्त जागा व्यापणारी (निरुपयोगी) व्हॅल्यू वापरल्यामुळे त्या लिस्टचा एक चांगला गुणधर्म असा मिळतो की 2 इंडेक्सला `'2'` ही स्ट्रिंग असते, 3 इंडेक्सला `'3'`, इ. लिस्टऐवजी डिक्शनरी वापरली असती तर ही युक्ती वापरायची गरज नसती पडली.

आतापर्यंत दिलेल्या मेथड्स वापरून आपण पत्ते तयार करून प्रिंट करू शकतो:

```
>>> card1 = Card(2, 11)
>>> print(card1)
Jack of Hearts
```

आकृती १८.१ ही `Card` क्लास ऑब्जेक्टची एका Card इन्स्टन्सची ऑब्जेक्ट डायग्राम आहे. `Card` एक क्लास ऑब्जेक्ट दर्शवते आणि त्याचा टाइप `type` आहे. आणि `card1` हे `Card` चा इन्स्टन्स दर्शवते, म्हणजेच त्याचा टाइप `Card` आहे. जागा वाचवण्यासाठी `suit_names` आणि `rank_names` हे दाखवले नाहीयेत.

१८.३ पत्त्यांची तुलना (Comparing cards)

बिल्ट-इन टाइप्सवर चालणारे रिलेश्नल ऑपरेटर्स (<, >, ==, इ.) व्हॅल्यूंची तुलना करून कोणती लहान, मोठी किंवा दुसऱ्यासारखीच आहे हे ठरवतात. प्रोग्रामर-परिभाषित टाइपवर आपण बिल्ट-इन ऑपरेटर्सना ओव्हर्राइड (override) करण्यासाठी `__lt__` ही मेथड देऊ शकतो (इकडे lt म्हणज 'less than' चे संक्षिप्त रूप).

`__lt__` ही `self` आणि `other` ही दोन परॅमीटर्स घेते आणि `self` जर `other` पेक्षा लहान असेल आणि सारखे नसेल तर `True` रिटर्न करते.

पत्त्यांचा बरोबर क्रम काय हे स्पष्ट नाहीये. उदा., किल्वरची तिर्री मोठी का चौकटची दुर्री? एकाचा रँक मोठा आहे पण दुसऱ्याचा गट. म्हणून पत्त्यांची तुलना करण्यासाठी तुम्हाला हे ठरवावे लागेल की रँक जास्ती महत्त्वाचा की गट.

ह्याचे उत्तर तुम्ही कोणता खेळ खेळताहात ह्यावर अवलंबून आहे, पण गोष्टी जास्ती अवघड न करता आपण एक अहेतुक निवड अशी करू की गट जास्ती महत्त्वाचा, म्हणजे इस्पिकचे सर्व पत्ते चौकटच्या कोणत्याही पत्त्याहून मोठे, इ.

आता हे ठरवलेच आहे तर आपण `__lt__` लिहून शकतो:

```
# inside class Card:

    def __lt__(self, other):
        # check the suits
        if self.suit < other.suit: return True
```

```
        if self.suit > other.suit: return False

        # suits are the same... check ranks
        return self.rank < other.rank
```

टपलची तुलना करून तुम्ही हे जास्ती संक्षेपाने लिहू शकता:

```
# inside class Card:

    def __lt__(self, other):
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return t1 < t2
```

सराव म्हणून, Time ऑब्जेक्ट्ससाठी `__lt__` मेथड लिहा. तुम्ही टपलची तुलना वापरू शकता पण इंटिजर्सची तुलनाही विचारात घेऊ शकता.

१८.४ पत्त्यांचे कॅट (Decks)

आपल्याकडे आता Cards आहेत. पुढची पायरी म्हणजे Deck (पत्त्यांचा कॅट, म्हणजेच एक संच) डिफाइन करणे. एक कॅट पत्त्यांचा बनतो, म्हणजेच प्रत्येक Deck मध्ये पत्त्यांची लिस्ट ॲट्रिब्युट म्हणून ठेवणे सरळ आहे.

खाली `Deck` ची क्लास डेफनिशन आहे; init मेथड `cards` ॲट्रिब्युट बनवते आणि बावन्न पत्त्यांचा साधा संच बनवते:

```
class Deck:

    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
                card = Card(suit, rank)
                self.cards.append(card)
```

कॅट बनवण्याचा सर्वांत सोपा मार्ग म्हणजे नेस्टेड लूप (nested loop) वापरून. बाहेरचा लूप 0 ते 3 गट इटरेट करतो आणि आतील लूप 1 ते 13 रँक्स. (आतील) प्रत्येक इटरेशन त्यावेळच्या `suit` आणि `rank` च्या व्हॅल्यूझ वापरून एक नवीन Card बनवते आणि `self.cards` च्या शेवटी जोडते (append करते).

१८.५ Deck प्रिंट करणे (Printing the deck)

`Deck` ची `__str__` मेथड ही अशी:

```
# inside class Deck:

    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)
```

ही मेथड, एक मोठी स्ट्रिंग जमा करण्याची कार्यक्षम पद्धत दाखवते: स्ट्रिंग्सची लिस्ट बनवून नंतर `join` ही स्ट्रिंग मेथड वापरणे. इथे `str` हे बिल्ट-इन फंक्शन प्रत्येक पत्त्यावर `__str__` मेथड इन्व्होक करते आणि त्याचे स्ट्रिंगरूप पाठवते.

आपण `join` हे एका न्यूलाइन कॅरेक्टर (newline character) वर इन्व्होक करत असल्यामुळे पत्ते वेगवेगळ्या ओळींवर दिसतात:

```
>>> deck = Deck()
>>> print(deck)
Ace of Clubs
2 of Clubs
3 of Clubs
...
10 of Spades
Jack of Spades
Queen of Spades
King of Spades
```

जरी उत्तर ५२ ओळींवर दिसत असले तरी ते एक न्यूलाइन्स असणारी स्ट्रिंग आहे.

१८.६ पिसणे, पत्ता काढणे, पत्ता लावणे, सॉर्ट करणे (Add, remove, shuffle and sort)

पत्ते वाटण्यासाठी आपल्याला कॅटमधून एक पत्ता काढून तो रिटर्न करणारी मेथड लागेल. लिस्टची `pop` मेथड हे करायला सोयीस्कर आहे:

```
# inside class Deck:

    def pop_card(self):
        return self.cards.pop()
```

`pop` लिस्टमधील *शेवटचा* पत्ता काढून रिटर्न करत असल्यामुळे आपण पत्ते खालून वाटतो.

पत्ता लावण्यासाठी, आपण `append` ही लिस्ट मेथड वापरू शकतो:

```
# inside class Deck:

    def add_card(self, card):
        self.cards.append(card)
```

अशी मेथड जी स्वत: काही काम *न* करता दुसरी मेथड वापरते तिला **आवरण** (**veneer**, व्हेनीर) म्हणतात. हा इंग्रजी शब्द सुतारकामातून आलेला आहे: veneer म्हणजे कमी दर्जाच्या लाकूड चांगले दिसावे म्हणून त्याला चिकटवलेला चांगल्या दर्जाच्या लाकडाचा पातळ पापुद्रा.

ह्याठिकाणी `add_card` ही एक 'पातळ' मेथड आहे जी एक लिस्ट ऑपरेशन पत्त्याच्या कॅटच्या भाषेत व्यक्त करते. ती इंप्लेमेंटेशन चा इंटरफेसचे (interface) चांगले रूप दाखवण्याचे काम करते.

अजून एक उदाहरण म्हणून आपण `random` मोड्युलमधील `shuffle` फंक्शन वापरून Deck मध्ये `shuffle` नावाची मेथड लिहू शकतो (पत्ते पिसण्यासाठी):

```
# inside class Deck:

    def shuffle(self):
        random.shuffle(self.cards)
```

`random` ला इंपोर्ट करायला विसरू नका.

सरावासाठी `sort` ही लिस्ट मेथड वापरून `sort` नावाची एक Deck मेथड लिहा जी एका कॅटमधील पत्ते क्रमाने लावते. `sort` ही मेथड क्रम ठरवण्यासाठी आपण लिहिलेली `__lt__` मेथड वापरते.

१८.७ इनहेरिटन्स (Inheritance)

इनहेरिटन्स म्हणजे एका क्लासपासून दुसरा नवीन क्लास बनवण्याची क्षमता. उदाहरण म्हणून समजा आपल्याला 'हात' ('hand') दर्शवण्यासाठी एक क्लास बनवायचा आहे, म्हणजे एका खेळाडूच्या हातात असलेले पत्ते[३]. हात थोडा कॅटसारखाच असतो: दोन्हींमध्ये काही पत्ते असतात, आणि दोन्हींमध्ये पत्ते काढणे आणि लावणे ह्या क्रिया असतात.

पण हात हा कॅटपासून वेगळाही आहे; हातांच्या काही क्रिया कॅटला लागू नाही होत. उदा., पोकरमध्ये आपल्याला दोन हातांची तुलना करून कोणता जिंकतो हे शोधावे लागू शकते. ब्रिजमध्ये प्रत्येक हाताचे गुण शोधावे लागू शकतात.

क्लासेसमधील हा संबंध—थोडा सारखा, पण वेगळा— इनहेरिटन्सने सोयीस्करपणे व्यक्त करता येतो. आधीच डिफाइन केलेल्या क्लासकडून इनहेरिट करणारा (वारशाने मिळवणारा) क्लास डिफाइन करण्यासाठी आधीच्या क्लासचे नाव कंसात लिहितात:

```
class Hand(Deck):
    """Represents a hand of playing cards."""
```

ही डेफनिशन दर्शवते की `Hand` क्लास `Deck` कडून इनहेरिट करतो; म्हणजेच आपण `pop_card` आणि `add_card` सारख्या मेथड्स Hand आणि Deck दोन्हींच्या इन्स्टन्सवर वापरू शकतो.

जेव्हा एक नवीन क्लास आधी डिफाइन केलेल्या क्लासकडून इनहेरिट करतो तेव्हा आधीच्या क्लास ला **पालक** (**parent**, पेरेंट) आणि नवीन क्लासला **बालक** (**child**, चाइल्ड) म्हणतात.

ह्या उदाहरणात `Hand` क्लास `__init__` मेथड `Deck` मधून इनहेरिट करतो, पण ती मेथड आपल्याला जे पाहिजे ते नाही करत: हाताला ५२ नवीन पत्ते देण्याऐवजी Hand च्या init मेथडने `cards` ला रिकाम्या लिस्टने इनिशलाइझ केले पाहिजे.

जर आपण `Hand` क्लासमध्ये init मेथड लिहिली तर ती `Deck` मधलीला ओव्हर्राइड (override) करते:

```
# inside class Hand:

    def __init__(self, label=''):
        self.cards = []
        self.label = label
```

जेव्हा तुम्ही नवीन Hand इन्स्टन्स बनवता तेव्हा पायथॉन ही init मेथड इन्व्होक करतो, `Deck` मधली नाही.

```
>>> hand = Hand('new hand')
>>> hand.cards
[]
>>> hand.label
'new hand'
```

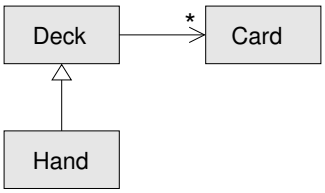
इतर मेथड्स `Deck` कडून इनहेरिट होतात, म्हणजे आपण पत्ते वाटताना `pop_card` आणि `add_card` वापरू शकतो:

```
>>> deck = Deck()
>>> card = deck.pop_card()
>>> hand.add_card(card)
>>> print(hand)
King of Spades
```

पुढची पायरी म्हणजे ह्या कोड-ला `move_cards` नावाच्या मेथडमध्ये एन्कॅप्सुलेट (encapsulate) करणे:

```
# inside class Deck:

    def move_cards(self, hand, num):
        for i in range(num):
            hand.add_card(self.pop_card())
```

[३] पत्त्यांमधल्या 'हाता'चा वेगळा अर्थही होतो, पण ह्याठिकाणी दिलेला अर्थ धरून चालूया.



आकृती १८.२: क्लास डायग्राम (Class diagram).

`move_cards` दोन अर्ग्युमेंट्स घेते, एक Hand ऑब्जेक्ट आणि किती पत्ते वाटायचे आहेत ते. ती `self` आणि `hand` ह्या दोन्हींमध्ये बदल करते.

काही खेळांमध्ये पत्ते एका हातातून दुसऱ्यात हलवले जातात, किंवा हातातून परत कॅटमध्ये. तुम्ही ह्यांपैकी कोणत्याही क्रियेसाठी `move_cards` वापरू शकता: `self` हे एक Deck किंवा Hand दोन्हींपैकी कोणतेही ऑब्जेक्ट दर्शवू शकते, आणि `hand` हे Deck ऑब्जेक्टही दर्शवू शकते.

इनहेरिटन्स हे उपयोगी वैशिष्ट्य आहे. काही प्रोग्राम्समध्ये इनहेरिटन्स वापरून कोड-ची पुनरावृत्ती टाळता येते आणि ते चांगल्याप्रकारे लिहिता येतात. इनहेरिटन्समुळे कोड-पुनर्वापराला चालना मिळते, कारण तुम्ही पालक क्लासमध्ये त्यांना *न* बदलता वैशिष्ट्यांची भर घालू शकता. काही ठिकाणी इनहेरिटन्स वापरून तयार केलेली संरचना प्रॉब्लेमची नैसर्गिक संरचना व्यक्त करते, ज्यामुळे डिझाइन समजायला सोपे जाते.

पण दुसऱ्या बाजूस, इनहेरिटन्समुळे प्रोग्राम समजायला अवघड होऊ शकतो. जेव्हा एखादी मेथड इन्व्होक केली जाते तेव्हा कधीकधी हे स्पष्ट नसते की तिची डेफनिशन कुठे शोधायची. संबंधित कोड अनेक मोड्युल्समध्ये पसरलेला असू शकतो. आणि, अशा अनेक गोष्टी ज्या इनहेरिटन्स वापरून करता येऊ शकतात त्या इनहेरिटन्स *न* वापरता तितक्याच किंवा अजून चांगल्या पद्धतीने करता येऊ शकतात.

१८.८ क्लास डायग्राम (Class diagrams)

आतापर्यंत आपण स्टॅक डायग्राम (जी प्रोग्रामची सद्यःस्थिती दाखवते) आणि ऑब्जेक्ट डायग्राम (जी ऑब्जेक्टचे ॲट्रिब्युट्स आणि त्यांच्या व्हॅल्यूझ दाखवते) पाहिल्या. ह्या डायग्राम्स प्रोग्रामच्या एक्सेक्युशनचे एक क्षणचित्र दाखवतात, म्हणजेच जसजसा प्रोग्राम रन होतो, तसतशा त्या बदलतात.

त्या अतिशय सविस्तर असतात; काही ठिकाणी, अतिजास्त सविस्तर. क्लास डायग्राम ही प्रोग्रामच्या संरचनेचे संकल्पनात्मक चित्रण करते. स्वतंत्र ऑब्जेक्ट्स दाखवण्याऐवजी ती क्लासेस आणि त्यांमधले संबंध दाखवते.

क्लासेसमध्ये अनेकप्रकारचे संबंध असतात:

- एका क्लासच्या ऑब्जेक्टमध्ये दुसऱ्या क्लासच्या ऑब्जेक्टला रेफ्रन्स असू शकतो. उदा., प्रत्येक Rectangle मध्ये एका Point ला रेफ्रन्स असतो, आणि प्रत्येक Deck मध्ये अनेक Cards ना रेफ्रन्सेस असतात. ह्याप्रकारच्या संबंधाला **HAS-A** म्हणतात, म्हणजे इंग्रजीमध्ये 'a Rectangle has a Point.'
- एक क्लास दुसऱ्या क्लासकडून इनहेरिट करू शकतो. ह्या संबंधाला **IS-A** म्हणतात, म्हणजे इंग्रजीमध्ये 'a Hand is a kind of a Deck.'
- एक क्लास दुसऱ्या क्लासवर अवलंबून असू शकतो: म्हणजे एका क्लासची मेथड दुसऱ्या क्लासचा ऑब्जेक्ट अर्ग्युमेंट म्हणून घेत असेल किंवा दुसऱ्या क्लासचा ऑब्जेक्ट वापरून एखादे कॉंप्युटेशन करत असेल, असे असू शकते. ह्या संबंधाला **डिपेंडन्सी** (**dependency**) म्हणतात.

क्लास डायग्राम (**class diagram**) हे संबंध व्यक्त करते. उदा., आकृती १८.२ `Card`, `Deck`, आणि `Hand` मधील संबंध दाखवते

रिकाम्या-त्रिकोणाचे टोक असलेला बाण IS-A संबंध दर्शवतो; ह्याठिकाणी तो Hand क्लास Deck कडून इनहेरिट करतो हे दाखवतो.

साध्या टोकाचा बाण HAS-A संबंध दर्शवतो; ह्याठिकाणी Deck मध्ये Card ऑब्जेक्ट्सला रेफ्रन्सेस आहेत.

बाणाच्या टोकाजवळ असलेले * चिन्ह **अनेकत्व** (**multiplicity**) दर्शवते; ते हे दाखवते की एका Deck मध्ये किती Card ऑब्जेक्ट्स आहेत. अनेकत्व ही साधी संख्या (उदा., 52), एक कक्षा (उदा., `5..7`), किंवा * असू शकते; * दर्शवते की Deck मध्ये कितीही Card ऑब्जेक्ट्स असू शकतात.

ह्या डायग्राममध्ये कोणत्याही डिपेंडन्सीझ (dependencies) नाहीत. साधारणपणे त्या तुटक बाणाने दाखवल्या जातात. किंवा जर खूप डिपेंडन्सीझ असतील तर त्या कधीकधी गाळल्या जातात.

अजून सविस्तर डायग्राम हे दाखवू शकते की एका Deck मध्ये खरे तर Card ची एक *list* आहे, पण लिस्ट आणि डिक्शनरी सारखे बिल्ट-इन टाइप्स क्लास डायग्राममध्ये सहसा दाखवत नाहीत.

१८.९ डीबगिंग (Debugging)

इनहेरिटन्समुळे डीबगिंग अवघड होऊ शकते, कारण जेव्हा तुम्ही एका ऑब्जेक्टवर एखादी मेथड इन्व्होक करता तेव्हा कोणती मेथड इन्व्होक होईल हे शोधणे अवघड जाऊ शकते.

समजा तुम्ही Hand ऑब्जेक्ट वापरणारे एक फंक्शन लिहित आहात. ते फंक्शन सर्व प्रकारचे Hand (उदा., PokerHand, BridgeHand, इ.) हाताळणारे असावे हा तुमचा उद्देश आहे. जर तुम्ही `shuffle` सारखी मेथड इन्व्होक केली तर `Deck` मध्ये दिलेली रन होऊ शकते, पण जर त्याच्या कोणत्याही बालक-क्लास (child-class) ने जर ही मेथड ओव्हर्राइड (override) केली असेल तर ती पण रन होऊ शकते. सहसा असे होणे रास्त आणि अपेक्षित असते पण त्यामुळे गोंधळ होऊ शकतो.

जेव्हाही तुम्हाला फ्लो-ऑफ-एक्सेक्युशन (flow-of-execution) बद्दल अशी शंका येईल, तेव्हा सर्वांत सोपा उपाय म्हणजे संबंधित मेथड्सच्या सुरुवातीला प्रिंट स्टेटमेंट्स लावणे. जर `Deck.shuffle` मेथड `Running Deck.shuffle` असा मेसेज प्रिंट करत असेल तर जसजसा प्रोग्राम रन होतो, तसतसा आपण फ्लो-ऑफ-एक्सेक्युशनचा माग घेऊ शकतो.

अजून एक पर्याय म्हणून तुम्ही खालील फंक्शन वापरू शकता. हे फंक्शन एक ऑब्जेक्ट आणि स्ट्रिंगस्वरुपात मेथडचे नाव घेते आणि त्या मेथडची डेफनिशन देणारा क्लास रिटर्न करते:

```
def find_defining_class(obj, meth_name):
    for ty in type(obj).mro():
        if meth_name in ty.__dict__:
            return ty
```

उदा.:

```
>>> hand = Hand()
>>> find_defining_class(hand, 'shuffle')
<class '__main__.Deck'>
```

ह्याठिकाणी ह्या Hand साठी `shuffle` मेथड `Deck` मधील आहे.

`find_defining_class` हे फंक्शन ज्या क्लास ऑब्जेक्ट्स (टाइप्स) मध्ये मेथड शोधली जाईल त्यांची लिस्ट मिळवण्यासाठी `mro` मेथड वापरते. 'MRO' हे 'method resolution order' चे संक्षिप्त रूप आहे; पायथॉन मेथडचे नाव शोधण्यासाठी ('resolve' करण्यासाठी) वापरतो तो क्लासेसचा क्रम ही order दर्शवते.

एक डिझाइन सूचना अशी: जेव्हा तुम्ही एक मेथड ओव्हर्राइड (override) करता, तेव्हा नवीन मेथडचा इंटरफेस हा जुन्या मेथडसारखाच असला पाहिजे. तिने सारखेच पॅरॅमीटर्स घेतले पाहिजेत, सारखाच टाइप रिटर्न केला पाहिजे, आणि सारख्याच प्रीकंडिशन्स (precoditions) आणि पोस्टकंडिशन्स (postconditions) ची पूर्तता केली पाहिजे.

जर तुम्ही ह्या सूचनेचे पालन केले, तर पालक-क्लासच्या इन्स्टन्स (उदा., Deck) वर चालणारे कोणतेही फंक्शन बालक-क्लासच्या इन्स्टन्स (उदा., Hand) वरही चालेल.

ह्या नियमाला 'Liskov substitution principle' म्हणतात, जो तुम्ही मोडलात तर (माफ करा), पण तुमचा कोड पत्त्यांच्या बंगल्याप्रमाणे कोसळेल.

१८.१० डेटा एन्कॅप्सुलेशन (Data encapsulation)

मागच्या काही प्रकरणांमध्ये आपण जो डेव्हेलपमेंट प्लान शिकलो आहोत त्याला 'ऑब्जेक्ट-ओरिएंटेड डिझाइन' ('object-oriented design') म्हणता येईल. आपण आपल्याला कोणते ऑब्जेक्ट्स पाहिजे आहेत ते ठरवले—जसे `Point`, `Rectangle`, आणि `Time`—आणि ते बनवता येण्यासाठी क्लासेस बनवले. प्रत्येक ठिकाणी, ऑब्जेक्टचा संबंध (आपल्या किंवा गणितीय जगातील) एका प्रत्यक्ष घटकाशी स्पष्टपणे लागत होता.

पण कधीकधी हा संबंध स्पष्ट नसतो; हे सरळपणे सांगता येत नाही की तुम्हाला कोणते ऑब्जेक्ट्स लागणार आहेत आणि त्यांतील संबंध कसे प्रस्थापित होणार आहेत. त्यावेळी तुम्हाला वेगळा डेव्हेलपमेंट प्लान लागतो. ज्याप्रकारे आपली फंक्शन इंटरफेसशी ओळख एन्कॅप्सुलेशन आणि जनरलायझेशन (encapsulation and generalization) करताना झाली, **डेटा एन्कॅप्सुलेशन** (**data encapsulation**, डेटा विभागीकरण) द्वारे आपली ओळख क्लास इंटरफेसेसशी होऊ शकते.

विभाग १३.८ मधील मार्कोव्ह विश्लेषण (Markov analysis) ह्याचे उत्तम उदाहरण देते. पुढील लिंकवरील कोड दोन ग्लोबल (global) व्हेरिएबल्स वापरतो, ती म्हणजे `suffix_map` आणि `prefix`, जी अनेक फंक्शन्समध्ये वाचली आणि लिहिली जातात: http://thinkpython2.com/code/markov.py.

```
suffix_map = {}
prefix = ()
```

ही व्हेरिएबल्स ग्लोबल असल्यामुळे आपण एकावेळी फक्त एकच विश्लेषण चालवू शकतो. जर आपण दोन वेगळे मजकूर वाचले तर त्यांची prefixes (शब्दमालिका) आणि suffixes (नंतर येऊ शकणाऱ्या शब्दांचा समूह) सारख्याच डेटा स्ट्रक्चर्समध्ये टाकली जातील (ज्यामुळे नाविन्यपूर्ण नवीन मजकूर निर्माण होतो).

पण अनेक विश्लेषण करण्यासाठी आणि त्यांना स्वतंत्र ठेवण्यासाठी आपण प्रत्येक विश्लेषणाची स्थिती (state) एका ऑब्जेक्टमध्ये एन्कॅप्सुलेट करू शकतो. ते असे दिसेल:

```
class Markov:

    def __init__(self):
        self.suffix_map = {}
        self.prefix = ()
```

मग, आपण फंक्शन्सपासून मेथड्स बनवूया. उदा., खाली `process_word` दिली आहे:

```
    def process_word(self, word, order=2):
        if len(self.prefix) < order:
            self.prefix += (word,)
            return

        try:
            self.suffix_map[self.prefix].append(word)
        except KeyError:
            # if there is no entry for this prefix, make one
            self.suffix_map[self.prefix] = [word]

        self.prefix = shift(self.prefix, word)
```

प्रोग्रामचे असे रुपांतर—त्याचे वर्तन *न* बदलता डिझाइन बदलणे—हे रिफॅक्टरिंग (refactoring, विभाग ४.७ बघा) चे अजून एक उदाहरण होय.

ह्या उदाहरणावरून आपल्याला ऑब्जेक्ट्स आणि मेथड्स डिझाइन करण्याच्या एका डेव्हेलपमेंट प्लानची कल्पना येते:

१. सुरुवातीला ती फंक्शन्स लिहा जी (गरजेनुसार) ग्लोबल व्हेरिएबल्स वाचतील आणि लिहितील.

२. एकदाचा प्रोग्राम ठीक चालू लागला की ग्लोबल व्हेरिएबल्स आणि ती वापरणारी फंक्शन्स ह्यांच्यांतल्या संबंधांचे निरीक्षण करा.

३. (एकमेकांशी) संबंधित असलेल्या व्हेरिएबल्सना एका ऑब्जेक्टचे ॲट्रिब्युट्स म्हणून एन्कॅप्सुलेट करा.

४. संबंधित फंक्शन्सचे नवीन क्लासच्या मेथड्समध्ये रुपांतर करा.

सराव म्हणून पुढील लिंकवरून मार्कोव्ह कोड डाऊनलोड करा आणि वरील सूचना अंमलात आणून ग्लोबल व्हेरिएबल्सना `Markov` नावाच्या क्लासचे ॲट्रिब्युट्स म्हणून एन्कॅप्सुलेट करा: http://thinkpython2.com/code/markov.py.

उत्तर: http://thinkpython2.com/code/markov2.py.

१८.११ शब्दार्थ

एन्कोड (encode, सांकेतिक चिन्हात रुपांतर करणे): एका संचातील व्हॅल्यूझ दुसऱ्या संचातील व्हॅल्यूझद्वारे दर्शवण्यासाठी त्यांतील मॅपिंग बनवणे.

क्लास ॲट्रिब्युट (class attribute): क्लास ऑब्जेक्टशी संबंधित ॲट्रिब्युट. क्लास ॲट्रिब्युट हा क्लास डेफनिशनच्या आतमध्ये पण कोणत्याही मेथडच्या बाहेर डिफाइन केलेला असतो.

इन्स्टन्स ॲट्रिब्युट (instance attribute): क्लासच्या इन्स्टन्सशी संबंधित ॲट्रिब्युट.

आवरण (veneer, व्हेनीर): एक फंक्शन किंवा मेथड ज्यात स्वतः जास्ती कॉंप्युटेशन *न* करता दुसऱ्या फंक्शनसाठी इंटरफेस पुरवला जातो.

इनहेरिटन्स (inheritance): आधी डिफाइन केलेल्या एका क्लासपासून त्यात बदल करून दुसरा नवीन क्लास बनवण्याची क्षमता.

पालक क्लास (parent class, पेरेंट क्लास): असा क्लास ज्याच्याकडून एक बालक क्लास इनहेरिट करतो.

बालक क्लास (child class, चाइल्ड क्लास): आधीपासून असलेल्या एका क्लासकडून इनहेरिट करून बनवलेला एक नवीन क्लास; 'सबक्लास' ('subclass') असेही म्हणतात.

IS-A संबंध (IS-A relationship): बालक-क्लास आणि पालक-क्लास ह्यांमधील संबंध.

HAS-A संबंध (HAS-A relationship): एका क्लासच्या इन्स्टन्समध्ये दुसऱ्या क्लासच्या इन्स्टन्सेसला रेफ्रन्सेस असल्यामुळे त्यांच्यांत बनलेला संबंध.

डिपेंडन्सी (dependency): एका क्लासच्या इन्स्टन्सने दुसऱ्या क्लासचे इन्स्टन्सेस त्यांना ॲट्रिब्युट्स म्हणून *न* ठेवता वापरण्यामुळे त्यांच्यांत बनलेला संबंध.

क्लास डायग्राम (class diagram): एका प्रोग्राममधील क्लासेस आणि त्यांच्यांतले संबंध दर्शवणारी आकृती.

अनेकत्व (multiplicity): HAS-A संबंध क्लास डायग्राममध्ये दाखवताना दुसऱ्या क्लासच्या इन्स्टन्सेसना किती रेफ्रन्सेस आहेत हे दर्शवण्याचे एक नोटेशन.

डेटा एन्कॅप्सुलेशन (data encapsulation): एक प्रोग्राम डेव्हेलपमेंट प्लान ज्यात ग्लोबल व्हेरिएबल्स वापरून नमुना (prototype) बनवला जातो आणि अंतिम आवृत्तीमध्ये त्यांचे इन्स्टन्स ॲट्रिब्युट्समध्ये रुपांतर केले जाते.

१८.१२ प्रश्नसंच (Exercises)

प्रश्न १८.१. *खालील प्रोग्राममधील क्लासेस आणि त्यांतले संबंध दाखवणारी क्लास डायग्राम काढा.*

```
class PingPongParent:
    pass

class Ping(PingPongParent):
    def __init__(self, pong):
        self.pong = pong


class Pong(PingPongParent):
    def __init__(self, pings=None):
        if pings is None:
            self.pings = []
        else:
            self.pings = pings

    def add_ping(self, ping):
        self.pings.append(ping)

pong = Pong()
ping = Ping(pong)
pong.add_ping(ping)
```

प्रश्न १८.२. *Deck मध्ये* `deal_hands` *नावाची मेथड लिहा जी हातांची संख्या आणि प्रत्येक हातातील पत्त्यांची संख्या हे दोन अर्ग्युमेंट्स घेते. आणि त्याप्रमाणे योग्य तितके Hand ऑब्जेक्ट्स बनवून प्रत्येकात योग्य तितके पत्ते वाटून त्या हातांची लिस्ट त्या मेथडने रिटर्न करायला हवी.*

प्रश्न १८.३. *पोकर (poker) मधील हात त्यांच्या मूल्यांच्या चढत्या क्रमाने आणि येण्याच्या संभाव्यतेच्या उतरत्या क्रमाने खाली दिले आहेत:*

जोडी (pair): *सारख्याच रँकचे दोन पत्ते (उदा., दोन चौक्या)*

दोन जोड्या (two pair): *सारख्याच रँकच्या दोन जोड्या (उदा., दोन पंज्या आणि दोन गुलाम)*

एकाप्रकारचे तीन (three of a kind): *सारख्याच रँकचे तीन पत्ते (उदा., तीन चौक्या)*

स्ट्रेट (straight): *रँकच्या क्रमाने असलेले पाच पत्ते. एक्का कमी किंवा जास्ती असू शकतो, म्हणजे* `Ace-2-3-4-5` *(एक्का, दुर्री, तिर्री, चौकी, पंजी) हा स्ट्रेट आहे आणि* `10-Jack-Queen-King-Ace` *(दश्शी, गुलाम, राणी, राजा, एक्का) हा पण स्ट्रेट आहे, पण* `Queen-King-Ace-2-3` *(राणी, राजा, एक्का, दुर्री, तिर्री) हा नाहीये.*

फ्लश (flush): *एकाच गटाचे (suit) पाच पत्ते.*

फुल हाऊस (full house): *एका रँकचे तीन पत्ते आणि दुसऱ्या रँकचे दोन.*

एकाप्रकारचे चार (four of a kind): *सारख्याच रँकचे चार पत्ते (उदा., चार चौक्या)*

स्ट्रेट फ्लश (straight flush): *एकाच गटाचे (suit) रँकच्या क्रमाने असलेले पाच पत्ते.*

ह्या प्रश्नाचा उद्देश हे विविध हात मिळण्याची संभाव्यता (probability) शोधणे हा आहे.

१. *पुढील लिंकवरून खालील फाइल्स डाऊनलोड करा: http://thinkpython2.com/code:*
   `Card.py` *: ह्या प्रकरणातील* `Card`, `Deck`, *आणि* `Hand` *ह्यांच्या पूर्ण आवृत्त्या.*

`PokerHand.py` : *पोकरचा हात दर्शवणाऱ्या एका क्लासचे अपूर्ण इंप्लेमेंटेशन आणि त्याला टेस्ट करणारा काही कोड.*

२. *तुम्ही जर* `PokerHand.py` *रन केला तर तो 7 पत्त्यांचे सात हात वाटतो आणि तपासतो की त्यांपैकी एकात फ्लश (flush) आहे का. पुढे जाण्याआधी हा कोड नीट वाचा.*

३. `PokerHand.py` *मध्ये* `has_pair`, `has_twopair`, *इ., नावाच्या मेथड्स बनवा ज्या जर हात योग्य त्या अटीची पूर्तता करत असेल तर* `True` *नाही तर* `False` *रिटर्न करतात (उदा.,* `has_pair` *जर हातात जोडी असेल तर* `True` *रिटर्न करेल). तुमचा कोड कितीही पत्त्यांच्या हातांसाठी चालला पाहिजे (जरी एका हातात ५ किंवा ७ पत्ते असणे सर्वांत कॉमन आहे).*

४. `classify` *(वर्गीकरण करणे) नावाची मेथड लिहा जी हाताचे सर्वांत जास्त मूल्य असलेले वर्गीकरण करते आणि* `label` *ॲट्रिब्युटची व्हॅल्यू त्याप्रमाणे ठेवते. उदा., जर एका 7 पत्त्यांच्या हातात एक फ्लश (flush) आणि एक जोडी असेल तर त्याचे लेबल 'flush' ठेवले पाहिजे.*

५. *तुमच्या वर्गीकरण करणाऱ्या मेथड्स नीट चालत आहेत ह्याची खात्री पटल्यावर पुढची पायरी ही विविध हातांची संभाव्यता शोधण्याची आहे.* `PokerHand.py` *मध्ये एक फंक्शन लिहा जे पत्त्यांचा एक कॅट पिसून (shuffle करून), त्याचे हातांत विभाजन करून त्या हातांचे वर्गीकरण करते, आणि हातांचा प्रत्येक प्रकार किती वेळा आला आहे ते मोजते.*

६. *हातांचे प्रकार आणि त्यांच्या संभाव्यतांचा तक्ता प्रिंट करा. जोपर्यंत तुमच्या संभाव्यता बरोबर किंमतीकडे एकवटत (converge होत) नाहीत तोपर्यंत हातांच्या अधिकाधिक संख्या वापरा. तुमच्या उत्तरांची पुढील लिंकवरील उत्तरांशी तुलना करा:* http://en.wikipedia.org/wiki/Hand_rankings.

उत्तर: http://thinkpython2.com/code/PokerHandSoln.py.

प्रकरण १९

साखरफुटाणे (The Goodies)

ह्या पुस्तकाचा एक उद्देश तुम्हाला शक्य तितके कमी पायथॉन शिकवणे हा होता. जिथे काही करण्याचे दोन मार्ग होते तिथे एक दाखवून दुसरा वगळला. कधीकधी दुसरा प्रश्नसंचात टाकला.

आता आपण काही *न* पाहिलेल्या चांगल्या गोष्टी बघू. पायथॉनमध्ये अशा अनेक गोष्टी आहेत ज्यांची खूप अशी गरज नाहीये—तुम्ही त्यांच्याशिवाय चांगला कोड लिहू शकता—पण कधीकधी त्या वापरून तुम्ही असा कोड लिहू शकता जो जास्ती संक्षिप्त, समजायला सोपा, किंवा कार्यक्षम, किंवा हे तिन्ही, असतो.

१९.१ कंडिश्नल एक्स्प्रेशन (Conditional expression)

विभाग ५.४ मध्ये आपण कंडिश्नल स्टेटमेंट्स पाहिली. दोनपैकी एक व्हॅल्यू निवडण्यासाठी ती वापरली जातात; उदा.:

```
if x > 0:
    y = math.log(x)
else:
    y = float('nan')
```

हे स्टेटमेंट `x` धन (positive) आहे का हे तपासते. जर असेल, तर ते `math.log` कॉंप्युट करते. नाहीतर `math.log` कॉल केल्यावर ते `ValueError` रेय्झ (raise) करेल. तसे होऊन प्रोग्राम बंद होणे टाळण्यासाठी आपण 'NaN' तयार करतो, जी एक विशेष फ्लोटिंग-पॉइंट व्हॅल्यू आहे जी 'Not a Number' व्यक्त करते.

आपण हे स्टेटमेंट **कंडिश्नल एक्स्प्रेशन** (**conditional expression**) वापरून जास्त संक्षेपाने लिहू शकतो:

```
y = math.log(x) if x > 0 else float('nan')
```

तुम्ही हे जवळजवळ इंग्रजीसारखेच वाचू शकता: 'y gets log-x if x is greater than 0; otherwise it gets NaN'

कधीकधी रिकर्सिव्ह फंक्शन कंडिश्नल एक्स्प्रेशन वापरून लिहिता येते. उदा., खाली `factorial` चे रिकर्सिव्ह स्वरूप आहे:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

ते आपण असे लिहू शकतो:

```
def factorial(n):
    return 1 if n == 0 else n * factorial(n-1)
```

कंडिश्नल एक्स्प्रेशनचा अजून एक उपयोग म्हणजे पर्यायी अर्ग्युमेंट्स (optional arguments) हाताळणे. उदा., खाली `GoodKangaroo` मधील init मेथड आहे (प्रश्न १७.२ बघा):

```
    def __init__(self, name, contents=None):
        self.name = name
        if contents == None:
            contents = []
        self.pouch_contents = contents
```

ती आपण चांगल्या पद्धतीने अशी लिहू शकतो:

```
    def __init__(self, name, contents=None):
        self.name = name
        self.pouch_contents = [] if contents == None else contents
```

साधारणपणे तुम्ही एक कंडिश्नल स्टेटमेंट एका कंडिश्नल एक्स्प्रेशनने तेव्हा बदलू शकता जेव्हा दोन्ही ब्रांचेसमध्ये (branches, फाटे) साधी एक्स्प्रेशन्स असतात जी एकतर रिटर्न केली जातात किंवा एकाच व्हेरिएबलला असाइन केली जातात.

१९.२ लिस्ट कॉंप्रहेन्शन (List comprehensions)

विभाग १०.७ मध्ये आपण मॅप आणि फिल्टर पॅटर्न्स पाहिले. उदा., खालील फंक्शन स्ट्रिंग्सची एक लिस्ट घेऊन स्ट्रिंगची `capitalize` मेथड एलेमेंट्सना मॅप करून स्ट्रिंग्सची नवीन लिस्ट रिटर्न करते:

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

लिस्ट कॉंप्रहेन्शन (**list comprehension**) वापरून आपण हे जास्ती संक्षेपाने लिहू शकतो:

```
def capitalize_all(t):
    return [s.capitalize() for s in t]
```

ह्याठिकाणी ब्रॅकेट (bracket) ऑपरेटर्स हे दर्शवतात की आपण एक नवीन लिस्ट बनवत आहोत. ब्रॅकेट्समधील (चौकटी कंसातील) एक्स्प्रेशन लिस्टमधील एलेमेंट्स देते आणि आपण कोणता सीक्वेन्स ट्रव्हर्स (traverse) करत आहोत हे `for` लूप-ने व्यक्त केले जाते.

लिस्ट कॉंप्रहेन्शनचा सिंटॅक्स थोडासा विचित्र आहे कारण लूप व्हेरिएबल (ह्याठिकाणी `s`) त्याच्या डेफनिशनच्या आधीच त्या एक्स्प्रेशनमध्ये येते.

लिस्ट कॉंप्रहेन्शन फिल्टरिंग (filtering) साठी सुद्धा वापरता येते. उदा., खालील फंक्शन `t` चे फक्त तेच एलेमेंट्स निवडते जे कॅपिटल (uppercase) आहेत आणि एक नवीन लिस्ट रिटर्न करते:

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

लिस्ट कॉंप्रहेन्शन वापरून आपण ते चांगल्या पद्धतीने असे लिहू शकतो:

```
def only_upper(t):
    return [s for s in t if s.isupper()]
```

लिस्ट कॉंप्रहेन्शन समजायला सोपे आणि संक्षिप्त असते, निदान सोप्या एक्स्प्रेशन्ससाठी तरी. आणि सहसा ते समान (तेच काम करणाऱ्या) `for` लूप-पेक्षा वेगवान असते, कधीकधी खूप वेगवान. तर त्याबद्दल आधी *न* शिकवल्याचा तुम्हाला राग आला असेल तर ते समजण्यासारखे आहे.

पण प्रत्युत्तरात असे म्हणता येईल की लिस्ट कॉंप्रहेन्शन डीबग करायला अवघड असते कारण तुम्ही त्यातील लूप-मध्ये प्रिंट स्टेटमेंट नाही टाकू शकत. तुम्ही ते तेव्हाच वापरा जेव्हा कॉंप्युटेशन इतके साधे असेल की ते पहिल्याच प्रयत्नात बरोबर असण्याची शक्यता जास्ती आहे. आणि नवशिक्यांसाठी त्याचा अर्थ 'कधीच नाही' असा होतो.

१९.३ जनरेटर एक्स्प्रेशन (Generator expressions)

जनरेटर एक्स्प्रेशन (**Generator expression**) लिस्ट कॉंप्रहेन्शनसारखेच असते पण चौकटी कंसाऐवजी साध्या कंसात असते:

```
>>> g = (x**2 for x in range(5))
>>> g
<generator object <genexpr> at 0x7f4c45a786c0>
```

ह्याचे उत्तर एक जनरेटर (generator) ऑब्जेक्ट असते ज्याला त्या व्हॅल्यूझच्या सीक्वेन्समधून कसे इटरेट (iterate) करायचे हे माहीत असते. पण लिस्ट कॉंप्रहेन्शनसारखे ते सर्व व्हॅल्यूझ एकदमच कॉंप्युट नाही करत. ते विचारण्यासाठी थांबते; `next` हे बिल्ट-इन फंक्शन जनरेटरमधून पुढील व्हॅल्यू मिळवते:

```
>>> next(g)
0
>>> next(g)
1
```

सीक्वेन्सच्या शेवटी पोहोचल्यावर `next` हे `StopIteration` एक्सेप्शन रेय्झ (raise) करते. व्हॅल्यूझमधून इटरेट करण्यासाठी तुम्ही `for` लूप-सुद्धा वापरू शकता:

```
>>> for val in g:
...     print(val)
4
9
16
```

जनरेटर ऑब्जेक्ट तो सीक्वेन्समध्ये सध्या कुठे आहे ह्याची नोंद ठेवतो, म्हणून `for` लूप तिथून सुरू होतो जिथे `next` ने सोडले होते. एकदा जनरेटर संपला की तो `StopIteration` रेय्झ करतो:

```
>>> next(g)
StopIteration
```

कधीकधी जनरेटर एक्स्प्रेशन `sum`, `max`, आणि `min` सारख्या फंक्शनमध्ये वापरले जाते:

```
>>> sum(x**2 for x in range(5))
30
```

१९.४ any आणि all (any and all)

पायथॉन `any` नावाचे बूलियन फंक्शन पुरवते जे बूलियन व्हॅल्यूझ-चा एक सीक्वेन्स घेऊन `True` रिटर्न करते जर कोणतीही व्हॅल्यू `True` असेल. ते लिस्टवर चालते:

```
>>> any([False, False, True])
True
```

पण सहसा जनरेटर एक्स्प्रेशनवर वापरले जाते:

```
>>> any(letter == 't' for letter in 'monty')
True
```

हे उदाहरण जास्ती काही कामाचे नाही कारण ते `in` ऑपरेटर जे करतो तेच करते. पण आपण `any` वापरून विभाग ९.३ मधील काही सर्च (search) फंक्शन्स थोड्या चांगल्या पद्धतीने लिहू शकतो. उदा., आपण `avoids` असे लिहू शकतो:

```
def avoids(word, forbidden):
    return not any(letter in forbidden for letter in word)
```

हे फंक्शन तुम्ही जवळजवळ इंग्रजीसारखेच वाचू शकता: 'word avoids forbidden if there are not any forbidden letters in word.'

`any` मध्ये जनरेटर एक्स्प्रेशन वापरणे कार्यक्षम आहे कारण ते `True` व्हॅल्यू मिळाल्यामिळाल्या ताबडतोब थांबते, म्हणजे त्याला पूर्ण सीक्वेन्स शोधायची गरज नाही.

पायथॉन `all` नावाचे अजून एक बिल्ट-इन फंक्शन पुरवते जे सीक्वेन्सचा प्रत्येक एलेमेंट `True` असेल तर `True` रिटर्न करते. सराव म्हणून `all` वापरून विभाग ९.३ मधील `uses_all` परत लिहा.

१९.५ सेट (Sets, संच)

विभाग १३.६ मध्ये आपण डिक्शनरी वापरून एका मजकुरात असलेले पण शब्दयादीत नसलेले शब्द शोधले. तिथले फंक्शन हे मजकुरातील शब्द keys म्हणून असलेली डिक्शनरी `d1` ह्या परॅमीटरमध्ये आणि शब्दयादीतील शब्द `d2` मध्ये घेते. आणि ते `d1` मधील keys असलेली पण `d2` मधल्या keys वगळणारी डिक्शनरी रिटर्न करते.

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

ह्या सर्व डिक्शनरीझमधील व्हॅल्यूझ `None` आहेत कारण आपण त्या कधीच वापरत नाही. ह्याचा परिणाम म्हणजे वाया गेलेली जागा.

पायथॉनमध्ये `set` नावाचा अजून एक बिल्ट-इन टाइप आहे, जो एका डिक्शनरीच्या keys च्या समूहासारखा (व्हॅल्यूझशिवाय) असतो; `set` मध्ये एलेमेंट टाकणे आणि एखादा एलेमेंट त्यात आहे का नाही हे तपासणे ह्या क्रिया जलद असतात. आणि, कॉमन ऑपरेशन्ससाठी सेट मेथड्स आणि ऑपरेटर्स पुरवते.

उदा., सेट वजाबाकी ही `difference` मेथडद्वारे किंवा - ऑपरेटरद्वारे उपलब्ध आहे. तर आपण `subtract` असे लिहू शकतो:

```
def subtract(d1, d2):
    return set(d1) - set(d2)
```

ह्याचे उत्तर डिक्शनरीऐवजी एक सेट आहे पण इटरेशनसारख्या ऑपरेशनसाठी त्यांचे वर्तन सारखेच आहे.

ह्या पुस्तकातील काही प्रश्न सेट वापरून संक्षेपाने आणि प्रभावीपणे सोडवता येऊ शकतात. उदा., प्रश्न १०.७ मधील `has_duplicates` चे हे एक उत्तर आहे जे एक डिक्शनरी वापरते:

```
def has_duplicates(t):
    d = {}
    for x in t:
        if x in d:
            return True
        d[x] = True
    return False
```

जेव्हा एखादा एलेमेंट पहिल्यांदा येतो तेव्हा तो डिक्शनरीमध्ये टाकला जातो. जर तो परत आला तर फंक्शन `True` रिटर्न करते.

सेट वापरून हेच फंक्शन असे लिहिता येईल:

```
def has_duplicates(t):
    return len(set(t)) < len(t)
```

एक एलेमेंट सेटमध्ये एकदाच येऊ शकतो, म्हणजे जर `t` मध्ये एखादा एलेमेंट एकाहून अधिक वेळा असेल तर सेट `t` पेक्षा लहान असेल. जर डुप्लिकेट्स (duplicates) नसतील तर सेट आणि `t` मधील एलेमेंट्सची संख्या सारखीच असेल.

प्रकरण ९ मधील काही प्रश्न सोडवण्यासाठी पण आपण सेट वापरू शकतो. उदा., `uses_only` चे लूप वापरणारे खालील स्वरूप:

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

`uses_only` हे तपासते की `word` मधील सर्व अक्षरे `available` मध्ये आहेत. हे आपण असे तपासू शकतो:

```
def uses_only(word, available):
    return set(word) <= set(available)
```

एक संच दुसऱ्याचा उपसंच (subset) आहे किंवा ते सारखेच आहेत हे <= ऑपरेटर तपासतो, जे खरे असेल जर `word` मधील सर्व अक्षरे `available` मध्ये असतील.

सराव म्हणून `avoids` सेट वापरून लिहा.

१९.६ काउंटर (Counters)

Counter हे सेटसारखा असते पण जर एखादा एलेमेंट एकाहून अधिक वेळेस असेल तर Counter ते किती वेळा आला आहे ह्याची नोंद ठेवते. जर तुम्हाला **मल्टीसेट** (**multiset**) ही गणितीय संकल्पना माहीत असेल तर Counter हे मल्टीसेट व्यक्त करण्याचा नैसर्गिक मार्ग आहे.

Counter `collections` नावाच्या प्रमाण मोड्युलमध्ये दिलेले असल्यामुळे तुम्हाला ते इंपोर्ट करावे लागते. तुम्ही Counter इनिशलाइझ करण्यासाठी एक स्ट्रिंग, लिस्ट, किंवा असे काही ज्यावर इटरेशन चालू शकते हे वापरू शकता:

```
>>> from collections import Counter
>>> count = Counter('parrot')
>>> count
Counter({'r': 2, 't': 1, 'o': 1, 'p': 1, 'a': 1})
```

Counter बरेचसे डिक्शनरीसारखे आहे; ते प्रत्येक key ती किती वेळा आली आहे ह्या संख्येला मॅप करते. डिक्शनरीसारखेच keys हॅशेबल (hashable) असणे गरजेचे आहे.

पण डिक्शनरीच्या विरुद्ध, जर तुम्ही नसलेला एलेमेंट बघायचा प्रयत्न केलात तर Counter एक्सेप्शन रेय्झ (raise) करत नाही. त्याऐवजी, ते 0 रिटर्न करते:

```
>>> count['d']
0
```

आपण Counter वापरून प्रश्न १०.६ मधील `is_anagram` असे लिहू शकतो:

```
def is_anagram(word1, word2):
    return Counter(word1) == Counter(word2)
```

जर दोन शब्द ॲनाग्राम्स असतील तर त्यात सारखीच अक्षरे असतात आणि प्रत्येक अक्षर दोन्ही शब्दांत सारख्याच वेळेला येते, म्हणजे त्यांची Counters सारखीच असतात.

Counters सेटसारखी ऑपरेशन्स असणाऱ्या मेथड्स आणि ऑपरेटर्स उपलब्ध करते, जसे बेरीज, वजाबाकी, संयोग (union), आणि छेद (intersection). आणि त्यातली अजून एक चांगली आणि उपयोगी मेथड म्हणजे `most_common`, जी वारंवारतेच्या उतरत्या क्रमाने असलेली व्हॅल्यू-वारंवारता जोड्यांची (pairs) लिस्ट रिटर्न करते (म्हणजेच सर्वांत कॉमन पासून ते सर्वांत कमी कॉमन पर्यंत):

```
>>> count = Counter('parrot')
>>> for val, freq in count.most_common(3):
...     print(val, freq)
r 2
p 1
a 1
```

१९.७ defaultdict

`collections` मोड्युल `defaultdict` सुद्धा पुरवते, जी डिक्शनरीसारखीच असते पण जर तुम्ही तिच्यात नसणारी key बघायचा किंवा वापरायचा प्रयत्न केलात तर ती नवीन व्हॅल्यू निर्माण करते.

जेव्हा तुम्ही defaultdict बनवता तेव्हा तुम्ही नवीन व्हॅल्यूझ बनवणारे फंक्शन पाठवता. ऑब्जेक्ट्स बनवण्यासाठी वापरल्या जाणाऱ्या फंक्शनला कधीकधी **फॅक्टरी** (**factory**) म्हणतात. लिस्ट्स, सेट्स, आणि इतर टाइप्स बनवणारी बिल्ट-इन फंक्शन्स फॅक्टरी म्हणून वापरता येतात:

```
>>> from collections import defaultdict
>>> d = defaultdict(list)
```

लक्षात घ्या की अर्ग्युमेंट `list` आहे, जे एक क्लास ऑब्जेक्ट आहे, `list()` नाही, जी एक नवीन लिस्ट आहे. जोपर्यंत नसलेली key तुम्ही बघत किंवा वापरत नाहीत तोपर्यंत तुम्ही दिलेले फंक्शन कॉल होत नाही.

```
>>> t = d['new key']
>>> t
[]
```

जिला आपण `t` म्हणतोय ती नवीन लिस्टसुद्धा defaultdict मध्ये टाकली जाते. म्हणजेच जर आपण `t` बदलले, तर `d` मध्ये सुद्धा तो बदल दिसून येतो:

```
>>> t.append('new value')
>>> d
defaultdict(<class 'list'>, {'new key': ['new value']})
```

जर तुम्ही लिस्ट्सची डिक्शनरी वापरत असाल तर सहसा तुम्ही `defaultdict` वापरून तो कोड सोप्या पद्धतीन लिहू शकता. प्रश्न १२.२ च्या दिलेल्या उत्तरात (ज्याची लिंक पुढे आहे), आपण एक डिक्शनरी बनवतो जी अक्षरांची सॉर्टेड स्ट्रिंग त्या अक्षरांपासून बनवता येऊ शकणाऱ्या शब्दांच्या लिस्टला मॅप करते (उत्तराची लिंक: http://thinkpython2.com/code/anagram_sets.py).

उदा., `'opst'` ही key `['opts', 'post', 'pots', 'spot', 'stop', 'tops']` ह्या लिस्टला मॅप होते.

मूळ कोड असा:

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
```

```
        if t not in d:
            d[t] = [word]
        else:
            d[t].append(word)
    return d
```

ह्याला `setdefault` ने सोपे करता येते, जी तुम्ही प्रश्न ११.२ मध्ये वापरली असेल:

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d.setdefault(t, []).append(word)
    return d
```

ह्या उत्तराची एक कमतरता म्हणजे त्यात प्रत्येक वेळेस नवीन लिस्ट बनवली जाते, जरी तिची गरज नसेल तरी. लिस्ट बनवणे ही मोठी गोष्ट नाही, पण जर फॅक्टरी फंक्शन गुंतागुंतीचे असेल, तर मोठा फरक पडू शकतो.

हे टाळण्यासाठी आपण defaultdict वापरू शकतो, ज्याने कोड पण सोपा होतो:

```
def all_anagrams(filename):
    d = defaultdict(list)
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d[t].append(word)
    return d
```

प्रश्न १८.३ च्या दिलेल्या उत्तरात (जे तुम्ही पुढील लिंकवरून डाऊनलोड करू शकता) `setdefault` ही `has_straightflush` मध्ये वापरली आहे: http://thinkpython2.com/code/PokerHandSoln.py.

ह्या उत्तरातही लूपमधून प्रत्येकवेळी जाताना `Hand` ऑब्जेक्ट बनवला जातो, जरी त्याची गरज नसेल तरी. सराव म्हणून ते defaultdict वापरून लिहा.

१९.८ नावांसहित टपल (Named tuples)

अनेक साधे ऑब्जेक्ट्स साधारणपणे संबंधित व्हॅल्यूझ-चा संग्रह असतो. उदा., प्रकरण १५ मध्ये पाहिलेल्या Point ऑब्जेक्टमध्ये दोन संख्या असतात, `x` आणि `y`. जेव्हा तुम्ही असा क्लास डिफाइन करता तेव्हा तुम्ही सहसा init आणि `str` मेथड्सनी सुरुवात करता:

```
class Point:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return '(%g, %g)' % (self.x, self.y)
```

ह्या कोड-ची लांबी त्यातील माहितीच्या तुलनेने जरा जास्तीच आहे. पायथॉनमध्ये हीच गोष्ट संक्षेपाने साध्य करण्यासाठी एक मार्ग आहे:

```
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
```

पहिले अर्ग्युमेंट हे तुम्हाला जो क्लास बनवायचा आहे त्याचे (स्ट्रिंगस्वरुपातील) नाव असते. आणि दुसरे हे Point ऑब्जेक्टमध्ये तुम्हाला पाहिजे असलेल्या ॲट्रिब्युट्सच्या (स्ट्रिंगस्वरुपातील) नावांची लिस्ट असते. ह्या `namedtuple` ची रिटर्न व्हॅल्यू एक क्लास ऑब्जेक्ट असते:

```
>>> Point
<class '__main__.Point'>
```

`Point` आपोआप `__init__` आणि `__str__` सारख्या मेथड्स देतो, म्हणजे तुम्हाला त्या बनवायची गरज नसते.

Point ऑब्जेक्ट बनवण्यासाठी तुम्ही Point क्लास एका फंक्शनसारखा वापरता:

```
>>> p = Point(1, 2)
>>> p
Point(x=1, y=2)
```

तुम्ही दिलेल्या नावांच्या ॲट्रिब्युट्सना init मेथड दिलेले अर्ग्युमेंट्स असाइन (assign) करते. आणि `str` मेथड Point ऑब्जेक्टचे आणि त्यातील ॲट्रिब्युट्सचे स्ट्रिंगरूप रिटर्न करते.

तुम्ही namedtuple चा कोणताही एलेमेंट नावाने बघू किंवा वापरू शकता:

```
>>> p.x, p.y
(1, 2)
```

पण तुम्ही namedtuple टपलसारखे सुद्धा वापरू शकता:

```
>>> p[0], p[1]
(1, 2)

>>> x, y = p
>>> x, y
(1, 2)
```

namedtuple वापरून चटकन साधे क्लासेस डिफाइन करता येतात. पण गोम अशी आहे की साधा क्लास नेहमी साधाच नाही राहत. एकदा namedtuple वापरून क्लास बनवल्यावर नंतर तुम्हाला असे वाटू शकते की अरे ह्यात आपल्याला काही मेथड्स लागतील. तसे झाले तर तुम्ही त्या namedtuple कडून इनहेरिट करणारा नवीन क्लास बनवू शकता:

```
class Pointier(Point):
    # add more methods here
```

किंवा तुम्ही परंपरागत क्लास डेफनिशनला स्थलांतर करू शकता.

१९.९ कीवर्ड अर्ग्युमेंट्स जमा करणे (Gathering keyword arguments)

विभाग १२.४ मध्ये आपण पाहिले की स्वतःची अर्ग्युमेंट्स एका टपलमध्ये जमा करणारे फंक्शन कसे लिहायचे:

```
def printall(*args):
    print(args)
```

तुम्ही हे फंक्शन कितीही पोझिशनल अर्ग्युमेंट्सनी (positional arguments, म्हणजेच अशी अर्ग्युमेंट्स ज्यांत कीवर्ड नसतो) कॉल करू शकता:

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

पण * ऑपरेटर कीवर्ड अर्ग्युमेंट्स जमा नाही करत:

```
>>> printall(1, 2.0, third='3')
TypeError: printall() got an unexpected keyword argument 'third'
```

कीवर्ड अर्ग्युमेंट्स जमा करण्यासाठी तुम्ही ** ऑपरेटर वापरू शकता:

```
def printall(*args, **kwargs):
    print(args, kwargs)
```

कीवर्ड जमा करणाऱ्या परॅमीटरला तुम्ही काहीही नाव देऊ शकता, पण सहसा `kwargs` हे नाव दिले जाते. ह्याचा परिणाम म्हणजे कीवर्ड त्याच्या व्हॅल्यूला मॅप करणारी डिक्शनरी:

```
>>> printall(1, 2.0, third='3')
(1, 2.0) {'third': '3'}
```

जर तुमच्याकडे कीवर्ड्स आणि व्हॅल्यूझ-ची डिक्शनरी असेल तर तुम्ही स्कॅटर (scatter) ऑपरेटर, **, वापरून फंक्शन कॉल करू शकता:

```
>>> d = dict(x=1, y=2)
>>> Point(**d)
Point(x=1, y=2)
```

स्कॅटर ऑपरेटरशिवाय ते फंक्शन `d` ला एका पोझिशनल अर्ग्युमेंट समजेल, आणि ते अर्ग्युमेंट `x` ला असाइन करून तक्रार करेल की `y` ला असाइन करायला काहीच नाहीये:

```
>>> d = dict(x=1, y=2)
>>> Point(d)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __new__() missing 1 required positional argument: 'y'
```

जेव्हा तुम्ही खूप परॅमीटर्स असलेली फंक्शन्स वापरता, तेव्हा सारखे वापरले जाणारे पर्याय डिक्शनरीमध्ये टाकून ती वापरणे कधीकधी फायदेशीर ठरते.

१९.१० शब्दार्थ

कंडिश्नल एक्स्प्रेशन (conditional expression): एक असे एक्स्प्रेशन ज्याची दोनपैकी एक व्हॅल्यू असते जी दिलेल्या कंडिशनवरून ठरते.

लिस्ट कॉंप्रहेन्शन (list comprehension): चौकटी कंसातील `for` लूप असणारे एक एक्स्प्रेशन जे नवीन लिस्ट बनवून देते.

जनरेटर एक्स्प्रेशन (generator expression): साध्या कंसातील `for` लूप असणारे एक एक्स्प्रेशन जे नवीन जनरेटर (generator) ऑब्जेक्ट बनवून देते.

मल्टीसेट (multiset): एका संचातील घटक (elements, एलेमेंट्स) ते किती वेळा येतात हे व्यक्त करणारी एक गणितीय संकल्पना.

फॅक्टरी (factory): ऑब्जेक्ट्स बनवण्यासाठी वापरले जाणारे फंक्शन, जे सहसा अर्ग्युमेंट म्हणून पाठवले जाते.

१९.११ प्रश्नसंच (Exercises)

प्रश्न १९.१. *खालील फंक्शन द्विपद सहगुणक (binomial coefficient, बायनॉमिअल कोइफिशंट) रिकर्सिव्हली (recursively) कॉंप्युट करते.*

```
def binomial_coeff(n, k):
    """Compute the binomial coefficient "n choose k".

    n: number of trials
```

```
    k: number of successes

    returns: int
    """
    if k == 0:
        return 1
    if n == 0:
        return 0

    res = binomial_coeff(n-1, k) + binomial_coeff(n-1, k-1)
    return res
```

हे फंक्शन एकात एक कंडिश्नल एक्स्प्रेशन्स (nested conditional expressions) वापरून परत लिहा.

एक नोंद: हे फंक्शन त्याच किंमती सारख्यासारख्या काँप्युट करत असल्यामुळे अकार्यक्षम आहे. तुम्ही त्याचे (विभाग ११.६ मध्ये आपण बघितल्याप्रमाणे) 'मेमो'करण (memoization) करून त्याला वेगवान बनवू शकता. पण तुमच्या हे लक्षात येईल की जर तुम्ही कंडिश्नल एक्स्प्रेशन वापरून हे फंक्शन लिहिले तर त्याचे 'मेमो'करण करणे अवघड आहे.

परिशिष्ट १

डीबगिंग (Debugging)

डीबगिंग करत असताना एरर लवकर हेरण्यासाठी वेगवेगळ्या प्रकारच्या एरर्समधील फरक तुम्हाला करता आला पाहिजे:

- सिंटॅक्स एरर्स हे सोर्स कोड (source code) चे बाइट कोड (byte code) मध्ये रुपांतर करताना इंटरप्रिटर शोधतो. ते एरर्स हे दर्शवतात की प्रोग्रामच्या संरचनेमध्ये काहीतरी बिघाड आहे. उदा., `def` स्टेटमेंटच्या शेवटी अपूर्णविराम (म्हणजे colon, `:` चिन्ह) विसरल्यास `SyntaxError: invalid syntax` असा अनावश्यक मेसेज मिळतो.

- रनटाइम (runtime) एरर्स हे प्रोग्राम चालत असताना काही गडबड झाल्यास इंटरप्रिटर दाखवतो. बहुतांश रनटाइम एरर मेसेजेस एरर कुठे आला आणि त्या क्षणी कोणती फंक्शन्स रन होत होती ह्याविषयी माहिती दाखवतात. उदा., इन्फिनेट रिकर्शनचा (infinite recursion) शेवट 'maximum recursion depth exceeded' ह्या रनटाइम एररने होतो.

- सिमँटिक एरर्स हे त्या प्रोग्राममधील चुका दर्शवतात जो कोणताही एरर मेसेज *न* दाखवता चालतो पण आपल्याला पाहिजे ती गोष्ट करत नाही. उदा., एखादे एक्स्प्रेशन तुमच्या अपेक्षित क्रमाने इव्हॅल्यूएट (evaluate) *न* झाल्यामुळे आलेले चुकीचे उत्तर.

डीबगिंगमधील पहिली पायरी म्हणजे तुमच्यासमोर कोणत्या प्रकारचा एरर बसलेला आहे हे ओळखणे. खालील विभाग हे जरी एररच्या प्रकारावर आधारित असले तरी काही तंत्रे एकाहून अधिक ठिकाणी वापरली जाऊ शकतात.

प.१.१ सिंटॅक्स एरर (Syntax errors)

सिंटॅक्स एरर्स कुठे आहेत हे समजल्यावर ते दुरूस्त करणे सोपे असते. दुर्दैवाने, कधीकधी एरर-मेसेज उपयोगी नसतो. सर्वांत कॉमन एरर मेसेजेस म्हणजे `SyntaxError: invalid syntax` आणि `SyntaxError: invalid token`, आणि हे दोन्हीही खूप काही कामाचे नाहीत.

पण दुसऱ्या बाजूस, मेसेज तुम्हाला हे नक्की सांगतो की प्रोग्राममध्ये चूक झाली कुठे. खरे तर, त्यात हे समजते की पायथॉनला चूक कुठे सापडली, पण खरा एरर तिकडेच असेल असे नाही. कधीकधी मेसेज सांगतो त्याच्या कुठेतरी आधी एरर असतो, बहुतेक मागच्याच ओळीवर.

जर तुम्ही हळूहळू प्रोग्रामची बांधणी करत असाल तर तुम्हाला एक साधारण कल्पना असते की एरर कुठे आहे. नुकत्याच लिहिलेल्या ओळीवरच तो असेल.

जर तुम्ही पुस्तकातून कोड कॉपी करत असाल तर तुमच्या आणि पुस्तकातल्या कोड-ची काळजीपूर्वक तुलना करा. अक्षर आणि अक्षर तपासा. त्याचवेळी, हे लक्षात असू द्या की पुस्तकात चूक असणे अशक्य नाही, म्हणजे जर तुम्हाला सिंटॅक्स एरर असेल असे काही दिसले तर तो असू शकतो.

सर्वांत कॉमन सिंटॅक्स एरर्स टाळण्याचे काही मार्ग खाली दिले आहेत:

१. तुम्ही कोणताही पायथॉन कीवर्ड व्हेरिएबलचे नाव म्हणून वापरत नाही आहात ह्याची खात्री करा.

२. प्रत्येक कंपाउंड स्टेटमेंट (compound statement) च्या हेडर (header) च्या शेवटी अपूर्णविराम (म्हणजे colon, `:` चिन्ह) ह्याची खात्री करा (उदा., `for`, `while`, `if`, आणि `def` ही स्टेटमेंट्स.).

३. कोड-मधील सर्व स्ट्रिंग्स अवतरण चिन्हांच्या जोडीत आहेत ह्याची खात्री करा. सगळी अवतरण चिन्हे 'सरळ आणि उभी' असून 'वळणदार अवतरण चिन्हे' नाहीत ह्याची खात्री करा.

४. जर एकेरी किंवा दुहेरी अवतरण चिन्ह तीन वेळा (म्हणजे `'''` किंवा `"""`) वापरून सुरू केलेली अनेक ओळी व्यापणारी (multiline) स्ट्रिंग तुम्ही वापरत असाल तर ती स्ट्रिंग व्यवस्थितपणे संपवली आहे हे तपासा. *न* संपवलेली स्ट्रिंग प्रोग्रामच्या शेवटी `invalid token` एरर देऊ शकते. किंवा प्रोग्रामचा त्यानंतरचा पुढील स्ट्रिंगपर्यंतचा भाग त्या स्ट्रिंगचाच भाग आहे असे समजले जाऊ शकते, म्हणजेच तुम्हाला कदाचित एरर मेसेज देखील मिळणार नाही!

५. बंद *न* केलेल्या ऑपरेटरमुळे—`(`, `{`, किंवा `[`—पायथॉन पुढच्या ओळीला सध्याच्या स्टेटमेंटचा भाग समजतो. साधारणपणे एरर ताबडतोब पुढील ओळीवरच येतो.

६. कंडिश्नलमध्ये == ऐवजी = तर नाही ना हे तपासा (प्रत्येकाने कधीनाकधी केलेला एरर).

७. इंडेंटेशन (indentation) तपासून बघा की ते पाहिजे तसे जुळते आहे की नाही. पायथॉनमध्ये स्पेस (space) आणि टॅब (tab) चालते, पण तुम्ही दोन्ही वापरले तर त्यामुळे एरर येऊ शकतो. हे टाळण्याचा उत्तम मार्ग म्हणजे पायथॉन प्रोग्रामवर नीट चालणारा टेक्स्ट एडिटर (text editor) वापरणे जो संगतवार इंडेंटेशन देतो.

८. तुमच्या कोड-मध्ये किंवा त्यातील स्ट्रिंग्स आणि कॉमेंट्समध्ये जर ASCII प्रकारात न मोडणारी अक्षरे असतील (उदा., देवनागरी), तर त्यामुळे एरर येऊ शकतो, जरी पायथॉन ३ सहसा असली अक्षरे व्यवस्थित हाताळत असला तरी. इंटरनेट किंवा अन्य कुठून कॉपी-पेस्ट (copy-paste) करताना काळजी घ्या.

जर काहीच चालले नाही तर पुढच्या विभागाकडे कूच करा...

प.१.१.१ कितीही बदल केले तरी त्याने काहीच फरक पडत नाहीये.

जर इंटरप्रिटर म्हणाला की एरर आहे आणि तो तुम्हाला दिसत नाही, तर त्याचे कारण हेही असू शकते की तुम्ही दोघे वेगळ्या कोड-कडे बघत आहात. तुमचे प्रोग्रामिंग एनव्हायर्नमेंट (programming environment) तपासून ह्याची खात्री करा की तुम्ही ज्या प्रोग्राममध्ये बदल करत आहात तोच पायथॉन रन करतोय.

जर तुम्हाला शंका असेल, तर प्रोग्रामच्या सुरुवातील मुद्दामहून स्पष्ट असा सिंटॅक्स एरर टाका. परत रन करा. जर इंटरप्रिटरला नवीन एरर नाही मिळाला, तर तुम्ही नवीन कोड रन नाही करत आहात हे सिद्ध होते.

काही संभाव्य अपराधी म्हणजे:

- तुम्ही फाइल बदलली पण रन करायच्या आधी ती सेव्ह (save) करायला विसरलात. काही प्रोग्रामिंग एनव्हायर्नमेंट्स तुमच्यासाठी हे करतात, पण काही नाही करत.
- तुम्ही फाइलचे नाव बदलले, पण तुम्ही अजूनही जुन्या नावाचीच फाइल रन करत आहात.
- तुमच्या डेव्हेलपमेंट एनव्हायर्नमेंट (development environment) मध्ये काहीतरी चुकीची सेटिंग झाली आहे.

- जर तुम्ही एक मोड्युल लिहित असाल आणि `import` वापरत असाल तर तुम्ही त्या मोड्युलला पायथॉनच्या कोणत्यातरी मोड्युलसारखेच नाव तर नाही दिले ना ह्याची खात्री करा.

- जर तुम्ही एखादे मोड्युल वापरण्यासाठी `import` वापरत असाल तर हे लक्षात असू द्या की बदललेली मोड्युल फाइल वापरण्यासाठी तुम्हाला इंटरप्रिटर पुन्हा सुरू करावा लागेल किंवा `reload` वापरावे लागेल. जर तुम्ही परत इंपोर्ट केले तर त्याने काही फरक पडत नाही.

जर तुम्ही अडलेले आहात आणि काय चालू आहे हे तुम्हाला समजत नाहीये तर एक मार्ग असा आहे की 'Hello, World!' सारख्या एका नवीन प्रोग्रामने सुरुवात करून काहीतरी रन होतेय ह्याची खात्री करणे. मग हळूहळू मूळ प्रोग्रामची त्यात तुकड्यातुकड्याने तुम्ही भर घालू शकता.

प.१.२ रनटाइम एरर (Runtime errors)

तुमच्या प्रोग्रामचा सिंटॅक्स एकदाचा बरोबर झाला की पायथॉन त्याला वाचून कमीतकमी चालवायला सुरुवात तरी करू शकतो. अशी कोणतीच चूक होऊ शकते?

प.१.२.१ माझा प्रोग्राम काहीच करत नाही.

जेव्हा तुमच्या फाइलमध्ये फंक्शन्स आणि क्लासेस असतात पण त्यात एक्सेक्युशन सुरू करणारा कोणताही फंक्शन कॉल नसतो असे असताना ही समस्या येणे सर्वांत कॉमन आहे. हे मुद्दाम केलेले असू शकते जर तुम्हाला हे मोड्युल फक्त क्लासेस आणि फंक्शन्स पुरवायला इंपोर्ट करायचे असेल.

पण हे जर मुद्दाम केलेले नसेल तर प्रोग्राममध्ये एक फंक्शन कॉल आहे ह्याची आणि फ्लो-ऑफ-एक्सेक्युशन तिथपर्यंत पोहोचतोय ह्याची खात्री करा (खाली 'फ्लो-ऑफ-एक्सेक्युशन' बघा).

प.१.२.२ माझा प्रोग्राम अडकतोय (My program hangs).

जर एखादा प्रोग्राम थांबून काहीच करत नाहीये असे भासवत असेल तर तो 'हँग' ('hang', अडकणे) झाला आहे असे म्हणतात. सहसा त्याचा अर्थ असा होतो की तो एका इन्फिनेट लूप-मध्ये किंवा इन्फिनेट रिकर्शनमध्ये अडकला आहे.

- जर एका विशिष्ठ लूप-वर तुम्हाला शंका येत असेल तर त्याच्या अगदी आधी 'entering the loop' म्हणणारे प्रिंट स्टेटमेंट टाका आणि अगदी नंतर 'exiting the loop' असे.

  प्रोग्राम रन करा. जर तुम्हाला पहिला मेसेज मिळाला पण दुसरा नाही तर तोच लूप इन्फिनेट लूप आहे. खाली 'इन्फिनेट लूप' ('Infinite Loop') भाग बघा.

- बहुतांश वेळा इन्फिनेट रिकर्शनमुळे प्रोग्राम थोडा वेळ चालून 'RuntimeError: Maximum recursion depth exceeded' एरर दाखवेल. जर तसे झाले तर खाली 'इन्फिनेट रिकर्शन' ('Infinite Recursion') भाग बघा.

  जर तुम्हाला हा एरर मिळत नसेल पण तुम्हाला अशी शंका असेल की एका रिकर्सिव्ह मेथड किंवा फंक्शनमध्येच चूक आहे, तरीसुद्धा तुम्ही खालच्या 'इन्फिनेट रिकर्शन' भागातली तंत्रे वापरू शकता.

- जर ह्या दोन्हींपैकी काहीच चालत नसेल तर इतर लूप आणि रिकर्सिव्ह मेथड्स आणि फंक्शन्सवर चाचण्या करा.

- जर तेही चालत नसेल तर हे शक्य आहे की तुम्हाला तुमच्या प्रोग्रामचा फ्लो-ऑफ-एक्सेक्युशन समजला नाहीये. खाली 'फ्लो-ऑफ-एक्सेक्युशन' ('Flow of Execution') भाग बघा.

इन्फिनेट लूप (Infinite Loop)

जर तुम्हाला असे वाटत असेल की प्रोग्राममध्ये एक इन्फिनेट लूप आहे आणि तुम्हाला अंदाज आहे तो कोणता आहे, तर कंडिशनमधील व्हेरिएबल्सच्या व्हॅल्यूझ प्रिंट करणारे स्टेटमेंट त्या लूप-च्या शेवटी टाका.

उदा.:

```
while x > 0 and y < 0 :
    # do something to x
    # do something to y

    print('x: ', x)
    print('y: ', y)
    print("condition: ", (x > 0 and y < 0))
```

आता तुम्ही जेव्हा प्रोग्राम रन कराल, तुम्हाला लूपमधून प्रत्येकवेळी जाताना आउटपुटच्या तीन ओळी दिसतील. शेवटच्या वेळी ती कंडिशन `False` असली पाहिजे. जर लूप चालत राहिला तर तुम्हाला `x` आणि `y` च्या व्हॅल्यूझ दिसतील आणि समजेल की त्या का व्यवस्थितपणे अपडेट होत नाहीयेत.

इन्फिनेट रिकर्शन (Infinite Recursion)

बहुतांशवेळा इन्फिनेट रिकर्शनमुळे प्रोग्राम थोडा वेळ चालून `Maximum recursion depth exceeded` एरर दाखवतो.

जर तुम्हाला शंका असेली की एखाद्या फंक्शनमुळे इन्फिनेट रिकर्शन होतेय, तर बेस-केस (base case) असल्याची खात्री करा. तिथे अशी कंडिशन असली पाहिजे ज्यामुळे फंक्शन रिकर्सिव्ह कॉल *न* करता रिटर्न करेल. जर नसेल तर तुम्हाला अल्गोरिदम चा पुनर्विचार करून बेस-केस शोधावी लागेल.

जर बेस-केस असेल पण प्रोग्राम तिथपर्यंत पोहोचत नसेल तर फंक्शनच्या सुरुवातीला पॅरॅमीटर्स प्रिंट करणारे एक स्टेटमेंट टाका. आता तुम्ही जर प्रोग्राम रन केलात तर तुम्हाला जेव्हा जेव्हा फंक्शन कॉल होईल तेव्हा तेव्हा आउटपुटच्या काही ओळी दिसतील, आणि तुम्हाला पॅरॅमीटर-व्हॅल्यूझ दिसतील. जर त्या व्हॅल्यूझ बेस-केसच्या जवळ जात नसतील, तर तुम्हाला तसे का होतेय ह्याची कल्पना येईल.

फ्लो-ऑफ-एक्सेक्युशन (Flow of Execution)

जर तुम्हाला प्रोग्राममधून फ्लो-ऑफ-एक्सेक्युशन कसा वाहतोय ह्याची कल्पना नसेल तर प्रत्येक फंक्शनच्या सुरुवातीला 'entering function `foo`' ह्याप्रकारचे `print` स्टेटमेंट टाका, जिथे `foo` हे त्या फंक्शनचे नाव आहे.

आता जेव्हा तुम्ही प्रोग्राम चालवाल, तेव्हा ते प्रत्येक कॉल होणाऱ्या फंक्शनचा माग घेईल.

प.१.२.३ प्रोग्राम रन केल्यावर मला एक्सेप्शन मिळते

रनटाइमला काही गडबड झाली की पायथॉन एक मेसेज प्रिंट करतो ज्यात एक्सेप्शनचे नाव, प्रोग्रामची ओळ ज्यावर एक्सेप्शन आले, आणि ट्रेसबॅक (traceback) ही माहिती असते.

ट्रेसबॅकमध्ये तेव्हा चालू असलेले फंक्शन, नंतर ते फंक्शन कॉल करणारे फंक्शन, आणि नंतर *ते* फंक्शन कॉल करणारे फंक्शन, इत्यादींची यादी असते. दुसऱ्या शब्दांत: कोणत्या क्रमाने फंक्शन्स कॉल होऊन तेव्हा चालू असलेल्या फंक्शनपर्यंत तुम्ही पोहोचलात ह्याचा माग ट्रेसबॅक घेते, आणि तेसुद्धा प्रत्येक फंक्शन कॉलच्या ओळ क्रमांकासहित.

पहिली पायरी म्हणजे प्रोग्राममध्ये ज्याठिकाणी एक्सेप्शन आले त्याठिकाणी कोड तपासून तुम्हाला काय झाले आहे हे समजते आहे का हे बघणे. खाली काही कॉमन रनटाइम एरर्स आहेत:

NameError: सध्याच्या एनव्हायर्नमेंटमध्ये अस्तित्वात नसलेले व्हेरिएबल वापरण्याचा प्रयत्न तुम्ही करत आहात. नावाची स्पेलिंग बरोबर किंवा सुसंगतपणे आहे का हे तपासा. हे लक्षात असू द्या की स्थानिक (local) व्हेरिएबल स्थानिक असते; ते ज्या फंक्शनमध्ये डिफाइन केले आहे त्याच्या बाहेर तुम्ही वापरू शकत नाहीत.

TypeError: ह्याची अनेक कारणे आहेत:

- एखादी व्हॅल्यू तुम्ही अयोग्यपणे वापरत आहात. उदा., स्ट्रिंग, लिस्ट, किंवा टपलला इंटिजर सोडून इतर कशाने इंडेक्स करायचा प्रयत्न करणे.
- फॉरमॅट स्ट्रिंग (format string) आणि पाठवलेले आयटम्स ह्यांच्यात काही विसंगती आहे. जर लागतील त्यापेक्षा जास्ती किंवा कमी आयटम्स असतील किंवा कोणत्या तरी आयटमचे रुपांतर अवैध असेल तर हे होऊ शकते.
- फंक्शनला पाठवलेल्या अर्ग्युमेंट्सच्या संख्येत विसंगती आहे (कमी किंवा जास्ती अर्ग्युमेंट्स पाठवली गेली आहेत). मेथडच्या बाबतीत: मेथड डेफनिशन बघा आणि तपासा की पहिला परॅमीटर `self` आहे. नंतर मेथड कशी इन्व्होक केली गेली आहे ते बघा; योग्य टाइपच्या ऑब्जेक्टवर ती मेथड इन्व्होक होत आहे आणि इतर अर्ग्युमेंट्स योग्यप्रकारे पाठवली गेली आहेत ह्याची खात्री करा.

KeyError: तुम्ही डिक्शनरीतील एलेमेंट बघण्याच्या उद्देशाने त्या डिक्शनरीमध्ये नसलेली key वापरली आहे. जर keys स्ट्रिंग्स असल्या तर हे लक्षात ठेवा की कॅपिटल आणि स्मॉल (uppercase, lowercase) ह्यांत फरक असतो.

AttributeError: अस्तित्वात नसलेले ॲट्रिब्युट किंवा मेथड वापरायचा प्रयत्न तुम्ही केला आहे. स्पेलिंग तपासा! तुम्ही `vars` हे बिल्ट-इन फंक्शन वापरून कोणते ॲट्रिब्युट्स आहेत हे बघू शकता.

जर AttributeError असे म्हणत असेल की एखाद्या ऑब्जेक्टचा `NoneType` आहे, तर त्याचा अर्थ हा की तो ऑब्जेक्ट `None` आहे. तर चूक ॲट्रिब्युटच्या नावात नाही तर ऑब्जेक्टमध्ये आहे.

तो ऑब्जेक्ट `None` असण्याचे कारण हे असू शकते की तुम्ही एखाद्या फंक्शनमधून व्हॅल्यू रिटर्न करायला विसरलात; जर फंक्शनच्या शेवटी पोहोचेपर्यंत `return` स्टेटमेंट लागले नाही तर ते `None` रिटर्न करते. अजून एक कॉमन कारण म्हणजे `sort` सारख्या `None` रिटर्न करणाऱ्या लिस्ट मेथडची रिटर्न व्हॅल्यू वापरणे.

IndexError: तुम्ही लिस्ट, स्ट्रिंग, किंवा टपल ह्यांसारख्या सीक्वेन्सची जी इंडेक्स वापरायचा प्रयत्न करत आहात ती इंडेक्स त्या सीक्वेन्सची लांबी वजा एक ह्यापेक्षा जास्ती आहे. एररस्थानाच्या अगदी आधी एक `print` स्टेटमेंट टाकून इंडेक्सची व्हॅल्यू आणि सीक्वेन्सची लांबी प्रिंट करा. सीक्वेन्स बरोबर लांबीचा आहे का? इंडेक्सची व्हॅल्यू बरोबर आहे का?

पायथॉन डीबगर (debugger) `pdb` एक्सेप्शनला उजेडात आणायला उपयोगी आहे कारण त्यात एक्सेप्शनच्या अगदी आधीची प्रोग्रामची स्थिती तुम्ही सविस्तरपणे तपासू शकता. `pdb` बद्दल तुम्ही पुढील लिंकवर वाचू शकता: https://docs.python.org/3/library/pdb.html.

प.१.२.४ मी इतकी `print` स्टेटमेंट्स टाकलीत की माझ्यावर आउटपुटचा भडिमार होत आहे.

डीबगिंगसाठी `print` स्टेटमेंट्स वापरल्याने कधीकधी आउटपुटचा ढिगारा तुमच्यावर कोसळू शकतो. ह्याठिकाणी दोन पर्याय आहेत: आउटपुट ठीक करा किंवा प्रोग्राम ठीक करा.

आउटपुट सोपे करण्यासाठी तुम्ही काही निरुपयोगी `print` स्टेटमेंट्स काढू शकता किंवा कॉमेंट करू शकता, किंवा त्यांना एकत्र करू शकता, किंवा आउटपुट फॉरमॅट करून समजायला सोपे होईल असे करू शकता.

प्रोग्राम सोपा करण्यासाठी तुम्ही अनेक गोष्टी करू शकता. पहिली म्हणजे प्रोग्राम जो प्रॉब्लेम सोडवत आहे त्याचा व्याप कमी करणे. उदा., जर तुम्ही लिस्ट सर्च करत असाल तर एक *लहान* लिस्ट सर्च करा. जर प्रोग्राम युझरकडून इनपुट घेत असेल तर त्याला सर्वांत सोपे इनपुट द्या ज्याने एरर येईल.

दुसरी म्हणजे प्रोग्रामची साफसफाई. परिणामशून्य कोड काढून प्रोग्रामची पुनर्रचना करा जेणेकरून तो समजायला शक्य तितका सोपा होईल. उदा., जर तुम्हाला शंका असेल की चूक ही प्रोग्रामच्या एका अतिशय गुंतागुंतीच्या भागात

आहे, तर तो भाग सोप्या रचनेने लिहून बघा. जर तुमची शंका एका मोठ्या फंक्शनवर असेल तर त्याचे तुकडे करून अनेक लहान फंक्शन्स बनवा आणि त्यांची स्वतंत्रपणे टेस्टिंग करा.

सहसा सर्वांत लहान टेस्ट-केस (test case) तुम्हाला बग-कडे घेऊन जाते. जर तुम्हाला असे आढळले की प्रोग्राम एका परिस्थितीत चालतोय पण दुसऱ्या परिस्थितीत नाही तर त्यावरून तुम्हाला अंदाज येऊ शकतो की काय चालू आहे.

त्याचप्रमाणे, कोड-चा काही भाग परत लिहिताना तुम्हाला सूक्ष्म बग्स मिळू शकतात. जर तुम्ही असा बदल केला ज्याने प्रोग्राममध्ये काही फरक पडायला नाही पाहिजे असे तुम्हाला वाटते, आणि तरी फरक पडत असेल तर तुम्हाला टीप मिळाली पाहिजे.

प.१.३ सिमँटिक एरर्स (Semantic errors)

एकप्रकारे सिमँटिक एरर डीबग करणे सर्वांत अवघड आहे कारण काय चुकले आहे ह्याविषयी इंटरप्रिटर काहीच माहिती देत नाही. फक्त तुम्हालाच माहीत असते की प्रोग्रामने काय केले पाहिजे.

पहिली पायरी म्हणजे प्रोग्रामचा कोड आणि त्याची दर्शनीय वर्तणूक ह्यांतील संबंध प्रस्थापित करणे. प्रोग्राम खरेच काय करतोय ह्याबद्दल एक हायपॉथसिस (hypothesis, गृहीतक) बनवले पाहिजे. हे करणे अवघड असण्याचे एक कारण म्हणजे काँप्युटर अतिशय वेगाने चालतो.

कधीकधी तुम्हाला वाटेल, जर प्रोग्रामला मानवाच्या गतीइतके संथपणे चालवता आले असते तर छान झाले असते, आणि डीबगर (debugger) वापरून तुम्ही ते करूही शकता. पण योग्य ठिकाणी `print` स्टेटमेंट्स टाकायला सहसा डीबगरची सेटिंग करणे, प्रोग्राममधून हळूहळू 'स्टेप' ('step') करणे, ब्रेकपॉइंट लावणे आणि काढणे ह्यांपेक्षा कमी वेळ लागतो.

प.१.३.१ माझा प्रोग्राम चालत नाहीये.

तुम्ही स्वतःला खालील प्रश्न विचारा:

- असे काही आहे का जे प्रोग्रामने करायला पाहिजे होते पण जे होत नाहीये? ती गोष्ट करणारा कोड-चा भाग शोधा आणि खात्री करा की तो भाग पाहिजे तेव्हा एक्सेक्युट होतोय.
- असे काही होत आहे का जे व्हायला नाही पाहिजे? ती गोष्ट करणारा कोड-चा भाग शोधा आणि तपासा की तो भाग नको तेव्हा एक्सेक्युट होतोय का.
- कोड-चा काही भाग अनपेक्षित काही करतोय का? तो भाग तुम्हाला नीट समजला आहे ह्याची खात्री करा, विशेषतः त्यात इतर पायथॉन मोड्युलमधील फंक्शन्स किंवा मेथड्स वापरल्या जात असतील तर. तुम्ही कॉल करत असलेल्या फंक्शन्सचे डॉक्युमेंटेशन (documentation) नीट वाचा. लहान टेस्ट केसेस (test cases) लिहून ती फंक्शन्स तपासून बघा की तुमच्या अपेक्षेप्रमाणे ती चालत आहेत की नाही.

प्रोग्रामिंग करण्यासाठी डोक्यात प्रोग्राम्स कसे चालतात ह्याची मांडणी स्पष्ट हवी. जर तुम्ही असा प्रोग्राम लिहिला जो तुम्हाला पाहिजे ते करत नाहीये, तर सहसा गडबड त्या प्रोग्राममध्ये नसून तुमच्या डोक्यातल्या मांडणीमध्ये असते.

तुमच्या डोक्यातली मांडणी दुरूस्त करण्याचा सर्वोत्तम मार्ग म्हणजे प्रोग्रामचे विविध भागांत (सहसा फंक्शन्स आणि मेथड्स ह्यांमध्ये) विभाजन करून प्रत्येक भाग स्वतंत्रपणे तपासणे. एकदा तुम्हाला तुमची मांडणी आणि सत्यपरिस्थिती ह्यांमधली विसंगती दिसली की तुम्ही बग सोडवू शकता.

साहजिक आहे की प्रोग्राम तयार करत असतानाच तुम्ही ह्या भागांची हळूहळू बांधणी आणि टेस्टिंग केली पाहिजे. जर काही गडबड झालीच तर आपल्याला हे माहित आहे की बरोबर नसण्याची शक्यता असलेला कोड एकदम कमी आहे (जो म्हणजे नुकताच लिहिलेला).

प.१.३.२ माझ्याकडे एक मोठे आणि गुंतागुंतीचे एक्स्प्रेशन आहे आणि ते माझ्या अपेक्षेप्रमाणे चालत नाही.

गुंतागुंतीचे एक्स्प्रेशन लिहिणे काही वाईट नाही, जर ते समजण्यासारखे असतील तर, पण ते डीबग करायला अवघड जाऊ शकतात. सहसा, असल्या एक्स्प्रेशनचे अनेक टेंपररी (temporary) व्हेरिएबल्सला केलेल्या असाइनमेंट्समध्ये विभाजन केलेले चांगले ठरते.

उदा.:

```
self.hands[i].addCard(self.hands[self.findNeighbor(i)].popCard())
```

हे समजायला सोपे पडेल अशा पद्धतीने खालीलप्रमाणे लिहिता येऊ शकते:

```
neighbor = self.findNeighbor(i)
pickedCard = self.hands[neighbor].popCard()
self.hands[i].addCard(pickedCard)
```

असे विभाजन केल्यावर ते समजायला सोपे जाते कारण व्हेरिएबल्सची नावे अधिक माहिती पुरवतात. आणि ते डीबग करायला ही सोपे जाते कारण तुम्ही अंतरिम व्हेरिएबल्सचे टाइप्स आणि व्हॅल्यूझ प्रिंट करू शकता.

मोठ्या एक्स्प्रेशनमुळे होऊ शकणारी अजून एक गडबड म्हणजे ते इव्हॅल्यूएट (evaluate) करण्याचा क्रम तुमच्या अपेक्षेहून वेगळा असू शकतो. उदा., जर तुम्ही $\frac{x}{2\pi}$ पायथॉनमध्ये लिहित असाल तर तुम्ही कदाचित असे लिहाल:

```
y = x / 2 * math.pi
```

पण हे चूक आहे कारण गुणाकार आणि भागाकार ह्यांना सारखेच प्राधान्य असून डावीकडून उजवीकडे इव्हॅल्यूएट केले जातात. म्हणजेच हे एक्स्प्रेशन $\frac{x\pi}{2}$ दर्शवते.

असली एक्स्प्रेशन्स डीबग करण्याचा चांगला मार्ग म्हणजे कंस टाकून इव्हॅल्यूएशनचा क्रम स्पष्ट करणे:

```
y = x / (2 * math.pi)
```

जेव्हाही तुम्हाला इव्हॅल्यूएशनच्या क्रमाबद्दल शंका असेल तेव्हा तुम्ही कंस वापरा. ह्याने ते एक्स्प्रेशन तुम्हाला पाहिजे ते करेलच आणि तुमचा कोड वाचणाऱ्या इतर लोकांना ज्यांना ऑपरेटर्सचा अनुक्रम (order of operators) पाठ नाहीये त्यांना समजायला सोपा जाईल.

प.१.३.३ माझ्याकडे एक फंक्शन आहे जे मला जे अपेक्षित आहे ते रिटर्न करत नाही.

जर तुमच्या `return` स्टेटमेंटमध्ये एक गुंतागुंतीचे एक्स्प्रेशन असेल तर तुम्हाला उत्तर रिटर्न करायच्या आधी ते प्रिंट करण्याची संधी मिळणार नाही. इथेही तुम्ही टेंपररी व्हेरिएबल्स वापरू शकता. उदा., खालील एक्स्प्रेशनऐवजी:

```
return self.hands[i].removeMatches()
```

तुम्ही हे लिहू शकता:

```
count = self.hands[i].removeMatches()
return count
```

आता तुम्हाला रिटर्न करायच्या आधी `count` ची व्हॅल्यू दाखवायची संधी आहे.

प.१.३.४ मी खरोखर अडकलेय, आणि मला मदतीची गरज आहे.

सर्वप्रथम, कॉंप्युटरपासून काही वेळ दूर व्हा. कॉंप्युटरमधून अशा लहरींचे उत्सर्जन होते ज्याने खालील लक्षणे दिसून येतात:

- निराशा आणि संताप.
- अंधश्रद्धा ('कॉंप्युटर माझा द्वेष करतो') आणि जादुई शक्तींवरचा विश्वास ('मी उजव्या हाताच्या बोटाने बटण दाबल्यावरच प्रोग्राम चालतो').

- रँडम वॉक प्रोग्रामिंग (random walk programming, म्हणजेच शक्य असतील तितके प्रोग्राम्स लिहून त्यांपैकी चालणारा निवडणे).

जर तुम्ही ह्यांपैकी कोणत्याही लक्षणाने ग्रस्त असाल तर उठा आणि बाहेर एक चक्कर मारून या. डोके थोडे शांत झाले की मगच प्रोग्रामचा विचार करा. तो काय करतोय? त्याच्या अशा वागण्याची कोणती कारणे असू शकतात? मागच्या वेळी तुमच्याकडे कधी चालणारा प्रोग्राम होता आणि त्यानंतर तुम्ही काय केले?

कधीकधी बग शोधायला वेळ लागतोच. अनेकदा कॉंप्युटरपासून दूर, मन भटकत असताना बग मिळतो. बस, ट्रेन, अंघोळ करताना, झोप लागायच्या आधी, ह्या बग्स शोधण्याच्या सर्वोत्तम जागा/वेळा आहेत.

प.१.३.५ नाही, मला खरेच मदत हवी आहे.

असे होते. अधूनमधून सर्वोत्तम प्रोग्रामर्सदेखील अडकतात. कधीकधी एका प्रोग्रामवर तुम्ही इतके काम करता की काही केल्या तुम्हाला त्यातील बग दिसतच नाही. दुसऱ्या कोणी तुमचा प्रोग्राम तपासायची गरज पडते.

दुसऱ्या कोणाला आणण्याआधी तुम्ही तयार आहात ह्याची खात्री करा. तुमचा प्रोग्राम शक्य तितका लहान असला पाहिजे, आणि तुमच्याकडे एरर देणारे सर्वांत लहान इनपुट असले पाहिजे. योग्य ठिकाणी `print` स्टेटमेंट्स असली पाहिजेत (आणि त्यांचे आउटपुट समजण्यासारखे पाहिजे). तुम्हाला चूक स्पष्टपणे, व्यवस्थितपणे, आणि संक्षेपाने समजावून सांगता येण्याइतकी समजली पाहिजे.

दुसऱ्या कोणाची मदत घेताना त्यांना जी माहिती लागेल ती देण्यास विसरू नका:

- जर एरर मेसेज असला तर तो काय आहे आणि प्रोग्रामचा कोणता भाग तो दर्शवतो?
- हा एरर येण्याच्या अगदी आधी तुम्ही काय केले? तेव्हा तुम्ही कोड-च्या कोणत्या ओळी लिहिल्या होत्या, किंवा कोणती नवीन टेस्ट केस (test case) पास होत नाहीये?
- आतापर्यंत तुम्ही काय करायचा प्रयत्न केलाय आणि तुम्ही काय शिकलात?

अखेरीस जेव्हा तुम्हाला बग मिळतो तेव्हा एक क्षण तुम्ही हा विचार करा की काय केल्याने तुम्हाला तो अजून लवकर मिळाला असता. पुढच्या वेळेला असे काही झाल्यावर तुम्हाला तो बग लवकर मिळेल.

हे विसरू नका की प्रोग्राम व्यवस्थित चालवणे हाच उद्देश नाहीये. प्रोग्राम व्यवस्थित चालवायला शिकणे हा उद्देश आहे.

परिशिष्ट २

अल्गोरिदम विश्लेषण (Analysis of Algorithms)

हे परिशिष्ट ॲलन डाउनी ह्यांच्या *Think Complexity* पुस्तकावर आधारित आहे, जे O'Reilly Media (2012) ने प्रकाशित केले आहे. कुतूहल असल्यास तुम्ही हे पुस्तक संपल्यावर ते वाचू शकता.

अल्गोरिदम विश्लेषण (**analysis of algorithms**) ही संगणक विज्ञानाची एक शाखा असून त्यात अल्गोरिदम्सची कार्यक्षमता, विशेषत: त्यांना लागणारा वेळ (running time, रनिंग टाइम) आणि मेमरीची जागा (memory/space usage) ह्यांचा अभ्यास केला जातो. पुढील लिंक बघा: http://en.wikipedia.org/wiki/Analysis_of_algorithms.

अल्गोरिदम्सच्या कार्यक्षमतेचे विश्लेषण करण्याचा मुख्य उद्देश म्हणजे डिझाइनविषयक निर्णय घेण्यासाठी मार्गदर्शन करणे.

२००८ सालच्या युनायटेड स्टेट्स अध्यक्षीय मोहिमेदरम्यान (United States Presidential Campaign) उमेदवार बराक ओबामा ह्यांनी गूगल (Google) ला भेट दिली तेव्हा त्यांना एक अल्गोरिदम विश्लेषण संबंधित प्रश्न विचारला गेला. मुख्य कार्यकारी एरिक श्मिट ह्यांनी गंमतीने त्यांना 'दहा लाख 32-बिट (bit) इंटिजर्स सॉर्ट करण्याचा सर्वांत प्रभावी मार्ग कोणता' हे विचारले. ओबामांना असे काही विचारतील ह्याची चाहूल लागली होती, कारण त्यांनी पटकन उत्तर दिले, 'मला वाटते की बबल सॉर्ट (bubble sort) हा नक्कीच चुकीचा मार्ग आहे.' हे बघा: http://www.youtube.com/watch?v=k4RRi_ntQc8.

हे बरोबर आहे: बबल सॉर्ट समजायला सोपा असला तरी मोठ्या डेटासेट्ससाठी अतिशय संथ ठरेल. श्मिटना बहुधा 'रॅडिक्स सॉर्ट' ('radix sort') हे उत्तर अपेक्षित असावे [१] (http://en.wikipedia.org/wiki/Radix_sort).

अल्गोरिदम-विश्लेषण करण्याचे उद्दिष्ट म्हणजे विविध अल्गोरिदम्सची अर्थपूर्ण तुलना करणे, पण ह्यात काही अडचणी आहेत:

- अल्गोरिदम्सची तुलनात्मक कार्यक्षमता हार्डवेअरच्या (hardware) गुणधर्मांवर अवलंबून असू शकते, तर एक अल्गोरिदम मशीन A वर दुसऱ्यापेक्षा जास्ती वेगवान असू शकतो आणि दुसरा मशीन B वर पहिल्यापेक्षा. ही अडचण सोडवण्याचा एक साधारण पर्याय म्हणजे एक विशिष्ट **मशीन मॉडेल** (**machine model**) ठरवून दिलेला अल्गोरिदम त्यात किती ऑपरेशन्स करतो ह्याचे विश्लेषण करणे.

[१] पण जर तुम्हाला हा प्रश्न जॉब इंटरव्यूमध्ये (job interview) विचारला तर योग्य उत्तर पुढीलप्रमाणे ठरेल, 'दहा लाख इंटिजर्स सॉर्ट करण्याचा सर्वांत वेगवान मार्ग म्हणजे मी वापरत असलेल्या प्रोग्रामिंग लँग्वेजने पुरवलेले सॉर्ट फंक्शन वापरणे. त्याची कार्यक्षमता जवळजवळ सर्व कामांसाठी चांगली आहे. पण अनायासे माझ्यासाठी जर ते संथ ठरले तर मी एक प्रोफाइलर (profiler) वापरून प्रोग्रामचा वेळखाऊ भाग शोधेन. जर असे आढळून आले की अजून वेगवान अल्गोरिदम वापरल्याने प्रोग्रामच्या कार्यक्षमतेत खूप सुधार होऊ शकतो, तर मी रॅडिक्स सॉर्टचे चांगले इंप्लेमेंटेशन शोधेन.'

- अल्गोरिदम्सची तुलनात्मक कार्यक्षमता डेटासेटच्या तपशीलावर अवलंबून असू शकते. उदा., जर डेटा अंशतः सॉर्टेड (sorted) असेल तर काही सॉर्टिंग अल्गोरिदम्स जास्ती वेगाने चालतात तर इतर कमी वेगाने. ही अडचण सोडवण्याचा एक साधारण पर्याय म्हणजे **सर्वांत वाईट केस** (**worst case**, वर्स्ट केस) चे विश्लेषण करणे. कधीकधी सरासरी कार्यक्षमतेचे विश्लेषण करणे उपयोगी ठरते, पण ते विश्लेषण करणे सहसा अवघड असते, आणि हेही स्पष्ट नसते की कोणत्या केसेसच्या समूहावर सरासरी घेतली पाहिजे.
- अल्गोरिदम्सची तुलनात्मक कार्यक्षमता डेटाच्या/इनपुटच्या साइझ (size) वरही अवलंबून असतो. लहान लिस्ट्ससाठी वेगाने चालणारा अल्गोरिदम मोठ्या लिस्ट्सवर अतिसंथ असू शकतो. ही अडचण सोडवण्याचा एक साधारण पर्याय म्हणजे रनिंग टाइम (किंवा ऑपरेशन्सची संख्या) इनपुट साइझच्या फलनस्वरुपात व्यक्त करणे, आणि त्या फलनांचे ते इनपुट साइझच्या प्रमाणात किती वेगाने वाढतात ह्यानुसार वर्गीकरण करणे. (अनुवादकाची टिप्पणी: ह्याठिकाणी फलन म्हणजे गणितातील फंक्शन, ज्याला आपण इथून पुढे फक्त फंक्शन असेच म्हणणार आहोत. हे लक्षात असू द्या की पायथॉनमधील फंक्शन आणि गणितातील फंक्शन ह्या वेगळ्या गोष्टी आहेत, जरी पायथॉनमधील फंक्शन हे गणितातील फंक्शनच्या संकल्पनेवर आधारित असले तरी. फंक्शनचे एक उदाहरण म्हणजे $f(x) = x^2$. पुढील लिंक बघा: https://marathivishwakosh.org/21979/)

ह्याप्रकारच्या तुलनेचा फायदा हा की अल्गोरिदम्सचे सरळ वर्गीकरण करता येते. उदा., जर अल्गोरिदम A चा रनिंग टाइम इनपुट साइझ n शी समप्रमाणात असेल आणि अल्गोरिदम B चा रनिंग टाइम n^2 शी समप्रमाणात असेल तर आपण ही अपेक्षा करू शकतो की A हा B पेक्षा वेगवान असेल, n च्या मोठ्या किंमतींसाठी तरी.

ह्याप्रकारच्या विश्लेषणाच्याही काही मर्यादा आहेत, पण त्या आपण नंतर पाहूया.

प.२.१ ऑर्डर-ऑफ-ग्रोथ (Order of growth)

समजा तुम्ही दोन अल्गोरिदम्सचे विश्लेषण करून त्यांच रनिंग टाइम्स इनपुट साइझच्या रुपात व्यक्त केले: n साइझच्या इनपुटवर अल्गोरिदम A ला $100n + 1$ ऑपरेशन्स लागतात आणि अल्गोरिदम B ला $n^2 + n + 1$ ऑपरेशन्स लागतात.

खालील तक्ता वेगवेगळ्या इनपुट साइझेसवर ह्या अल्गोरिदम्सचे रनिंग टाइम्स दाखवतो:

इनपुट साइझ	अल्गोरिदम A चा रनिंग टाइम	अल्गोरिदम B चा रनिंग टाइम
10	1 001	111
100	10 001	10 101
1 000	100 001	1 001 001
10 000	1 000 001	100 010 001

$n = 10$ असताना अल्गोरिदम A जवळजवळ अल्गोरिदम B च्या दहापट वेळ घेतो. पण $n = 100$ असताना दोन्ही जवळजवळ सारखेच आहेत, आणि मोठ्या किंमतींसाठी A खूप चांगला आहे.

ह्याचे मूलभूत कारण म्हणजे n च्या मोठ्या किंमतींसाठी n^2 term असणारे कोणतेही फंक्शन n ही leading-term असणाऱ्या फंक्शनपेक्षा वेगाने वाढते; **leading term** म्हणजे सर्वांत जास्ती घात (exponent) असलेली term.

अल्गोरिदम A साठी leading-term चा सहगुणक (coefficient) 100 म्हणजे मोठा आहे, म्हणून n च्या कमी किंमतींसाठी अल्गोरिदम B हा अल्गोरिदम A पेक्षा चांगला आहे. पण coeffcient काहीही असले तरी n ची अशी कोणतीतरी किंमत असेल ज्यानंतर $an^2 > bn$ असेल (a आणि b ह्यांची किंमत काहीही असली तरी).

हाच युक्तिवाद non-leading-terms ना सुद्धा लागू होतो. अल्गोरिदम A चा रनिंग टाइम $n + 1000000$ जरी असला असता तरीसुद्धा तो n च्या पुरेशा मोठ्या किंमतींसाठी अल्गोरिदम B पेक्षा चांगला ठरला असता.

साधारणपणे, आपली अशी अपेक्षा असते की लहान leading-term असलेला अल्गोरिदम मोठ्या इनपुटसाठी जास्ती चांगला असेल, पण लहान इनपुट साठी एक **crossover point** असू शकतो जिथे दुसरा अल्गोरिदम जास्ती चांगला

ठरेल. ह्या crossover point चे स्थान अल्गोरिदम्सचे तपशील, इनपुट्स, आणि हार्डवेअर ह्यांवर अवलंबून असते, तर अल्गोरिदम-विश्लेषण करताना ह्याचा विचार नाही केला जात. पण त्याचा अर्थ हा नाही की तुम्ही त्याबद्दल विसरून जाऊ शकता.

जर दोन अल्गोरिदम्सची leading-term सारखीच असेल तर कोणता जास्ती चांगला आहे हे सांगणे कठीण असते; उत्तर इतर तपशीलांवर अवलंबून असते. म्हणून अल्गोरिदम-विश्लेषण करताना सारख्याच leading-terms असणारी फंक्शन्स समरूप मानली जातात, जरी त्यांचे coefficients वेगळे असले तरी.

Order-of-growth म्हणजे अशा फंक्शन्सचा संच ज्यांना वाढीच्या (growth) दृष्टीने समरूप मानले जाते. उदा., $2n$, $100n$, आणि $n+1$ हे एकाच order-of-growth मध्ये आहेत, जिला $O(n)$ असे **Big-Oh notation** मध्ये लिहिले जाते, आणि त्यांना **linear** म्हटले जाते कारण त्यातील प्रत्येक फंक्शन n च्या समप्रमाणात वाढते आणि तशा फंक्शन्सचा आलेख हा एक रेषा असतो.

Leading-term n^2 असणारी सर्व फंक्शन्स $O(n^2)$ मध्ये असतात; त्यांना **quadratic** म्हटले जाते.

खालील तक्ता अल्गोरिदम-विश्लेषण करताना दिसणाऱ्या सर्वांत कॉमन orders-of-growth वाईटपणाच्या चढत्या क्रमाने दाखवतो.

Order of growth	नाव
$O(1)$	constant
$O(\log_b n)$	logarithmic (कोणत्याही b साठी)
$O(n)$	linear
$O(n \log_b n)$	linearithmic
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(2^n)$	exponential
$O(3^n)$	exponential
$\vdots$	$\vdots$

Logarithmic terms साठी logarithm चा पाया किती आहे ह्याचा फरक नाही पडत; पाया बदलणे म्हणजे constant ने गुणण्यासारखेच आहे, ज्याने order-of-growth बदलत नाही. Exponential फंक्शन्स झपाट्याने वाढतात, म्हणून exponential अल्गोरिदम्स कमी कामांसाठी उपयोगी असतात.

प्रश्न २.१. *पुढील लिंकवर Big-Oh नोटेशन बद्दलचे विकिपीडिया पेज वाचा आणि खालील प्रश्नांची उत्तरे द्या* http://en.wikipedia.org/wiki/Big_O_notation:

http://en.wikipedia.org/wiki/Big_O_notation *and answer the following questions:*

१. $n^3 + n^2$ *ची order-of-growth काय आहे?* $1000000n^3 + n^2$ *ची?* $n^3 + 1000000n^2$ *ची?*

२. $(n^2 + n) \cdot (n + 1)$ *ची order-of-growth काय आहे? पूर्ण गुणाकार करण्याआधी हे ध्यानात घ्या की तुम्हाला फक्त leading-term लागेल.*

३. *जर f हे फंक्शन $O(g)$ मध्ये असेल; इथे g हे एक (कोणतेतरी) फंक्शन आहे. जर a आणि b हे constants असतील, तर $af + b$ ह्या फंक्शनच्या order-of-growth बद्दल आपण काय सांगू शकतो?*

४. *जर f_1 आणि f_2 हे $O(g)$ मध्ये असतील तर $f_1 + f_2$ ह्या फंक्शनच्या order-of-growth बद्दल आपण काय सांगू शकतो?*

५. *जर f_1 हे $O(g)$ मध्ये असेल आणि f_2 हे $O(h)$ मध्ये असेल तर $f_1 + f_2$ ह्या फंक्शनच्या order-of-growth बद्दल आपण काय सांगू शकतो?*

६. *जर f_1 हे $O(g)$ मध्ये असेल आणि f_2 हे $O(h)$ मध्ये असेल तर $f_1 \cdot f_2$ ह्या फंक्शनच्या order-of-growth बद्दल आपण काय सांगू शकतो?*

ज्या प्रोग्रामर्सना कार्यक्षमतेची पर्वा आहे त्यांना ह्याप्रकारचे विश्लेषण पचायला अवघड जाते. त्यात तथ्यदेखील आहे: कधीकधी coefficients आणि non-leading-terms मुळे मोठा फरक पडतो. कधीकधी हार्डवेअर, प्रोग्रामिंग लँग्वेज, आणि इनपुटची वैशिष्ट्ये ह्यांनीही खूप फरक पडतो. आणि लहान इनपुटसाठी order-of-growth ने (बहुतांश वेळा) काहीच फरक पडत नाही.

पण हे मुद्दे ध्यानात ठेवले तर अल्गोरिदम-विश्लेषण हे उपयोगी तंत्र आहे. निदान मोठ्या इनपुटसाठी 'चांगला' अल्गोरिदम सहसा चांगला असतो, आणि कधीकधी *खूप* चांगला. सारख्याच order-of-growth च्या दोन अल्गोरिदम्समधील फरक सहसा एका constant factor चा असतो, पण एक चांगला अल्गोरिदम आणि एक वाईट अल्गोरिदम ह्यांमधील फरक अमर्याद असतो!

प.२.२ पायथॉनच्या मूलभूत ऑपरेशन्सचे विश्लेषण

पायथॉनमध्ये बहुतांश अंकगणितीय ऑपरेशन्स constant time आहेत; गुणाकाराला बेरीज आणि वजाबाकीपेक्षा जास्ती वेळ लागतो, आणि भागाकाराला त्याहूनही जास्ती, पण हे रनिंग टाइम्स ऑपरँड्सच्या व्हॅल्यूझ-वर अवलंबून नाहीत. खूप मोठे इंटिजर्स अपवाद आहेत; त्यांच्यासाठी रनिंग टाइम त्यांमधील अंकांच्या संख्येवर अवलंबून आहेत.

इंडेक्स ऑपरेशन्स—सीक्वेन्समधून एलेमेंट वाचणे किंवा त्यात लिहिणे—सुद्धा constant time आहे, डेटा स्ट्रक्चरची साइझ कितीही असली तरी.

सीक्वेन्स किंवा डिक्शनरी ट्रव्हर्स करणारा `for` लूप linear असतो, जर लूप-च्या बॉडीतील सर्व ऑपरेशन्स constant time असतील तर. उदा., लिस्टमधील एलेमेंट्सची बेरीज करणे linear आहे:

```
total = 0
for x in t:
    total += x
```

`sum` हे बिल्ट-इन फंक्शनसुद्धा linear आहे कारण ते हीच गोष्ट करते, पण ते जास्ती वेगाने चालते कारण त्याचे इंप्लेमेंटेशन जास्ती कार्यक्षम आहे; अल्गोरिदम-विश्लेषणमधील भाषेत त्याचा leading-coefficient लहान आहे.

एक ढोबळ मार्गदर्शक सूचना म्हणजे जर लूप-ची बॉडी $O(n^a)$ असेल तर पूर्ण लूप $O(n^{a+1})$ असतो. अपवाद म्हणजे तुम्ही जर हे सिद्ध करू शकलात की constant इटरेशन्स नंतर लूप संपतो. जर c एक constant असताना लूप c वेळा रन होत असेल, n कितीही असला तरी, तर लूप $O(n^a)$ असतो (जरी constant c कितीही मोठा असला तरी).

Constant c ने गुणल्यावर order-of-growth बदलत नाही आणि भागल्यावरही नाही. जर लूप-ची बॉडी $O(n^a)$ असेल आणि तो n/c वेळा रन होत असेल, तर लूप $O(n^{a+1})$ असतो (जरी constant c कितीही मोठा असला तरी).

बहुतांश स्ट्रिंग आणि टपल ऑपरेशन्स हे linear असतात, इंडेक्सिंग आणि `len` ह्यांचा अपवाद वगळून जे constant time आहेत. `min` आणि `max` हे बिल्ट-इन फंक्शन्स linear आहेत. स्लाइस (slice) ऑपरेशन चा रनिंग टाइम त्याच्या आउटपुटच्या लांबीच्या प्रमाणात असतो पण इनपुटच्या लांबीवर अवलंबून नसतो.

स्ट्रिंग कन्कॅटनेशन (concatenation) linear असते; रनिंग टाइम ऑपरँड्सच्या लांबींच्या बेरजेवर अवलंबून असतो.

सर्व स्ट्रिंग मेथड्स linear आहेत पण जर सर्व स्ट्रिंग्स constant लांबीच्या असतील—उदा., एका कॅरेक्टरवरील ऑपरेशन्स—तर त्यांना constant time मानले जाते. `join` ही स्ट्रिंग मेथड linear आहे; तिचा रनिंग टाइम सर्व स्ट्रिंग्सच्या एकूण लांबीवर अवलंबून असतो.

बहुतांश लिस्ट मेथड्स linear आहेत, पण काही अपवाद असे:

- लिस्टच्या शेवटी एलेमेंट जोडायला सरासरी constant time लागतो; जागा संपली की लिस्ट मोठ्या जागेत हलवली जाते, पण n ऑपरेशन्ससाठी $O(n)$ वेळ लागतो, म्हणून प्रत्येक ऑपरेशनचा सरासरी वेळ $O(1)$ (म्हणजेच constant) आहे.
- लिस्टच्या शेवटून एलेमेंट हटवणे constant time आहे.

- सॉर्टिंग $O(n \log n)$ आहे.

बहुतांश डिक्शनरी ऑपरेशन्स आणि मेथड्स linear आहेत, पण काही अपवाद असे:

- `update` चा रनिंग टाइम जी डिक्शनरी अर्ग्युमेंट म्हणून पाठवली आहे तिच्या साइझ च्या प्रमाणात असतो, जी डिक्शनरी अपडेट होत आहे तिच्या नाही.
- `keys`, `values`, आणि `items` constant time आहेत कारण ते इटरेटर (iterator) रिटर्न करतात. पण जर तुम्ही त्या इटरेटरमधून लूप केले तर तो linear time असेल.

डिक्शनरीची कार्यक्षमता ही संगणक विज्ञानाची एक छोटी जादूच आहे. ती कशी चालते हे आपण विभाग प.२.४ मध्ये बघणार आहोत.

प्रश्न २.२. *सॉर्टिंग अल्गोरिदम्सवरचे विकिपीडिया पेज पुढील लिंकवर वाचून खालील प्रश्नांची उत्तरे द्या: http://en.wikipedia.org/wiki/Sorting_algorithm:*

१. *'Comparison sort' म्हणजे काय? Comparison sort ची सर्वोत्तम worst-case order-of-growth काय आहे? कोणत्याही सॉर्टिंग अल्गोरिदमची सर्वोत्तम worst-case order-of-growth काय आहे?*

२. *बबल सॉर्ट (bubble sort) ची order-of-growth काय आहे, आणि बराक ओबामांना असे का वाटते की 'बबल सॉर्ट (bubble sort) हा नक्कीच चुकीचा मार्ग आहे'?*

३. *Radix sort ची order-of-growth काय आहे? तो वापरण्यासाठी आपल्याला काय प्रीकंडिशन्स लागतात?*

४. *Stable sort म्हणजे काय आणि त्याचे व्यवहारोपयोगीपणात काय महत्त्व आहे?*

५. *सर्वांत वाईट सॉर्टिंग अल्गोरिदम कोणता (ज्याचे नाव आहे)?*

६. *C library कोणता सॉर्टिंग अल्गोरिदम वापरते? पायथॉन कोणता सॉर्टिंग अल्गोरिदम वापरतो? हे अल्गोरिदम्स stable आहेत का? ह्यांची उत्तरे शोधण्यासाठी तुम्हाला थोडे गूगल करावे लागेल.*

७. *अनेक comparison sorts नसलेले सॉर्टिंग अल्गोरिदम्स linear आहेत, तर पायथॉन $O(n \log n)$ comparison sort का वापरतो?*

प.२.३ सर्च अल्गोरिदमचे विश्लेषण

एक collection आणि एक target item घेऊन ते target त्या collection मध्ये आहे की नाही हे शोधणाऱ्या अल्गोरिदमला **search** म्हणतात; सहसा तो target ची इंडेक्ससुद्धा रिटर्न करतो.

सर्वांत सोपा सर्च अल्गोरिदम म्हणजे 'linear search' जो दिलेल्या collection मधील items क्रमाने ट्रव्हर्स करतो आणि target मिळाल्यावर थांबतो. Worst-case मध्ये तो संपूर्ण collection ट्रव्हर्स करतो, म्हणजे रनिंग टाइम linear आहे.

सीक्वेन्सेसवर `in` ऑपरेटर linear search वापरतो; आणि `find` आणि `count` सारख्या स्ट्रिंग मेथड्ससुद्धा.

जर सीक्वेन्समधील एलेमेंट्स (चढत्या किंवा उतरत्या) क्रमाने असतील तर तुम्ही **bisection search** म्हणजेच **binary search** वापरू शकता, जो $O(\log n)$ आहे. आपण ह्याविषयी प्रश्न १०.१० मध्ये चर्चा केली होती. त्याची पुनरावृत्ती करूया. Binary search हा तुम्ही (पुस्तकरुपातील) शब्दकोशात शब्द शोधताना जो अल्गोरिदम वापरता त्यासारखाच आहे. तुम्ही मध्यभागी बघून तपासता की तुम्हाला जो शब्द पाहिजे आहे तो बघितलेल्या शब्दाच्या आधी आहे का नंतर. आधी असेल तर तुम्ही पहिल्या अर्ध्या भागात शोधता, नाहीतर दुसऱ्या. ह्या-नाहीतर-त्या-प्रकारे तुम्ही शोधक्षेत्र अर्धे करता.

जर सीक्वेन्समध्ये दहा लाख म्हणजेच 1,000,000 items असतील तर अंदाजे 20 ऑपरेशन्सनंतर आपल्याला एकतर item सापडेल किंवा असा निष्कर्ष काढता येईल की तो अनुपस्थित आहे. म्हणजेच हे linear search पेक्षा जवळजवळ 50,000 पट जलद आहे.

Binary search हा linear search पेक्षा खूप जलद जरी असला तरी तो वापरता येण्यासाठी सीक्वेन्स सॉर्टेड असला पाहिजे, ज्यासाठी जास्तीचे काम लागू शकते.

एक **hash table** नावाचे डेटा स्ट्रक्चर आहे जे अजून जलद आहे—ते constant time मध्ये सर्च करते—आणि ते वापरण्यासाठी items सॉर्टेड असण्याची गरज नाही. पायथॉन डिक्शनरी ही hash table वापरून इंप्लेमेंट केली आहे, म्हणून `in` ऑपरेटरसह बहुतांश डिक्शनरी ऑपरेशन्स constant time आहेत.

प.२.४ हॅश टेबल (Hash tables)

Hash-table कसे चालतात आणि त्यांची कार्यक्षमता इतकी भारी का आहे हे बघण्यासाठी आपण map च्या एका साध्या इंप्लेमेंटेशनने सुरुवात करून त्याचा hash-table होत नाही तोपर्यंत त्यात हळूहळू सुधारणा करू.

ह्यातील संकल्पना दर्शवण्यासाठी आपण पायथॉन वापरणार आहोत पण व्यवहारात तुम्ही असा कोड पायथॉनमध्ये असा कोड कधीच लिहिणार नाही; त्याऐवजी तुम्ही एक डिक्शनरीच वापराल! तर ह्या परिशिष्टाच्या उर्वरीत भागात तुम्ही अशी कल्पना करा की डिक्शनरी अस्तित्वात नाहीत आणि तुम्हाला असे एक डेटा स्ट्रक्चर इंप्लेमेंट करायचे आहे जे keys व्हॅल्यूझ-ना मॅप करते. तुम्हाला खालील ऑपरेशन्स इंप्लेमेंट करायची आहेत:

`add(k, v)`: `k` ही key `v` ह्या व्हॅल्यूला मॅप करणारा एक नवीन item डेटा-स्ट्रक्चरमध्ये टाका. पायथॉन डिक्शनरी `d` वापरून हे ऑपरेशन `d[k] = v` असे लिहिता येईल.

`get(k)`: `get(k)` शी संबंधित व्हॅल्यू शोधून रिटर्न करा. पायथॉन डिक्शनरी `d` वापरून हे ऑपरेशन `d` किंवा `d[k]` असे लिहिता येईल.

आतापुरते आपण असे गृहीत धरूया की प्रत्येक key फक्त एकदाच येऊ शकते. ह्या इंटरफेसचे सर्वांत सोपे इंप्लेमेंटेशन टपल्सची लिस्ट वापरते ज्यात प्रत्येक टपल हे एक key-व्हॅल्यू जोडी असते.

```
class LinearMap:

    def __init__(self):
        self.items = []

    def add(self, k, v):
        self.items.append((k, v))

    def get(self, k):
        for key, val in self.items:
            if key == k:
                return val
        raise KeyError
```

`add` मेथड key-व्हॅल्यू टपल items च्या लिस्टमध्ये जोडते ज्याला constant time लागतो.

`get` मेथड `for` लूप वापरून लिस्ट सर्च करते: जर तिला target key मिळाली तर ती संबंधित व्हॅल्यू रिटर्न करते नाहीतर ती `KeyError` raise करते. म्हणजेच `get` linear आहे.

एक पर्याय असा आहे की लिस्टला key च्या क्रमाने सॉर्टेड ठेवणे. म्हणजे `get` साठी आपल्याला binary search वापरता येईल जो $O(\log n)$ आहे. पण लिस्टच्यामध्ये नवीन item टाकणे linear आहे, तर हा पर्याय चांगला नाही.

`LinearMap` सुधारण्याचा अजून एक पर्याय म्हणजे key-व्हॅल्यू जोड्यांच्या लिस्टचे लहान-लहान लिस्ट्समध्ये विभाजन करणे. खाली एक `BetterMap` नावाचे इंप्लेमेंटेशन आहे ज्यात 100 LinearMaps ची लिस्ट आहे. आपण बघूच की `get` ची order-of-growth ह्यातही linear आहे पण तरीही `BetterMap` हे hash-table च्या दिशेने जाणारे एक पाऊल आहे:

```
class BetterMap:

    def __init__(self, n=100):
        self.maps = []
        for i in range(n):
            self.maps.append(LinearMap())

    def find_map(self, k):
        index = hash(k) % len(self.maps)
        return self.maps[index]

    def add(self, k, v):
        m = self.find_map(k)
        m.add(k, v)

    def get(self, k):
        m = self.find_map(k)
        return m.get(k)
```

`__init__` मेथड n `LinearMaps` ची लिस्ट बनवते.

कोणत्या मॅपमध्ये नवीन item टाकायचा किंवा शोधायचा हे बघण्यासाठी `add` आणि `get` ह्या मेथड्स `find_map` वापरतात.

`find_map` मेथड `hash` बिल्ट-इन फंक्शनला वापरते, जे जवळजवळ कोणताही पायथॉन ऑब्जेक्ट घेऊन एक इंटिजर रिटर्न करते. ह्या इंप्लेमेंटेशनची एक उणीव म्हणजे ते फक्त हॅशेबल keys वरच चालते. लिस्ट आणि डिक्शनरी सारखे म्युटबल टाइप्स हॅशेबल नाहीत.

सारखे समजले जाणारे ऑब्जेक्ट्स एकच हॅश व्हॅल्यू रिटर्न करतात, पण ह्याउलट विधान सत्य असेलच असे नाही: दोन वेगळ्या व्हॅल्यूझ असलेले ऑब्जेक्ट्स एकच हॅश व्हॅल्यू रिटर्न करू शकतात.

`find_map` मॉड्युलस ऑपरेटरचा वापर करून हॅश व्हॅल्यूचे 0 ते `len(self.maps)` मधील इंटिजरमध्ये रुपांतर करतो, जेणेकरून उत्तर हे लिस्टमधील वैध इंडेक्स असेल. म्हणजेच अनेक हॅश व्हॅल्यूझचे एकाच इंडेक्समध्ये रुपांतर होऊ शकते. पण जर हॅश फंक्शनने किंमती सारख्याच प्रमाणात पसरल्या (हे करण्यासाठीच हॅश फंक्शन डिझाइन केले जाते), तर आपण प्रत्येक LinearMap मध्ये $n/100$ items असतील अशी अपेक्षा करू शकतो.

`LinearMap.get` चा रनिंग टाइम items च्या संख्येच्या प्रमाणात असल्याने BetterMap LinearMap पेक्षा 100 पट वेगवान असेल अशी आपण अपेक्षा करू शकतो. Order-of-growth अजूनही linear च आहे, पण leading-coefficient लहान आहे. छान, पण अजूनही hash-table इतके चांगले नाही.

ही शेवटची आणि महत्त्वाची आयडिया जी hash-table ला जलद बनवते: जर तुम्ही प्रत्येकं LinearMap ची लांबी constant ठेवली तर `LinearMap.get` constant time असेल. तुम्हाला फक्त किती items आहेत ह्याची नोंद ठेवायची आहे आणि जेव्हा items ची संख्या प्रति LinearMap (म्हणजेच, सरासरी) एका विशिष्ठ संख्येहून जास्ती होते तेव्हा तुम्ही अजून LinearMaps टाकून hash-table ची साइझ वाढवू शकता.

खाली hash-table चे इंप्लेमेंटेशन आहे:

```
class HashMap:

    def __init__(self):
        self.maps = BetterMap(2)
        self.num = 0

    def get(self, k):
```

```
        return self.maps.get(k)

    def add(self, k, v):
        if self.num == len(self.maps.maps):
            self.resize()

        self.maps.add(k, v)
        self.num += 1

    def resize(self):
        new_maps = BetterMap(self.num * 2)

        for m in self.maps.maps:
            for k, v in m.items:
                new_maps.add(k, v)

        self.maps = new_maps
```

`__init__` एक `BetterMap` बनवून items च्या संख्येची नोंद ठेवणारे `num` इनिशलाइझ करते.

`get` तर कॉल `BetterMap` ला पाठवते. खरे काम `add` मध्ये होते; जी items ची संख्या आणि `BetterMap` ची साइझ तपासते: जर त्या सारख्या असतील तर items ची सरासरी संख्या प्रति LinearMap 1 आहे, तर ती `resize` इन्व्होक करते.

`resize` एक नवीन `BetterMap` बनवते जो आधीपेक्षा दुप्पट मोठा असतो, आणि सर्व items नवीन मॅपमध्ये 'rehash' करते.

Rehashing करणे गरजेचे आहे कारण LinearMaps ची संख्या बदलल्यामुळे `find_map` मधील मॉड्युलस ऑपरेटरचा 'छेद' (उजवा ऑपरँड) बदलतो. म्हणजेच काही ऑब्जेक्ट्स जे एकाच LinearMap मध्ये आधी हॅश होत होते ते आता वेगळे होतील (आणि आपल्याला हेच पाहिजे होते, बरोबर?).

Rehashing हे linear आहे, म्हणजे `resize` linear आहे, आणि हे दिसायला वाईट आहे कारण आपण असे ठरवले होते की `add` ला constant time बनवायचे. पण हे लक्षात असू द्या की आपण दरवेळी resize नाही करत, म्हणजेच `add` बहुतांश वेळा constant time आहे आणि क्वचित linear. `add` ला n वेळा रन करण्यासाठी लागणारा एकूण वेळ हा n शी समप्रमाणात आहे, म्हणजेच प्रत्येक `add` चा सरासरी वेळ constant च आहे!

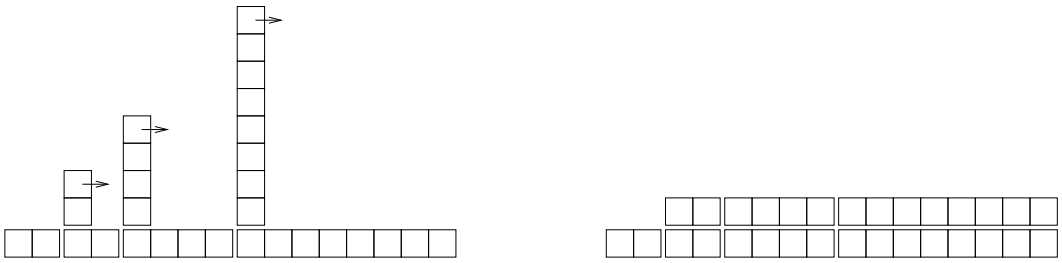
हे कसे ते बघण्यासाठी एका रिकाम्या HashTable ने सुरुवात करून items चा एक सीक्वेन्स टाकल्यावर काय होईल ह्याचा विचार करा. आपण 2 LinearMaps ने सुरुवात करतो, तर पहिले 2 adds जलदगतीने होतात (resizing ची गरज नाही). असे समजा त्यांना एक एकक वेळ लागतो. पुढील add ला resize लागते, तर आपण पहिले दोन items rehash करतो (म्हणजे अजून 2 एकक वेळ) आणि नंतर तिसरा item टाकतो (ज्यासाठी अजून एक एकक वेळ). पुढील item ला एक एकक वेळ, तर आतापर्यंत 4 items साठी 6 एकक वेळ.

पुढील `add` ची किंमत 5 एकक आहे, पण त्यानंतरच्या तीनची फक्त प्रत्येकी एक एकक, म्हणजे पहिल्या 8 `add` साठी एकूण 14 एकक.

पुढील `add` ची किंमत 9 एकक आहे, पण नंतरच्या resize च्या आधी आपण अजून 7 add करू, म्हणजे एकूण किंमत 30 एकक पहिल्या 16 adds साठी.

32 adds नंतर किंमत आहे 62 एकक, आणि आता तुम्हाला ह्यात एक पॅटर्न दिसत असेलच: n adds नंतर, ह्याठिकाणी n ही संख्या 2 चा घात आहे (power of 2), एकूण किंमत $2n - 2$ एकक असते, म्हणजे सरासरी वेळ (किंमत) प्रति add 2 एकक पेक्षा थोडे कमी आहे. जेव्हा n ही 2 चा घात असते तेव्हा best case असते, आणि n च्या इतर व्हॅल्यूझसाठी सरासरी वेळ थोडा जास्ती आहे पण ते महत्त्वाचे नाहीये. महत्त्वाचे हे आहे की तो $O(1)$ आहे.

आकृती २.१ ह्याचे चित्रस्वरुपात वर्णन करते. प्रत्येक चौरस कामाचे एक एकक दर्शवतो. रकाने प्रत्येक add चा एकूण वेळ (किंमत) डावीकडून उजवीकडे ह्या क्रमाने दाखवतात: पहिल्या दोन `adds` ना प्रत्येकी 1 एकक लागतो, तिसऱ्याला 3 एकक, इ.



आकृती २.१: hash-table add ची किंमत.

Rehashing चे अतिरिक्त काम उंची वाढत जाणाऱ्या मनोऱ्यांसारखे दिसते ज्यांतील अंतरही वाढत जाते. आता तुम्ही हे मनोरे जर पाडले आणि resizing ची किंमत सर्व adds ना वाटली तर तुम्ही चित्रस्वरुपात हे बघू शकता की n adds नंतर एकूण किंमत $2n - 2$ आहे.

ह्या अल्गोरिदमचे एक महत्त्वाचे वैशिष्ट्य म्हणजे आपण जेव्हा HashTabel resize करतो तेव्हा त्याची वाढ geometrically (भूमितीय श्रेणीमध्ये) होते; म्हणजेच आपण साइझला एका constant ने गुणतो. जर तुम्ही साइझ arithmetically (अंकगणितीय श्रेणीमध्ये) वाढवली—प्रत्येक वेळी constant संख्येने वाढवली—तर add चा सरासरी रनिंग टाइम linear होईल.

हे इंप्लेमेंटेशन तुम्ही पुढील लिंकवरून डाऊनलोड करू शकता, पण लक्षात घ्या की ते खरेच वापरायची काहीच गरज नाही; जर तुम्हाला एक मॅप पाहिजे असेल तर पायथॉन डिक्शनरीच वापरा: http://thinkpython2.com/code/Map.py.

प.२.५ शब्दार्थ

अल्गोरिदम विश्लेषण (analysis of algorithms): अल्गोरिदम्सची तुलना करण्याचे तंत्र ज्यात त्यांना लागणारा वेळ (running time) आणि मेमरीची जागा (memory/space usage) हे निकष वापरले जातात.

machine model: अल्गोरिदमचे वर्णन करण्यासाठी वापरलेले कॉंप्युटरचे साधे मॉडेल.

worst case: असे इनपुट ज्यावर अल्गोरिदम सर्वांत हळू चालतो (किंवा सर्वांत जास्ती space वापरतो).

leading term: एका बहुपदीमधील सर्वांत जास्ती घात (exponent) असलेले पद (term)

crossover point: ती इनपुट साइझ ज्यात दोन अल्गोरिदम्सना सारखाच रनिंग टाइम किंवा space लागते.

order of growth: अशा फंक्शन्सचा संच ज्यांच्या वाढण्याचा दर अल्गोरिदम-विश्लेषण संदर्भात सारखा मानला जातो. उदा., सर्व समप्रमाणात वाढणारी फंक्शन्स (linear functions) एकाच order-of-growth मध्ये असतात.

Big-Oh notation: Order-of-growth व्यक्त करण्याचे नोटेशन; उदा., $O(n)$ linearly वाढणाऱ्या फंक्शन्सचा संच दर्शवते.

linear: ज्याचा रनिंग टाइम इनपुट साइझच्या (निदान मोठ्या साइझेससाठी) समप्रमाणात वाढतो असा अल्गोरिदम.

quadratic: ज्याचा रनिंग टाइम n^2 च्या प्रमाणात वाढतो असा अल्गोरिदम; इथे n इनपुट साइझ दर्शवते.

search: दिलेल्या collection (उदा., लिस्ट, डिक्शनरी) मध्ये एक item शोधणे किंवा तो नाहीये असा निष्कर्ष काढणे.

hash-table: key-व्हॅल्यू जोड्यांचा समूह दर्शवणारे एक डेटा स्ट्रक्चर ज्यात सर्च constant time मध्ये होतो.

सूची