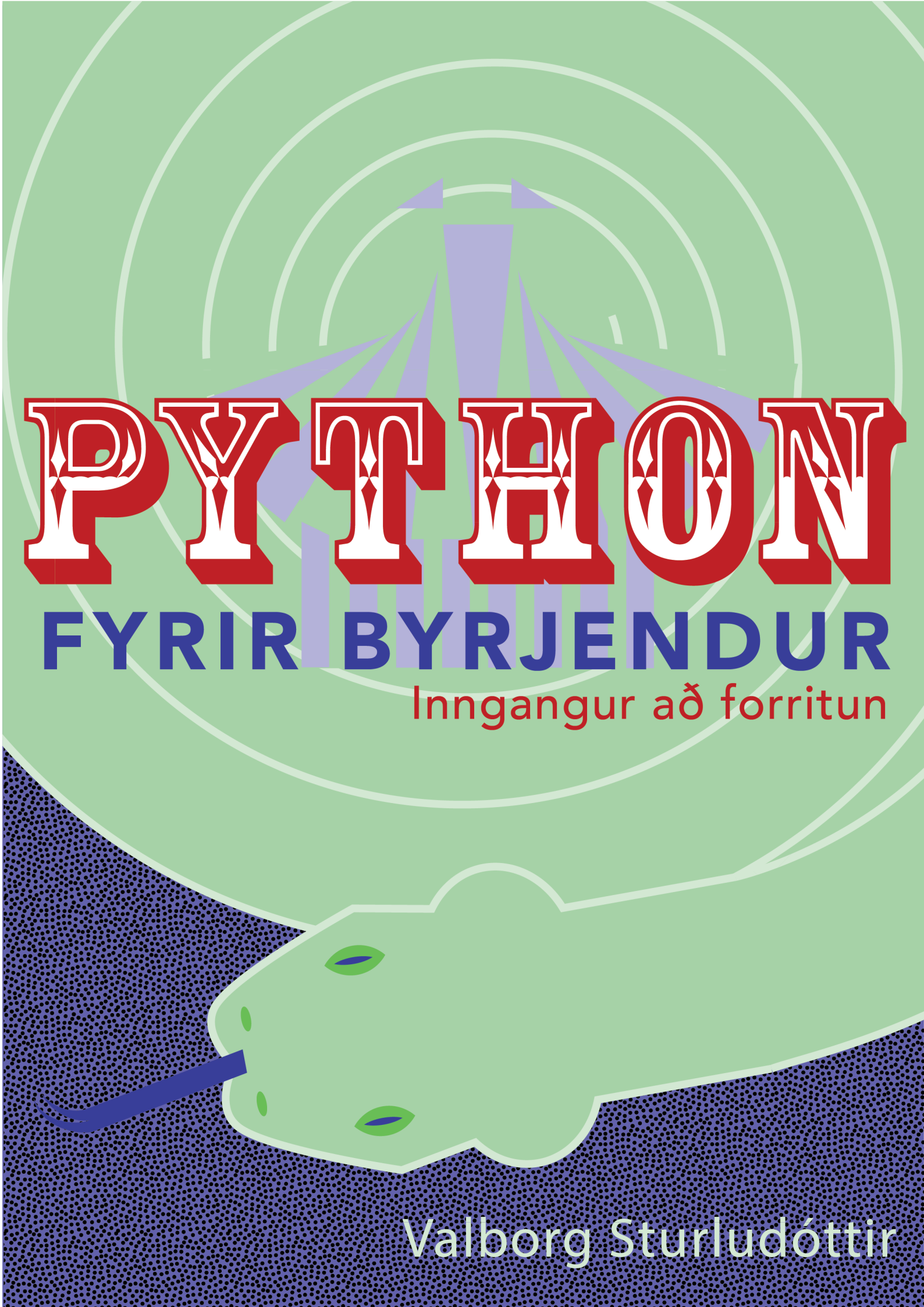
PYTHON

FYRIR BYRJENDUR

Inngangur að forritun



Valborg Sturludóttir

Fyrstu drög, janúar 2022

Efnisyfirlit

Fyrri hluti
Grunnurinn

1. Inngangur

1.1 Tilgangur bókarinnar

Þessi bók fjallar um þau undirstöðu atriði sem þarf að kynna til að ná tökum á forritun í Python. Höfundi finnst mikilvægt að kenna námsefnið með íslenskum hugtökum þar sem ætlunin er að nota hana í kennslu í íslenskum framhaldsskólum. Ef nemendur ætla að leggja fyrir sig tölvunarfræði í framhaldsnámi er nauðsynlegt að búa yfir ríkulegu íðorðasafni, þess þá heldur ef nemandi hyggst framfleyta fræðunum. Hugtök verða þó líka sett fram á ensku því lesandi gæti óskað að fletta upp ítarefni sem meira er til af á netinu á ensku en íslensku.

Það er algengur misskilningur að forritarar kunni rosalega mörg forritunarmál, eins og fólk sem getur talað mörg tungumál, eða það að kunna rosalega mörg mál geri þig að góðum forritara. Þvert á móti. Að sýna hæfni og leikni í einu máli er auðveldlega yfirfæranlegt á önnur mál sé þess þörf. Þess vegna er út í hött að spyrja: „hvað kanntu mörg forritunarmál?" Ekki aðeins eru tungumál og forritunarmál gerólík, forritunarmál eru formleg mál og þekking á einu hlutbundnu máli er nær því að vera jafn frábrugðið öðru í grunninn eins og mállýskur innan tungumála. Nær væri að spyrja hvort viðkomandi hafi meiri áhuga á framenda eða bakenda forritun, hvert er skemmtilegasta reikniritið sem viðkomandi hefur útfært eða hvert er það forritunarmál sem viðkomandi grípur oftast í.

Einnig er það algengur misskilningur að það fyrsta sem fólk gerir er að búa til tölvuleik. Það þarf mikla undirstöðu kunnáttu til þess að geta búið til tölvuleiki, alveg eins og áður en hafist er handa við að skrifa bók þarf að læra stafrófið. Þessi grunnvinna finnst mörgum vera leiðigjörn. Að mati höfundar er það vegna þess að við erum svo vön því að nota tölvur dagsdaglega, svo fræðigreinin sem tæknin byggir á hlýtur líka að vera okkur kunnug ekki satt? Nei, alveg eins og dýralækningar eru okkur ekki augljósar við að eiga gæludýr og pípulagnir heldur ekki við að nota klósett. Innan tölvunnar eru ákveðnar grunneiningar sem eru notandanum ekki augljósar, af góðri ástæðu, það væri hrikalegt ef við þyrftum öll að vera píparar til þess að geta notað klósett. Þó að þessi samlíking hafi verið heldur gróf þá sýnir hún að það eru svo margir hlutar sem eru okkur huldir að við hreinlega vitum ekki hvað við vitum ekki. Því er nauðsynlegt að læra grunninn vel og fara rólega yfir hann svo þegar við ætlum að fara að afrita og líma kóða, frá síðum eins og stackoverflow, þá vitum við allavega hvað sá kóði gerir (nokkurn veginn).

Það er gífurlega mikilvægt að kunna að nota tólin sem við nýtum okkur til þess að við getum nýtt þau skynsamlega. Fartölvur í dag eru mun öflugri en tölvan sem kom fólki á tunglið upphaflega en þó ósennilegt að margar þeirra séu nýttar í eins flókna hluti og geimferðir. Hins vegar lendum við oft í því að þurfa að leysa einhver tiltekin verkefni og ef við gætum nýtt okkur vélaraflið betur værum við sneggri að því.

Það síðasta sem er þess virði að taka fram er mikilvægi opins (e. open source) og frjáls (e. free) hugbúnaðar. Þessi bók er gefin út undir Creative Commons 3.0 sem þýðir að hver sem er má afrita hana, prenta hana og nota með því skilyrði að afleidd verk vísi til höfundar en að óheimils sé að hagnast efninu.

Internetið eins og við þekkjum það myndi ekki keyra ef það væri ekki fyrir framlög forritara sem viðhalda opnum hugbúnaði. Stefnan í heiminum í dag virðist þó vera í áttina að lokuðum réttindavörðum hugbúnaði sem er mjög miður.

Það að hugbúnaður sé opinn þýði að hver sem er geti skoðað kóðann á bak við hann, gert breytingar og deilt með öðrum. Hugmyndin á bak við frjálsan hugbúnað er svipuð nema lagt er upp með þá hugmynd að það sé siðferðisleg skylda fólks að hugbúnaður sé sem aðgengilegastur því frelsi í hugbúnaði skilar sér í frjálsara samfélagi.

1.2 Hvers vegna Python?

Ástæður þess að Python er gott mál til þess að byrja á að skoða eru eftirfarandi: [1].

1. *Málskipanin* er mjög svipuð mannlegu máli svo það ætti að vera auðvelt að læra hvernig eigi að „tala" við tölvuna.
2. Python er *kvikt tagað* forritunarmál, það þýðir að notandinn þarf ekki að gefa upp hvers konar *gagnatýpur* er unnið með. Þetta gerir það að verkum að notandinn þarf ekki að læra urmull af lykilorðum áður en byrjað er að forrita.
3. Python er ekki alveg *hlutbundið* forritunarmál, sem gerir það að verkum að notandinn þarf ekki að læra hvernig á að beita hlutbundinni forritun fyrr en góð undirstaða er þegar komin.
4. Python er frítt og aðgengilegt öllum helstu stýrikerfum og einnig er hægt að forrita yfir netið í vafra og því óþarfi fyrir notandann að setja nokkuð upp sé þess óskað.
5. Python er æðra forritunarmál (e. high level programming language), ekki það að margir séu að kenna assembly í dag.
6. Python er mikið notað, algengt mál svo það er praktískt að hafa undirstöðu skilning á því.
7. Nefnt í höfuð á Monty Pyhton grínhópsins.

1.3 Uppsetning

Uppbygging bókarinnar er þannig að fyrri hlutinn snýr að því að kynna lesandann fyrir grunnvirkni Python; málskipan, lykilhugtök og lykilorð, gagnatýpur, lykkjur og föll. Seinni hlutinn snýr svo að því að beita þekkingu úr fyrri hlutanum í hlutbundinni forritun, þar verða kynntir til sögunnar klasar og aðferðir sem lesandinn útfærir upp á eigin spýtur. Í lok hvers kafla eru svo verkefni til að reyna á leikni lesandans, lausnir við þeim öllum má finna í lausnarhluta aftast í bókinni.

Ekki er búist við neinni fyrri kunnáttu við lestur þessarar bókar, hún á að geta staðið fyrir sínu án þess að lesandinn búi yfir nokkurri þekkingu á sviði tölvunarfræði eða forritunar. Ef slík þekking er fyrir hendi gæti lesandanum þótt ágætt að fara hratt í gegnum fyrri hluta bókarinnar og einbeita sér að verkefnum úr seinni hlutanum.

Víðs vegar um bókina má finna númeraða kóðabúta. Ástæðan fyrir því er að þessum kóðabútum er auðveldara að viðhalda heldur en skjáskotum úr vinnubókum og því er heldur vísað í bækur sem eru aðgengilegar lesendum og frumstæðari framsetning ræður heldur ríkjum hér.

[1] Strax í þessum texta koma fyrir hugtök sem verða skýrð betur seinna, ekki missa kjarkinn.

Kóðabútur 1.1: Kóðabútar kynntir til sögunnar

```
# Svona líta kóðabútar út
# kóðann má allan afrita og keyra til að sjá þá virkni sem verið er að kynna
```

```
# Svona lítur svarið út þegar kóðabúturinn fyrir ofan er keyrður
# Notið úttakið til að bera við eigin niðurstöður
```

Einnig eru á nokrum stöðum stuttar efnisgreinar af ítarefni sem er ekki nauðsynlegt að hafa fullan skilning á en þó gott að skoða, sérlega fyrir þá lesendur sem vilja leggja frekara nám í tölvunarfræðum fyrir sig. Þær líta svona út:

Ítarefni 1.1 Titill á ítarefni Þennan texta má leiða hjá sér við flýtilestur en gott að hafa skilning á ef lesandi vill ná góðum tökum á efninu.

1.4 Að keyra kóða

Það fyrsta sem nemendur vilja yfirleitt gera er að byrja að skrifa sinn eigin kóða. Áður en við komumst svo langt þarf að útskýra hvernig það er gert. Þessi kennslubók byggir á notkun Jupyter Notebooks með hjálp Anaconda hugbúnaðarins, sem er öflugt pakkakerfi og tólakista sem hefur upp á mikið meira en bara Jupyter að bjóða. Hægt er að nálgast Anaconda á anaconda.com. Hægt er að nota Jupyter án þess að ná í Anaconda með síðum eins og cocalc.com. Einnig er hægt að keyra kóða á netinu í gegnum síður eins og repl.it, nota ritla (eins og notepad eða sublime) og keyra .py skrár í skipanalínu, eða nota þyngri umhverfi eins og pycharm sem eru sérhönnuð fyrir hugbúnaðarþróun. Hér er gert ráð fyrir Juptyter umhverfinu og verður bókin öll miðuð að því.

er verið að gera fyrir því?

Þessari bók fylgja einnig vinnubækur úr Jupyter sem lesandinn getur nýtt sér . Virkninni er skipt upp í sellur og keyrsluröð sellanna skiptir máli, við sjáum seinna mikilvægi þess að geta skipt upp kóða svona og hvers vegna þetta umhverfi er þægilegt til að byrja í. En hver sella hefur aðgang að svokölluðu skilgreiningarsvæði vinnubókarinnar en er þó sín eigin eining, því má keyra eina sellu í einu án þess að keyra allan kóðann í vinnubókinni. Þetta er þægilegt ef við lendum í því að fá villu í einni sellu þá hefur ekki áhrif á neina aðra sellu og við getum haldið áfram.

setja hlekk á þæ

En við keyrslu á kóða þarf einnig að hafa í huga að tölvan gerir nákvæmlega það sem við segjum henni að gera og ekkert annað. Og þá komum við niður á stórt vandamál, að tölvur eru mjög bókstaflegar og vitlausar. Þær skortir allt vit, þær reyna ekki að hafa vit fyrir okkur. Þær gera nákvæmlega það sem við biðjum um. Nákvæmlega eins og við biðjum um það.

Þannig að ef ég ætlaði að segja tölvu að smyrja handa mér hnetusmjörs og sultu samloku þá þyrfti ég að segja vélinni að gera eftirfarandi í nákvæmlega þessari röð:

1. taka fram hníf
2. taka fram tvær brauðsneiðar
3. opna hnetusmjörið
4. setja beitta endann ofan í hnetusmjörið þannig að hann nái upp 50gr af hnetusmjöri
5. setja hnetusmjörið sem er á hnífnum á miðja brauðsneiðina
6. nota hnífinn til þess að smyrja hnetusmjörinu á þá hlið sem hnetusmjörið er nú þegar á, og enga aðra
7. taka fram skeið
8. opna sultuna
9. setja kúpta enda skeiðarinnar ofan í sultukrukkuna
10. taka skeiðina upp úr sultukrukkunni með kúfaða skeið af sultu
11. setja sultuna á hina brauðsneiðina

12. nota skeiðina til að smyrja sultunni yfir þá hlið brauðsneiðarinnar sem sultan er á og enga aðra hlið
13. setja brauðsneiðarnar saman þannig að hnetusmjörið og sultan snertist og hornin mætast öll.

Takið eftir að hér er gert ráð fyrir þó nokkru og ef vélin kann ekki nú þegar skil á eftirfarandi, mun þetta klúðrast:

1. taka fram
2. hnífur
3. opna
4. mæla 50 gr
5. smyrja
6. hlið á brauðsneið
7. miðja á brauðsneið
8. skeið
9. kúfað

Þessi útskýring á samlokugerð kann að vera alveg ofboðslega óþarflega nákvæm þá er ekki víst að úr þessu verði nokkur samloka. Þetta könnumst við öll við, að tölvur gera það sem þeim er sagt, ekki það sem við viljum.

Helsta verkefni forritara er að búta niður verkefni í svo litla hluta að hægt er að útskýra þá fyrir tölvu. Ekki búast við því að setjast niður við fyrsta verkefni og ætlast svo til að búa til tölvuleik eða hakka banka. Forritun er einnig frábrugðin þeirri venjulegu tölvunotkun sem við höfum vanist dagsdaglega. Dagsdaglega erum við ekki að gefa tölvunni okkar eigin skipanir heldur að beita skipunum sem aðrir forritara hafa samið og sett upp í hugbúnaðinn sem við erum að nota.

1.5 Málskipan

Málskipan (e. syntax) er safn þeirra reglna sem við þurftum til þess að skrifa kóða sem tölvan skilur, svo að hann þýðist í vélamál, þær reglur sem við þurfum að fara eftir þegar við forritum, þær reglur sem forritunarmálið býst við að við förum eftir. Ef við brjótum þessar reglur fáum við villu, og einhver algengasta villa sem hægt er að fá er málskipanarvilla (e. syntax error). Python er frábrugðið öðrum forritunarmálum á þann hátt að málskipanin krefst þess að kóðinn sé settur upp á ákveðinn hátt. Líkja má því við að þurfa ekki að hafa greinarmerki í huga þegar við ljúkum setningum heldur setjum við orðin okkar á réttan stað í samræðum.

1.5.1 Uppsetning á kóða

Þessi kóðabútur er þannig uppsettur að allar línur byrja jafnlangt til vinstri, eins og hver setning í töluðu máli stendur hver lína fyrir sínu, ein og sér.

Kóðabútur 1.2: Réttur Python kóði

```
1 # Réttur Python kóði sem keyrist
2 4 + 8
3 5 + 6
4 breyta = 9 * 2
```

Þessi næsti kóðabútur hinsvegar er ekki nógu vel uppsettur, þar eru „setningar” sem virðast hanga undir öðrum og vera þeim háðar.

Kóðabútur 1.3: Rangur Python kóði sem veldur villu

```
1 # Illa skrifaður Python kóði sem keyrist ekki
2 4 + 8. # punkturinn veldur málskipanarvillu
```

```
    5 + 6 # inndrátturinn hér er rangur
breyta = 9 * 2 * # málskipanarvilla fæst hér því síðasta táknið er á röngum stað
```

Svona inndrætti er einungis beitt ef lína á beinlínis að hanga undir línunni að ofan og tilheyrir henni. Þess vegna þarf að huga að því hvernig kóði er uppsettur. Í öðrum málum eru notuð greinamerki til að segja tölvunni að lína sé búin og að aðrar línur eigi að heyra undir eitthvað ákveðið samhengi en ekki í Python, þar er treyst á að forritarinn setji kóðann upp á máta sem hægt er að sjá að sé réttur. Dæmi um hvernig línur geta verið aðgreindar í öðrum málum:

Kóðabútur 1.4: Dæmi um annað mál sem er strangt tagað og með greinamerkjum

```
// Dæmi um kóða í forritunarmálinu Java
int i = 7;
i + 5;

// Þetta myndi líka ganga í Java en ekki í Python
int i = 7; i + 5;
```

Kóðabútur 1.5: Dæmi um annað mál sem byggir á afmörkuðu samhengi en með greinamerkjum

```
; Lisp
(setq x 10)
(setq y 34.567)

(print x)
(print y)
```

Í þessum tveimur frábrugðnu málum sem voru tekin sem dæmi var óþarfi að setja kóðann í mismunandi línur, því greinamerkin væru nóg til að aðgreina hverja línu fyrir sig. Hins vegar er það góð venja að skrifa kóða sem er læsilegur öðru fólki. Í Java eru greinarmerkin semikommur (;) en í Lisp eru línur og samhengi afmörkuð með svigum. Python byggist hinsvegar á því að forritarinn stilli öllu upp rétt með réttum inndrætti.

1.5.2 Gagnatýpur og lykilorð

Í Python eru nokkrar grunn gagnatýpur sem við munum kynnast í þessari bók. Ástæðan fyrir því að þær eru kallað grunntýpur er sú að þær fylgja með Python uppsetningunni og notandinn getur beitt þeim í samræmi við það sem þær eru færar um, sem má skoða í skjölun Python https://www.python.org/doc/. Týpa eða tag er hugtak sem þýðir að hlutur sé af einhverri ákveðinni tegund sem má framkvæma ákveðnar aðgerðir á, í þessari bók verða týpur ýmist kallarðar það eða tög. Lesandi þekkir muninn á orðum og tölum úr daglegu tali og veit að hægt er að framkvæma mismunandi aðgerðir á þessum mismunandi týpum, eins og hægt er að skipta út hástöfum fyrir lágstafi í orðum en ekki tölum og hægt er að hefja tölur í veldi en ekki orð. Að sama skapi eru til aðgreinanlegar týpur sem tölvan kann skil á og leyfir ákveðnar aðgerðir á. Í fyrri hluta þessarar bókar verða gerð skil á tveimur talnatýpum (heiltölum og fleytitölum), strengjum, listum, sanngildum, orðabókum (einnig kallaðar hakkatöflur), nd-um og mengjum.

Lykilorð eru orð sem eru frátekin og birtast þau græn í Jupyter vinnubók. Hver gagnatýpa hefur eitt lykilorð og eru einnig nokkur innbyggð föll í Python, sem við kynnumst fljótlega, með frátekin orð. Forðast skal að yfirskrifa þessi lykilorð, en gerist það þá er auðvelt að laga það í Jupyter. Hver vinnubók hefur sinn kjarna til að vinna á og það eina sem þarf að gera í aðstæðum þar sem innbyggt orð er allt í einu farið að þýða eitthvað annað þá dugir að endurræsa kjarnann. Kjarninn í vinnubókinni er hvaða túlk eða þýðanda er verið að nota til þess að láta tölvuna skilja kóðann. Í okkar tilfelli erum við að nota Python 3.

2. Tölur og breytur

Í þessum kafla ætlum við að hefjast handa við að forrita. Það fyrsta sem við ætlum að gera er að kynnast *talnatýpum* og keyra kóða eins og við værum að nota reiknivél. Við könnumst við reiknivélar og hvernig þær afgreiða röð aðgerða. Nú viljum við sannreyna að þær reikniaðgerðir sem við þekkjum séu til í Python og að þegar við keyrum kóðann okkar þá verði útkoman sú sama og við áttum von á. Við viljum líka geta geymt útkomuna okkar til að nota aftur seinna, til þess þurfum við *breytur* (e. variables).

2.1 Tölur - talnatýpur

Í Pyhton eru í grunninn tvær týpur af tölum (en til eru tvær týpur af hvorri fyrir sig, sem snýr meira að minnisnotkun og er út fyrir svið þessarar bókar). Þær eru:

- **Heiltölur** - tölur sem eru ekki með neinum aukastaf. Á ensku eru þessar tölur kallaðar *integers* og er lykilorð þeirra því **int**.
- **Fleytitölur** - tölur sem eru með aukastaf, sem er fyrir aftan punkt (ekki kommu, fleytitölur eru oft kallaðar kommutölur á íslensku). Á ensku eru þessar tölur kallaðar *floating point numbers* og er því lykilorðið þeirra **float**.

Kóðabútur 2.1: Heiltölur og fleytitölur

```
# Heiltölur, enginn aukastafur
42
-139

# Fleytitölur, aukastafur/ir fyrir aftan punkt
4.0
-3.1415926
```

```
-3.1415926
```

Heiltölur og fleytitölur eru sambærilegar og má nota þær saman í reikningi. Báðar talnatýpurnar ná niður í ansi mikinn mínus og ansi hátt upp. Í kóðabút 2.1 sjáum við tvennt áhugavert, þegar við

keyrðum þennan kóða þá kom úttakið `-3.1415926`, af hverju skyldi það vera? Þetta gerist í Jupyter Notebooks, úttakið verður það síðasta sem við *köllluðum* í, eða báðum vélina um að segja okkur.

2.2 Reikniaðgerðir og tákn

Grunnreikniaðgerðir eru nokkrar sem við könnumst við úr grunnskóla en aðrar eru framandi og við skulum skoða aðeins betur.

Táknin eru flest eins og á reiknivélum +, -, *, / en þar að auki er annars konar deiling sem er táknuð með tveimur deilimerkjum //, veldishafning er táknuð með tveimur margföldunarmerkjum **, og svo er leifareikningur táknaður með %. Munurinn á deilingunni er sá að með einu skástriki er beitt hefðbundinni deilingu[1] en tvö skástrik er heiltöludeiling.

Í eftirfarandi dæmum í kóðabút, 2.2, er vert að draga fram nokkur atriði sem eru ekki augljós byrjanda. Það fyrsta er að myllumerkið (#) þýðir að allt sem kemur fyrir aftan það er *athugasemd* (e. comments), þær eru engöngu til að gera kóða læsilegri fyrir fólk, þær eru hunsaðar af tölvunni þegar hún breytir kóðanum í eitthvað sem hún skilur.

Eins og sést í línu merktri númer 20 er athugasemdin svo löng að hún birtist okkur sem tvær línur, en hún er í keyrslu tölvunnar álitin ein heild línu 20. Þess vegna þurfum við ekki að hafa áhyggjur af þessum inndrætti sem birtist, hann er í rauninni ekki til staðar þar sem þessi hluti textans er ein lína. Einnig eru bil á milli talna og tákna, það er líka til að gera kóðan læsilegri, bilin mega bara ekki vera fremst í línunni.

Athugasemdir í kóða eru mjög mikilvægur hluti af *skjölun* (e. documentation) og ættu öll sem vilja tileinka sér forritun að venja sig á að skrifa athugasemdir. Í fyrstu, skrifum við ekki flókinn kóða svo athugasemdirnar segja okkur ekki mikið, en þegar kóðinn er ekki augljós eða lausn á verkefni flókin er gott að skrifa athugasemdir. Kóði sem þið komið til með að skrifa á einnig að vera ykkur sjálfum skiljanlegur þegar þið komið að honum seinna. Því er gott að venja sig strax á að skrifa lýsandi athugasemdir.

Kóðabútur 2.2: Reikniaðgerðir

```
1  # Samlagning framkvæmd með +
2  6 + 4
3
4  # Frádráttur framkvæmdur með -
5  14 - 4
6
7  # Margföldun framkvæmd með *
8  10 * 2
9
10 # Deiling framkvæmd með /
11 60 / 6
12
13 # Heiltöludeiling framkvæmd með //
14 177 // 17
15
16 # Veldishafning framkvæmd með **
17 3 ** 2
18
19 # Leifareikningur framkvæmdur með % (e. modulus)
20 177 % 17
```

```
1  7
```

[1] stundum kölluð fullkomin deiling

Hér vorum við einnig að biðja tölvuna um að reikna einhvern heilan helling en við báðum vélina aldrei um að segja okkur neitt frá útkomunni sérstaklega eða geymdum hana neinstaðar. Í kóðabút 2.2 sést hvernig á að keyra kóða án þess að fá einhverjar villur (sjá kafla 2.4) eða gera neitt sérstakt. Markmiðið hjá okkur núna er að skilja hvernig reikniaðgerðirnar virka og því ættuð þið núna að prófa ykkur áfram, skoða forgangsröð aðgerða[2], og þora að gera villur og reyna að lesa í þær. Nánar um hvernig þessir *reiknivirkjar* (e. arithmic operators) verka má sjá á síðu W3Schools, sem er frábær síða til að læra meira um forritun, hér.

Ítarefni 2.1 Heiltöludeiling og leifareikningur
Heiltöludeiling og leifarreikningur eru líklega ný á nálinni fyrir flestum lesendum og því allt í lagi að útskýra þær aðgerðir aðeins nánar. Þessar aðgerðir eru einmitt mjög skyldar í raun. Deilingin segir okkur hversu oft ein tala gengur upp í aðra þar sem útkoman er heil tala (eða fleytitala með 0 sem eina aukastafinn), okkur er sama um afganginn sem verður eftir. Í þessari deilingu er svarið 2 við bæði 5//2 og 4//2. En í leifareikningnum viljum við eingöngu vita hver er afgangurinn þegar heiltöludeilingu er beitt svo 5%2 væri 1, því það er einn í afgang þegar fimm er deilt með tveimur. Og það er 0 í afgang þegar fjórum er deilt með tveimur svo 4%2 er 0.

Það er gott að byrja á því að átta sig á því hvernig á að keyra kóða og í kóðabút 2.2 vorum við vissulega að keyra kóða. Hann gerði ekkert merkilegt en við erum strax byrjuð að forrita. Við skulum svo ekki gleyma þessum undirstöðu reikniaðgerðum með þessum reiknivirkjum þegar við höldum áfram og gerum flóknari hluti[3].

Reiknivirkjarnir gera svo ólíka hluti eftir því með hvaða týpum þeir eru notaðir, í þessum kafla notum við eingöngu talnatýpur en munum seinna sjá þessa sömu reiknivirkja skila okkur öðrum niðurstöðum með öðrum týpum.

2.3 Breytur

Nú höfum við séð hvernig má keyra kóða einfaldlega eins og í reiknivél. Höldum okkur við samlíkinguna um reiknivélina til að útskýra *breytur* (e. variables). Á hefbundinni reiknivél sem notuð er í stærðfræðitíma í framhaldsskóla er takki sem á stendur ANS (stytting á enska orðinu *answer*). Vélin getur geymt síðasta svar sem fékkst og þegar ýtt er á þennan takka sækir vélin úr minni það svar. Flottari vélar geta svo geymt nokkuð mörg svör en það er útfyrir gagnsemi þessarar samlíkingar. Þegar ýtt er á þennan takka er gildi sótt úr minnisvæðinu og á skjáinn kemur textinn ANS en á bakvið tjöldin er reiknivélin að nota gildið úr minnissvæðinu þó að við sjáum ekki hvað það er, sem er mjög gagnlegt þegar talan sem við ætluðum að nota er eitthvað á við $\frac{3}{4}\sqrt[3]{\pi * 7}$ og við nennum ekki að skrifa það aftur og aftur.

Að sama skapi má láta Python úthluta minnissvæði í tölvunni fyrir þær breytur sem við viljum geyma. Munurinn er sá að við nefnum sjálf hvað minnisvæðið heitir, erum ekki bundin við að nota ANS og að við erum með, svo gott sem, óteljandi minnissvæði.

Að gefa minnissvæði merkingu og gildi er gert með *gildisveitingu*. Gildisveiting (e. assignment) þýðir að nú er einhver ákveðinn merkimiði kominn með eitthvað til að geyma. Gildisveiting er gerð með því að skrifa nafnið sem við viljum geta vísað í vinstra megin við jafnaðarmerki og gildið sem

[2] https://is.wikibooks.org/wiki/Forgangsröð_aðgerða

[3] Í öllum verkefnum í kóðabút 2.2 var verið að vinna með heiltölur, þó var útkoman úr deilingunni fleytitala. Ef við myndum skipta út hverri tölu fyrir sig og setja í staðinn sömu tölu með .0 fyrir aftan þá yrðu útkomurnar þær sömu nema fleytitölur. Ef við notum ólíkar týpur, og slíkt er vandmeðfarið, er það í lagi í þessu tilfelli þar sem Python gerir þá ráð fyrir að það sé í lagi að reikna allt með fleytitölum og framkvæmir reikninginn eins og við höfum verið að beita fleytitölum í hvívetna og niðurstaðan verður þá að sjálfsögðu fleytitala.

við viljum geyma í þeirri breytu hægra megin við jafnaðarmerkið, eins og sést í kóðabút 2.3

Kóðabútur 2.3: Breytur kynntar

```
1 val = 5
2
3 val + 5
```

```
1 10
```

Þegar við keyrum línu 1 í kóðabút 2.3 segjum við vélinni að hafa aðgengilegt minnissvæði sem við getum notað með því að skrifa orðið `val`, og setja í það minnissvæði gildið 5. Breytan er svo notuð í línu 3, nafnið hennar er notað í útreikningi sem sambærilegum kóðabút 2.2. Það sem gerist er að við veitum breytinni `val` gildi og svo köllum við í breytuna okkar með því að skrifa nafnið hennar. Prófið að breyta röðinni og sjá hvað gerist, en hvað gerist ef þið breytið bæði röðinni á línunum tveimur og nafninu á breyunni? Prófið ykkur áfram.

Ítarefni 2.2 Hvað gerist þegar við keyrum kóða?
Ef þú prófar þig áfram við að búa til breytur í vinnubók og keyra sellur gætir þú rekist á svolítið sem hefur ekki gerst áður, að þegar sella inniheldur eingöngu gildisveitingu og er keyrð þá „gerist ekkert". Þetta finnst mörgum mjög skrýtið því þau vilja fá einhverja útkomu. En útkoman er sú að þið sögðuð vélinni að geyma eitthvað, þið sögðuð henni ekki að gera neitt annað.

Breytur eru skilgreindar vinstra megin við jafnaðarmerki í Python. Eins og það væri lesið, `val` fær gildið 5. Það væri lítið vit í því að hafa það öfugt, 5 er núna jafngilt `val`. Það sem við værum þá að segja tölvunni að í hvert sinn sem hún vill nota heiltöluna fimm þá á hún að hætta við að nota töluna sjálfa og í staðinn vísa eingöngu í það sem er í minnissvæði merktu `val`. Það er alls ekki það sem við viljum (eða mjög ólíklega).

Nokkrar **reglur** í nafnavali á breytum, þetta vill vefjast fyrir sumum en lærist fljótlega:

1. Kóðalitunin á breytuheitinu má ekki vera annað en venjulegi liturinn fyrir kóða, þannig að ef nafnið fær áherslumerkingu (annan lit) er það ekki löglegt breytuheiti því það er frátekið lykilorð í Python. Dæmi um það sem fær áherslulitun eru frátekin lykilorð og tölustafir. Athugið að áherslulitunin í númeruðu kóðabútunum í þessari bók er mjög frumstæð.
2. Breytuheitið ætti ekki að innihalda séríslenskan staf (það er löglegt í jupyter vinnubókum en er hrikalega slæmur ávani því það er ekki löglegt allsstaðar).
3. Breytuheitið má ekki innihalda bil.

Nokkur **tilmæli** um breytunöfn með tilliti til nafnavenja í Python:

1. Breytuheiti byrja á litlum staf.
2. Ef það þarf að nota löng breytuheiti er venjan að nota snákaframsetningu (e. snake casing) sem felur í sér að gera niðurstrik á milli orða, dæmi `thetta_er_langt_nafn_a_breytu`. Annars er til kamelframsetning (e. camel casing) sem felur í sér að annað hvert orð er með stórum staf, dæmi `thettaErLikaLangtBreytuheiti`. Hvort sem þið endið á að nota meira, haldið ykkur bara við annað þeirra.
3. Breytuheiti eiga að vera lýsandi. Ef ég væri að reikna hliðar í þríhyrningi væri gott að eiga breyturnar `a`, `b` og `c`. En ef ég væri að búa til reiknirit sem býr til tölvuleikjapersónu af handahófi með því að velja tilviljanakennt nafn, aldur og starf þá væru breytuheitin `a`, `b` og `c` alveg glötuð því þegar ég kæmi aftur að kóðanum mínum myndi ég ekki hafa hugmynd um

hvað a, b og c væru. Betra væri að breyturnar hétu `nafn`, `aldur` og `starf`.

Í kóðabút 2.4 eru nokkrar gildisveitingar, það er látið eftir sem æfing að laga þennan kóðabút þannig að hann keyri villulaust (án þess að gera allt að athugasemdum) og átta sig á því hvað var að hverju sinni. Ein gildisveiting er þó svo svæsið vitlaus að þið þurfið að endurræsa kjarnann í Jupyter vinnubókinni ykkar til að geta haldið áfram (fyrir hvert skipti sem þið keyrið kóðabútinn án þess að hafa lagað villuna).

Kóðabútur 2.4: Dæmi um gildisvetingar réttar og rangar

```
val = 5

5 = val

heiltala = 0

int = 0

Gott nafn = 1.0

3_litlar_mys = 3

utreiknud_laun_eftir_skatt = 0.65 * laun
```

Nú þegar við höfum séð hvernig má skilgreina breytu með gildisveitingu viljum við vita hvernig á að nota þessa breytu. Ef við snúum okkur aftur að reiknivélasamlíkingunni um ANS takkann þá ætti kóðabútur 2.5 að geta sýnt með eðlislægum hætti hvernig breytur nýtast. Fyrst segjum við vélinni hvað það er sem ANS á að vísa á. Svo í línu 5 segjum við vélinni að geyma aðra breytu, `x`, sem byggir á `ANS` breytunni okkar. Í þessum kóðabút er svo haldið áfram með þessa afleiddu breytu `x` og önnur afleidd breyta búin til útfrá henni. Það sem gerist svo í endann er sambærilegt við það að ýta á „=” takkann á reiknivélinni.

Takið eftir að þarna er notuð ný framsetning sem við höfum ekki séð áður, þarna stendur `print` með svigum fyrir aftan og inni í svigunum er breyta. Ef þessi kóðabútur er keyrður þá kemur á *staðalúttak* [4] það gildi sem breytan inniheldur. Ef þar hefði staðið `print(halft_x)` hefðum við fengið svarið sem er geymt í breytunni `print(halft_x)`.

Kóðabútur 2.5: Að nota breytu

```
ANS = 3/4 * (3.1415 * 7)**(1/3)
print(ANS)
x = 3 * ANS
print(x)
halft_x = x/2
print(halft_x)
```

```
2.1012269615581634
6.30368088467449
3.151840442337245
```

Í línu 1 í kóðabút 2.5 er breytan `ANS` skilgreind, í línum 2 og 3 er hún svo notuð, Í línu 3 er breytan `x` skilgreind og í línum 4 og 5 er hún notuð, sama gildir um `halft_x` Breytu má einungis nota eftir að hún hefur verið skilgreind. Ef við reynum að vísa í breytu án þess að hafa tekið frá minnishólf með sama nafni þá lendir vélin í vandræðum. Eins ef við værum að segja vinum okkar sögu og nefna Hafstein á nafn án þess að kynna hann myndi viðmælendur okkar truflast og biðja

[4] Þann stað sem texti myndi prentast þegar forritið er notað, hvort sem það er á skjá eða beint á pappír úr prentara eða eitthvað allt annað. Kannski verður úttaki varpað beint inn í heilann á forriturum einhvern tíma?

um útskýringu. Gerið tilraun á þessu með því að færa til línurnar í kóðabútnum og sjá hvað gerist þegar kóðinn er keyrður.

Skipunin *print* sem kemur þarna oft fyrir er *fall*, við skoðum föll nánar í kafla 10 en þangað til munum við kynnast nokkrum innbyggðum föllum eins og `print()`, `type()` og `len()`. Við beitum föllunum með því að skrifa nöfn þeirra og setja inn í þau *viðföng* (e. parameters). Með því er átt að þið setjið inn í svigana fyrir aftan print, það sem þið viljið að prentist. Við megum beita print skipuninni óspart, það hjálpar gífurlega við að skilja hvað það er sem er að gerast og hvað, ef eitthvað, er að fara úrskeiðis.

Þið eruð einnig hvött til þess að venja ykkur á að skoða úttakið í hverju þrepi áður en leitað er hjálpar til annarra, einnig má fletta upp í listanum yfir algengar villur í kafla 14.1.

Núna höfum við séð tvær týpur, heiltölur og fleytitölur. Breyta getur innihaldið hvernig týpu sem er. Þá þurfum við að athuga hvað má gera við breyturnar okkar. Hingað til höfum við eingöngu skoðað reikniaðgerðir sem eru framkvæmdar með kunnuglegum táknum, við höfum ekki verið að beita neinum innbyggðum *aðferðum* á tölurnar okkar. Við sjáum það gert í kafla 3 þegar við skoðum hvernig megi vinna með texta.

Að því sögðu þá þurfum við að skoða breytur nokkuð betur áður en við förum að beita þeim á skilvirkan hátt. Við erum búin að skoða reiknivirkja og gildisveitingu, og nú ætlum við að skoða *reiknivirkjagildisveitingu* (e. assignment operators) þar sem við uppfærum gildi í breytu með því að nota reiknivirkja (eins og + eða -) með gildisveitingu (=). Þetta sést betur í kóðabút 2.6 þar sem við byrjum á að búa til breytu því að reiknivirkjagildisveiting snýst um að gefa gömlu minnissvæði gildi sem byggir á því sem þar áður var. Línur 2 og 3 í kóðaútnum eru jafngildar en vegna þess að þær eru báðar keyrðar þá uppfærist gildið á breytunni `nem` tvisvar.

Kóðabútur 2.6: "Reiknivirkjagildisveiting"

```
1  nem = 0
2  nem = nem + 1
3  nem += 1
4  print(nem)
5  nem += 2
6
7  stofn_staerd = 30
8  stofn_staerd *= 2
9
10 thusund = 1000
11 fimm_hundrud = thusund/2
12 tvo_hundrud_og_fimmtiu = fimm_hundrud/2
13 print(tvo_hundrud_og_fimmtiu)
14
15 allt = 1000
16 allt /= 2
17 allt /= 2
18 print(allt)
19
20 laun = 100
21 verdbolga = 0.02
22 laun *=  1 + verdbolga
23 print(laun)
```

```
1  2
2  250.0
3  250.0
4  102.0
```

Hér í kóðabút 2.6 sést að það er gagnlegt og fljótlesið þegar það á að uppfæra gildi á breytu að gera það með því að nota reiknivirkjann með gildisveitingunni. Það er skýrara því við sjáum

þá strax að eiga á við breytuna eins og hún er en ekki með einhverju öðru gildi. Dæmin sem eru tekin eru í röð: fjöldi nemenda aukinn línulega, stofnstærð á einhverri tegund margfaldast, peningafjárhæð minnkuð (í fyrra dæminu eru fleiri breytur sem gæti verið gagnlegt en í því seinna er ein sem breytist) og svo það síðasta er launahækkun sem um nemur einhverri verðbólguspá.

2.4 Villur

Að svo stöddu er gott að hafa í huga hinar ýmsu *villur* (e. errors) sem geta komið upp. Þegar við forritum í Python fáum við yfirleitt í hendurnar mjög lýsandi villur sem segja okkur hvað við gerðum vitlaust og hvar[5]. Áður en lengra er haldið er vitlegt að kynnast þeim algengu villum sem koma upp, hvað þær þýða og hvernig á að taka á þeim.

- **Nafna villa** (e. NameError), þar sem við reyndum að nota breytu sem við höfum enn ekki skilgreint. Þessi villa kemur upp þegar við skrifum nafnið á breytunni okkar vitlaust þegar við ætlum að beita henni eða við gleymdum að skilgreina hana áður en við beittum henni. Til dæmis myndum við fá nafnavillu við að skilgreina breytuna `Valborg` en nota svo breytuna `valborg`, þar sem hástafir og lágstafir skipta máli í Python og eru þessar tvær breytur algerlega óskyldar í minninu. Einnig getur þessi villa komið upp þegar við ætlum að nota strengi en gleymum að skilgreina þá sem strengi með gæsalöppum og látum vélina halda að við séum að vinna með breytur. Sjáum í kafla 3 hvernig á að skilgreina strengi.
  Þetta lögum við með því að sjá hvaða nafn við ætluðum að nota og pössum okkur að það nafn hefur fengið einhverja skilgreiningu áður en að við reynum að nota það (það er ofar í kóðanum).
- **Týpu villa** (e. TypeError), þar sem eitthvað var gert við gögn sem týpan af gögnunum leyfir ekki. Til dæmis að beita reiknivirkja á breytu sem styður ekki notkun hans.
- **Málskipanar villa** (e. SyntaxError), þar sem eitthvað var vitlaust skrifað, vitlaust tákn á vitlausum stað. Til dæmis var komma notuð í stað punkts eða breytuheiti byrjaði á tákni eða tölu.
  Það sem við gerum í því að fá þessa villu er að lesa okkur til um hvað það er sem má gera við týpuna okkar og athuga hvernig við fáum fram þá virkni sem við vildum með því að leysa vandann á annan máta. Hér erum við í rauninni að misskilja hvað má og hvað má ekki og við þurfum að átta okkur betur á því hvað er í boði.
- **Inndráttar villa** (e. IndendationError), þar sem kóðinn er ekki rétt inndreginn. Eins og kom fram í inngangi þá þarf Python kóði að vera vel uppsettur til þess að hann keyri. Við sjáum fyrst í kafla 5 hvernig við beitum inndrætti svo núna ef við fáum þessa villu þá er það vegna þess að við erum með óþarfa bil fyrir framan kóðann okkar.
  Það sem við gerum er að skilja hvað á að vera í hvaða inndrætti og laga kóðann þannig að hann sé vel uppsettur.
- **Vísis villa** (e. IndexError), þar sem við reyndum að ná í sætisvísi sem er ekki til í gögnunum okkar. Í kafla 3 sjáum við týpu sem hefur sætisvísa.
  Það sem við gerum í þessari villu er að átta okkur á því hversu margir sætisvísar eru til staðar og hvaða vísa við vildum fá, laga svo kóðann í samræmi við það sem við ætluðum að sækja.
- **Gildis villa** (e. ValueError), þar sem við reyndum að ná í eða nota gildi úr gögnum sem er ekki til. Gögnin eru til en gildið er ekki til staðar í þeim.
  Athugum hér að við reyndum að sækja eitthvað, næstum því eins og upp úr poka, og í stað þess að segja áttu þetta til? Þá sögðum við við vélina „þú átt þetta til, láttu mig hafa það!". Svo það sem við getum gert til að laga það er að segja „áttu þetta til?" eða „reyndu að láta mig hafa það ef þú átt það". Þessi lausn er ansi óljós að svo stödu þar sem við höfum ekki farið yfir hvernig á að ná í gildi upp úr gögnunum okkar en vonandi verður þetta ljósara þegar þar að kemur.
- **Eiginda villa** (e. AttributeError), þar sem við reynum að ná í eða beita eigindum sem gögnin

[5] td. í kóðabútum 3.2 og 6.13 má sjá villuskilaboð í úttaki

rna aftur í tab

okkar búa ekki yfir. Við getum séð hvaða eigindum gögnin okkar búa yfir með því að lesa skjölun um þau eða kíkja snöggt undir húddið með því að gera punkt fyrir aftan gögnin og ýta á tab takkann og þá sjáum við þau eigindi sem týpan býr yfir. Ef við skoðum þau eigindi sem eru til fyrir heiltölur og fleytitölur sjáum við að þau eru ekki alveg eins og ef við reynum að nota þau sem eru til fyrir annað en ekki hitt á víxl þá fáum við eigindavillu.

Það sem við gerum í þessu er svipað týpu villunni, það er að lesa okkur til um hvaða eigindi gögnin okkar búa yfir og hvað við getum náð í og notað. Eigindi eru mismunandi eftir týpum og er mikilvægt að átta sig á því hvaða eigindi eru í boði hverju sinni svo að við veljum rétta týpu fyrir gögnin okkar.

2.5 Æfingar

Æfing 2.1 Búðu til breytu sem inniheldur heiltölu ■

Æfing 2.2 Búðu til breytu sem inniheldur fleytitölu ■

Æfing 2.3 Búðu til breytu sem inniheldur samlagningu breytanna úr verkefnum 2.1 og 2.2. Af hvaða týpu er þessi þriðja breyta? ■

Æfing 2.4 Gefum okkur að við eigum eftirfarandi kóða, hvernig fáum við breytuna helmingur til að innihalda helminginn af því sem er í breytunni allt. Nú megum við ekki bara skrifa inn töluna 1000 í útreikninginn, við þurfum að beita breytunni *allt* því að hún gæti breyst og við viljum ekki að kóðinn okkar gefi vitlaust svar ef *allt* breytist. Einnig viljum við búa til þriðju breytuna sem á að vera helmingurinn af helmingnum, hvernig fáum við helminginn af breytunni *helmingur*.

```
allt = 1000
helmingur =
```

■

Æfing 2.5 Nú langar okkur að reikna helminginn af breytunni *allt* alveg eins og í verkefni 2.4 nema við viljum alls ekki geyma niðurstöðurnar í breytu heldur viljum við hafa áhrif á breytuna *allt*. Hvernig getum við prentað út fyrst helminginn af *allt* og svo helminginn af því? ■

Æfing 2.6 Hvernig fyllirðu inn í eftirfarandi kóða þannig að hann reikni út langhlið í rétthyrndum þríhyrningi? Þar sem a og b eru skammhliðar þríhyrningsins og þær þarf að skilgreina sem einhverjar stærðir áður en hægt er að reikna c.

```
??
??

c_i_odru_veldi = a**2 + b**2
c = c_i_odru_veldi**0.5
print(c)
```

■

Æfing 2.7 Skrifaðu kóða sem veldur eftirfarandi villum:

a) Nafnavilla.

b) Málskipunarvilla.

c) Inndráttarvilla.

■

3. Strengir

Til þess að geta sýnt og notað texta þarf gagnatýpu til að halda utan um hann. Í flestum æðri forritunarmálum, og Python er ekki undantekning, eru gögn af þeirri týpu kölluð **strengir**. Lykilorð fyrir þessa týpu er **str**.

3.1 Strengir skilgreindir

Til þess að afmarka texta og segja vélinni að fara með hann sem af týpunni strengur þarf að nota tákn. Við þurftum ekki að gera það þegar við skrifuðum tölurnar en nú, og seinna, notum við ákveðin tákn fyrir ákveðnar týpur.

Táknin sem skilgreina strengi eru gæsalappir. Einfaldar gæsalappir eru úrfellingakomman sem venjulega er lengst til hægri á lyklaborði, tvöfaldar gæsalappir eru sér tákn sem er venjulega á sama takka og 2. Einnig er hægt að gera streng með þremur einföldum gæsalöppum. Dæmi um streng væri `"halló heimur"` eða `'halló heimur'`, takið eftir að gæsalappirnar þurfa að passa. Og fyrir lengri texta notum við þrjár einfaldar gæsalappir.

Kóðabútur 3.1: Strengir skilgreindir

```
1 textinn_minn = "halló ég má skrifa mörg orð inn í þessar gæsalappir"
2
3 einfaldar_gaesalappir = 'ég má líka skrifa innan einfaldra gæsalappa'
4
5 thetta_virkar_ekki = 'gæsalappirnar þurfa að passa saman"
```

Í sumum forritunarmálum er munur á því að nota einfaldar og gæsalappir, þar sem einfaldar eru notaðar fyrir staka stafi (sér gagnatýpa) og tvöfaldar fyrir strengi. En það er enginn raun munur á því hvernig Python meðhöndlar þær.

3.2 Strengir og reikniaðgerðir

Við erum búin að sjá að það megi leggja tölur saman og margfalda þær. Nú ætlum við að skoða hvaða reikniaðgerðir er hægt að framkvæma með strengi og hvaða áhrif það hefur. Rifjið

upp reiknivirkjana úr kafla 2.2 og prófið ykkur svo áfram með breyturnar úr kóðabút 3.1, notið reiknivirkjana á breyturnar eða á einhverja breytu og einhverja tölu. Skoðið hvað má og hvað má ekki.

Kóðabútur 3.2: Strengir og reikniaðgerðir

```
1 "halló" // 3
```

```
1 ---------------------------------------------------------------------------
2 TypeError                                 Traceback (most recent call last)
3 <ipython-input-56-720336c54f29> in <module>
4 ----> 1 "halló" // 3
5
6 TypeError: unsupported operand type(s) for //: 'str' and 'int'
```

Þetta eru fyrstu villuskilaboðin okkar í bókinni, við fengum týpuvillu það sést í línu 2 í úttakinu í kóðabút 3.2, okkur er bent á (lína 4) að villan gerist í línu 1 í kóðanum sem var keyrður og að hún sé vegna þess (lína 6) að aðgerðin `//` sé ekki heimil fyrir týpurnar streng og heiltölu.

Hér er aðalatriðið að fá annað hvort eitthvað út á úttakið eða að fá týpuvillu, ef þið fenguð málskipanarvillu (SyntaxError) þá skrifuðuð þið bara eitthvað vitlaust og þurfið að laga það.

Nú þegar þið eruð búin að gera nokkuð margar tilraunir og komast á því að tvær reikniaðgerðir eru leyfilegar.

Þegar við notum + til að setja saman strengi þá erum við að beita *samskeytingu* (e. concatenation). Samskeyting þýðir að einum streng er bætt við fyrir aftan annan streng. Það skiptir máli hvor er fyrir framan: `"halló"+ "bless"` verður að `"hallóbless"` en `"bless"+ "halló"` verður að `"blesshalló"`.

Þegar við notum * til að margfalda streng með heilli tölu erum við að beita *lengingu* (e. multiply) og strengurinn er endurtekinn ákveðið oft. Þannig að `"halló"* 3` verður `"hallóhallóhalló"` en takið eftir að `"halló "* 3` verður `"halló halló halló "`, sjáiði muninn? Í línu 2 í kóðabút 3.3 er þessu einnig beitt, að setja bil þar sem búist er við að strengjum sem skeytt saman. Þetta er eingöngu gert til að einfalda okkur lífið að svo stöddu, við skulum ekki venja okkur á að setja óþarfa bil fyrir aftan strengina okkar. Sjá má hvernig hægt að er að komast hjá þessari bilnotkun í línu 9. Takið einnig eftir hvernig `print()` skipunin er látin prenta út nokkrar breytur með því að setja kommur á milli.

Kóðabútur 3.3: Samskeyting og lenging strengja

```
1  strengur_a = "a"
2  strengur_b = " og b!"
3
4  sameinadir_a_og_b = strengur_a + strengur_b
5  sameinadir_b_og_a = strengur_b + strengur_a
6
7  fyrsta_nafn = "Valborg"
8  seinna_nafn = "Sturludóttir"
9  fullt_nafn = fyrsta_nafn + " " + seinna_nafn
10
11 print(sameinadir_a_og_b, sameinadir_b_og_a, fullt_nafn)
12
13 eitt_ord = "kex"
14 eitt_ord*3
```

```
1 a og b!  og b!a Valborg Sturludóttir
2
3 'kexkexkex'
```

3.3 Vísar í streng

Strengir eru ákveðin röð tákna[1]. Táknin, eða stafirnir, sitja á sínum stað og eru ákveðið mörg. Því er hægt að tala um stafabil og lengd í strengjum. Við getum séð hversu mörg stafabil eru í streng með því að telja þau sjálf eða láta tölvuna segja okkur það með innbyggða fallinu `len()` (stytting á enska orðinu length).

Kóðabútur 3.4: Stafabilafjöldi

```
1 strengur1 = "kex"
2 print(len(strengur1))
3
4 strengur2 = "kex með smjöri, osti og sultu"
5 print(len(strengur2))
```

```
1 3
2 29
```

Nú þegar við vitum hversu mörg stafabil eru í strengnum getum við notað þau. Við getum sagt við vélina mig langar til að fá *vísi*(e. index) (einnig kallað sætisnúmer, sæti og stæði) númer 1 og séð hvaða tákn er í þeim vísi. Til að ná í eitthvað upp úr streng þurfum við að nota hornklofa (e. square brackets), tákn sem eru eins og kassalaga svigar [og]. Við notum þessi tákn í Python til að ná í gögn upp úr einhverri gagnagrind, sjáum nánari útskýringu á því fyrirbæri í kaflanum 4. Nú lítum við svo á að strengir séu til þess að geyma fyrir okkur tákn í ákveðinni röð og við getum nálgast þessi tákn með því að nota hornklofa. Inn í hornklofann ætlum við að láta þann vísi (eða það sætisnúmer) sem við viljum vinna með. Skoðum aftur sama streng og í kóðabút 3.4 og sjáum hvað er í vísi 1 í þeim streng.

Kóðabútur 3.5: Vísir 1

```
1 strengur = "kex"
2
3 print(strengur[1])
```

```
1 e
```

Eins og sést í kóðabút 3.5 þá vísar vísir númer 1 ekki á fremsta stafinn sem er í þessu tilfelli k heldur vísar hann á stafinn e. Það er vegna þess að í Python og flestum öðrum forritunarmálum (ekki öllum) er byrjað að telja í núll. Þannig að fremsti vísirinn í streng (og öðrum gagnagrindum) er núllti vísirinn. Hver er þá síðasti vísirinn? Nú höfum við komist að þeirri niðurstöðu (í kóðabút 3.4) að strengurinn "kex"hefur þrjú stafabil, að það séu þrír sætisvísar í strengnum, að k sé í vísi 0, e sé í vísi 1 og þá hlýtur x að vera í vísi 2. Síðasti vísirinn í streng er því einum lægri heldur en lengdin á strengnum. Þannig að strengur af lengdinni fimm, eins og strengurinn "texti", hefur fimm stafabil sem eru í vísum númer 0,1,2,3 og 4.

Ítarefni 3.1 Vísar í streng

Strengur af lengd n hefur n tákn augljóslega, en númerin á vísunum fyrir táknin ná frá 0 upp í n-1. Þetta er gömul hefð og er hún hluti af flestum æðri forritunarmálum. Það er þó ekki þannig að tölvan byrji að telja í 0, heldur byrjar hún að geyma röð hluta í minnishólfi og fremsta minnishólfið í röðuðum hlut fær sætisnúmerið 0. Ástæðan er

[1] Vegna þess að strengir eru í ákveðinni röð og af ákveðinni lengd eru þeir *ítranlegir* (e. iterable) sjá nánar í kafla 6

einfaldari útreikningar við að sækja röðuð gögn. Fremsta stakið í röðuðum hlut er þá kallað núllta stakið.

3.3.1 Óbreytanleiki

Nú höfum við séð að það er hægt að sækja stafabil í streng, eins og tildæmi núllt táknið í strengnum. Þá er mikilvægt að hafa í huga að í Python er ekki leyfilegt að endurskilgreina hluta úr streng. Byrjum á því að skoða hvað endurskilgreining þýðir. Ef við búum til breytu eins og í kóðabút 2.6 og notum nafnið á henni aftur til að segja vélinni að endurnýta minnissvæði með ákveðnu nafni. Þá erum við búin að endurskilgreina breytuna okkar, hún var eitthvað áður en nú er hún eitthvað annað.

Þar sem strengir eru með ákveðin númeraða vísa sem benda á ákveðin tákn gætum við þá ekki bara sagt við vélina „mig langar að breyta bara tákni númer 0”? Það er ekki í boði því að í Python eru strengir *óbreytanlegir* (e. immutable) og því er bara hægt að vinna með þá eins og þeir eru eða endurskilgreina þá alveg.

3.3.2 Neikvæðir vísar

Nú höfum við talið frá 0 og upp í n-1, frá vinstri til hægri en það má einnig telja frá hægri til vinstri. Ef okkur langar að vinna með öftustu stökin í streng þurfum við ekki að vita hvað strengurinn er langur, við getum talið frá hægri endanum og unnið með neikvæða vísa. Í því tilfelli byrjum við ekki að telja í 0, því það væri *tvírætt* (e. ambiguous). Tölvan myndi ekki vita hvorn 0 vísinn við værum að biðja um þegar við segðum `strengur[0]`, hvort við værum að tala um núll frá vinstri eða hægri. Þess vegna byrjum við að telja frá hægri í -1, og höldum þannig áfram þar til við erum komin niður í -n þar sem n er lengdin á strengnum. Svo strengurinn "kexęr með vísana 0,1 og 2 en einnig vísana -3, -2 og -1 bæði í þessari röð, svo vísir -1 er alltaf síðasta táknið í streng.

3.3.3 Hlutstrengir

Nú vitum við hvernig á að sækja eitt stakt tákn upp úr streng. En hvernig náum við í einhvern hluta úr honum? Það er einnig gert með hornklofunum og við notum þá með ákveðnum hætti, við fáum að setja inn fleiri upplýsingar heldur en bara hvaða staka vísi við viljum. Þá megum við nýta okkur allt að þrjá *stika* (e. parameters).

Stikarnir okkar eru hvar við viljum byrja að lesa hlutstrenginn okkar, hvar við viljum hætta og hvað við viljum taka stór skref. Þetta er gert með heilum tölum með tvípunktum á milli, sem má sjá í kóðabút 3.6. Í línu 3 eru tveir tvípunktar innan hornklofanna og tölurnar sem koma á milli þeirra er afmörkunin á því hvað við viljum lesa upp úr `strengur`. Nú er vert að nefna að þegar við notum þessa málskipan eru ákveðin gildi sjálfgefin, skoðum hvað það þýðir.

Tökum dæmi um `strengurinn_minn[a:b:c]` þar sem a, b og c eru stikar til að sækja hlutstreng, hvað getur staðið fyrir a, b, og c? Hvað ef við sleppum þeim? Hver eru sjálfgefin gildi þessara stika?

Stikarnir a, b, og c verða að vera:

1. a er vísirinn sem við byrjum fyrir framan, ef þetta væri 0 væri *leshaus* vélarinnar staddur fyrir framan núllta táknið og það yrði lesið næst. Þessi vísir verður að vera lægri en b (annars fæst tómur strengur). Sjálfgefið er að a sé fremsti stafurinn í strengnum.
2. b er vísirinn sem við hættum fyrir framan, þar stoppum við leshausinn og vélin les ekki það tákn. Sjálfgefið er fyrir aftan aftasta stafinn svo síðasta táknið er lesið, sem gerir það að verkum að við þurfum ekki að vita hvað strengur er langur til að geta sótt hann allan.
3. c er skrefastærðin sem er sjálfgefin 1, það er að við skoðum hvert einasta tákn og hoppum ekki yfir neitt stak. Ef c er valið stærra er það fjöldinn tákna sem á að hoppa yfir frá lesstað að næsta tákni.

Kóðabútur 3.6: Hlutstrengir

```
strengur = "kex með smjöri, osti og sultu"

print(strengur[1:8:1])

sami_strengur = strengur[::]

aftan_x = strengur[3::]

kex = strengur[0:3]

sultu = strengur[-5:]

nema_sidasti = strengur[:-1:]

annar_hver = strengur[::2]

ofugur = strengur[::-1]

print(sami_strengur, aftan_x, kex, sultu, nema_sidasti, annar_hver, ofugur)
```

```
ex með
kex með smjöri, osti og sultu  með smjöri, osti og sultu kex sultu kex með smjöri, osti
    og sult kxmðsjr,ot gslu utlus go itso ,iröjms ðem xek
```

Takið eftir að hvar sem engin tala kemur fyrir þegar tvípunktur er notaður þá er setur vélin sjálfgefið gildi í staðinn.

3.4 Strengjaaðferðir

Áður en aðferðir á strengi eru kynntar þarf að útskýra stuttlega hvað aðferðir eru. Við höfum séð `print()` fallið notað, það er innbyggt fall í Python sem prentar það sem beðið er um á staðalúttak. Það að fall sé innbyggt þýðir að nafnið á því er frátekið og hægt er að beita því án þess að beita kóðasafni (sjá kafla 11). Innbyggð föll í Python eru nokkur og koma þau fyrir hér og þar í bókinni, ekki er þörf á að kynna virkni þeirra sérstaklega heldur er gagnlegra að kynna þau til sögunnar jafnóðum eftir því sem við þurfum á þeim að halda.

Í fljótu bragði virkar fall eins og við þekkjum föll úr stærðfræði, það heitir einhverju nafni, eins og `cos`, og tekur við einhverju viðfangi innan sviga, eins og `cos(x)`, og getur skilað einhverri niðurstöðu, sjá má meira um föll í kafla 10. Aðferðir eru sérhæfð föll sem virka á ákveðnar gagnatýpur. Þannig að allar aðferðir eru föll, ekki öll föll eru aðferðir. Á ensku eru aðferðir kallaðar *methods* og föll *functions*. Aðferðir eru í raun „hengdar aftan á" þá týpu sem þær eiga að verka á. Það er gert með því að skrifa nafnið á breytunni sem inniheldur gögnin sem við viljum framkvæma aðferðina á, gera svo punkt, skrifa nafnið á aðferðinni og setja sviga, inn í svigana fara öll þau viðföng sem aðferðin tekur við.

Þetta er í raun fyrstu kynni okkar af hlutbundinni forritun. Strengurinn er hlutur og aðferðin verkar á hlutinn.

Annað sem þarf að hafa í huga áður en við vinnum með aðferðir á strengi er að strengir eru óbreytanlegir (e. immutable) sem þýðir að aðferðir sem eru notaðar á þá *skila* öðrum strengjum í stað þess að breyta strengnum sem við keyrðum aðferðina á. Með það í huga skulum við skoða eftirfarandi lista af aðferðum sem áhugavert er að taka fyrir.

Hér koma fyrir nokkrar aðferðir, gerum ráð fyrir að þær séu að verka á breytuna strengur sem inniheldur táknin "valborg Sturludóttir".

- `strengur.capitalize()` skilar strengnum "Valborg sturludóttir"þar sem fremsti táknið er nú hástafur.
- `strengur.upper()` skilar strengnum "VALBORG STURLUDÓTTIR"þar sem allir stafir eru nú háfstafir.
- `strengur.lower()` skilar strengnum "valborg sturludóttir"þar sem allir stafir eru nú lágstafir.
- `strengur.switchcase()` skilar strengnum "VALBORG sTURLUDÓTTIR"þar sem búið er að skipta út lágstöfum fyrir hástafi og öfugt.
- `strengur.index('v')` skilar tölunni 0 þar sem fyrsta 'v' táknið kemur fyrir í vísi 0
- `strengur.index('x')` skilar villu þar sem táknið 'x' kemur ekki fyrir í strengnum
- `strengur.find('v')` skilar tölunni 0 þar sem fyrsta 'v' táknið kemur fyrir í vísi 0
- `strengur.find('x')` skilar tölunni -1 þar sem 'x' finnst ekki í stregnum.

Takið eftir því að þarna er orðið lykilorðið „skilar", sem þýðir að við fáum í hendurnar eitthvað til að vinna með sem við getum t.d. vistað í breytu, við skoðum þetta nánar þegar við gerum okkar eigin föll í kafla 10. Það er þörf á því að vinna með aðferðir á strengi með þessum hætti því að við munum að strengir eru óbreytanlegir. Þannig að ef við viljum vinna með einhverja útkomu byggða á streng þá þurfum við að fá útkomuna í hendurnar, því strengurinn sem aðferðinni var beitt á breytist ekki neitt við að kalla í aðferðina. Í upptalningunni hér að ofan getum við keyrt allar þessar línur í röð eins og kóða og búist við að fá þessi svör því að breytan strengur verður aldrei fyrir neinum breytingum, hún helst sem "valborg Sturludóttir"þrátt fyrir að við köllum í alla þessa fylkingu af aðferðum.

Þar sem það sem strengjaaðferðirnar skila flestar eru strengir má setja hverja aðferðina á eftir annarri, eins og `"Valborg".upper().lower().swapcase().capitalize()`. Þessi aðgerðarsúpa er tiltölulega vitlaus en leyfileg, það sem er að gerast er að fyrst er `upper` keyrt og svo er `lower` keyrt á það sem upper skilaði, og svo koll af kolli. Einnig má þarna á milli ná í hlutstreng og gera t.d. `"Valborg"[0:3].lower()`. Svona vinnur vélin sig frá vinstri til hægri svo lengi sem að það sem skilast vinstra megin sé eitthvað sem er löglegt að beita hægri hliðinni á. Dæmi um ólöglegt væri `"Valborg".index('b').upper()` þar sem .index() skilar heiltölu og á þær er ekki hægt að beita aðferinni .upper().

Í þessari bók verða ekki gerð skil á öllum þeim aðferðum sem eru í boði fyrir þær týpur sem við skoðum. Þær eru mýmargar og til ýmiss gagnlegar en það er út fyrir svið þessarar bókar að taka hverja fyrir sig fyrir og því munum við bara nefna þær sem gagnast okkur að skoða.

Gerið ítarlegar tilraunir. Ekki lesa þennan undirkafla bara, gerið ykkar eigin prófanir og áttið ykkur á því hvernig þetta hangir saman. Prófið allavega þangað til að þið sjáið eftirfarandi skilaboð:

1. <function str.upper()>
2. TypeError: index() takes at least 1 argument (0 given)
3. TypeError: must be str, not int (á index() aðferðinni)
4. TypeError: upper() takes no arguments (1 given)

Fyrsta villan þarna gefur ekki villu því að það er lögleg skipun að spyrja „hvað er þetta?" án þess að kalla í aðferðina til að nota hana. Hvernig náðuð þið að gera það?

Að því sögðu ætlum við að skoða eina strengjaaðferð sérstaklega `.format()` sem tekur við eins mörgum viðföngum og við viljum setja inn í einhvern annan streng sem inniheldur jafn marga slaugusviga, {}, og við viljum setja inn í staðinn fyrir.

Kóðabútur 3.7: Aðferðin .format() kynnt

```
1 strengur = "kex með {}, {} og {}"
2 matur = strengur.format("avókadó", "majónesi", "eggi")
3 print(matur)
```

```
1 kex með avókadó, majónesi og eggi
```

3.5 Æfingar

Æfing 3.1 Búðu til breytu sem inniheldur streng ■

Æfing 3.2 Búðu til breytu sem inniheldur streng, búðu til aðra breytu sem geymir fremsta stafinn úr þeirri breytu. ■

Æfing 3.3 Notaðu innbyggt fall til þess að finna lengdina á strengnum "halló góðan daginn í dag". ■

Æfing 3.4 Notaðu innbyggt fall til þess prenta út þann staf sem er í þriðja stæði í strengnum 'kex!'. ■

Æfing 3.5 Notaðu innbyggt fall til þess finna lengdina á strengnum 'kex!'. ■

Æfing 3.6 Notaðu heiltöludeilingu til að prenta út þann staf sem er í miðju strengsins "allra handa", sem er strengur af lengd 11 og því skilgreiningaratriði hvort stakið í stæði 4 eða 5 sé í miðjunni. Hvort kemur tómt bil eða stafurinn a? ■

Æfing 3.7 Búðu til tvær breytur sem innihalda strengi, búðu svo til þriðju breytuna sem inniheldur samskeytingu af þessum tveimur breytum. Lengdu þriðju breytuna, þannig að samskeytingin sé endurtekin að minnsta kosti tvisvar sinnum. ■

Æfing 3.8 Gefum okkur að til séu tveir strengir, n1 og n2, þeir innihalda fyrsta nafnið þitt og eftirnafn þitt. Gefum okkur einnig að þeir séu ekki rétt ritaðir skv. íslenskum ritunarreglum. Hvernig setjum við þá saman í einn streng eftir að hafa beitt á þá aðferðum til að þeir séu örugglega með fyrsta staf stóran og alla aðra litla? ■

Æfing 3.9 Nú gerum við ráð fyrir að vera með streng í höndunum sem er geymdur í breytunni lykill. Strengurinn á að verða sterkt lykilorð og við viljum rugla hann töluvert til þess að hann verði ekki einfalt orð sem auðvelt er að giska á. Til þess ætlum við að gera eftirfarandi:

1. Búa til annan streng sem inniheldur einhver tákn (tölur, bókstafi og önnur tákn)
2. Búa til breytu sem inniheldur lengdina á tákna strengnum og aðra sem inniheldur lengdina á lykilstrengnum
3. Búta upp lykil strenginn (sama sætisnúmer má nota oftar en einu sinni en öll þurfa að vera til staðar), skeyta við hvern bút einhverjum bút úr táknastrengnum. Í þessu skrefi má lengja bútana og/eða beyta strengjaaðferðum á þá og endurskilgreina lykil strenginn sem þessa breytingu.
4. Að lokum á að snúa strengnum við, þannig að hann sé afturábak.

■

4. Listar

Listar eru gagnagrindur, sem þýðir að þeir geta geymt fyrir okkur hin ýmsu gögn og gert okkur þau aðgengileg á ákveðinn máta. Listar eru skilgreindir með hornklofum [ ] og er lykilorðið þeirra **list**.

4.1 Listar skilgreindir

Listar geyma, í ákveðinni röð (eins og strengir), þau gögn sem við viljum geta notað, þau mega vera af hvaða týpu sem er. Gögnin sem eru sett inn í listann eru kölluð stök og röðin sem þau eru í eru aðgengileg eftir vísum eða sætisnúmerum alveg eins og tákn í strengjum. Stökin eru aðgreind með kommum. Þær týpur sem við höfum séð hingað til eru heiltölur, fleytitölur, strengir og listar. Allt eru þetta möguleg stök í lista.

Kóðabútur 4.1: Listar skilgreindir

```
1  listinn_minn = []
2
3  talna_listi = [1, 2, -3000, 4.8, -3.14, 9]
4  strengja_listi = ["halló", "bless", "11", "6"]
5
6  talan_null = talna_listi[0]
7  strengur_eitt = strengja_listi[1]
8  listi_af_strengjunum = strengja_listi[1:3]
9
10 nyr_listi = ["núllta stakið", 1, 2, 3.0, "fjórða stakið", [5]]
11 talan_fimm = nyr_listi[5][0]
12
13 print(listinn_minn,talan_null,strengur_eitt,listi_af_strengjunum,talan_fimm)
```

```
1  [] 1 bless ['bless', '11'] 5
```

Í kóðabút 4.1 sjáum við fimm lista skilgreinda, sá fyrsti inniheldur ekkert stak og er tómur listi, næstu tveir innihalda einsleit gögn, sá fjórði er búinn til sem sneið (e. slice) og sá fimmti inniheldur fjölbreytt gögn. Breytan í línu 6 fær gildið 1 sem er heiltala, breytan í línu 7 verður strengurinn `bless` og breytan í línu 8 verður listi sem inniheldur strengina `bless` og `11`.

Við sjáum einnig að við erum með 6 stök í listanum nyr_listi sem er skilgreindur í línu 10. Fremsta stakið er strengur, næstu þrjú eru tölur, síðan kemur annar strengur og síðasta stakið í sæti 5 er listi. Sá listi inniheldur eitt stak sem er þá í núllta vísi í þessum innri lista. Það sem sést svo í línu 11 er keðjun (e. chain) hornklofa, þannig að hornklofa er beitt á það sem fyrri hornklofinn skilaði. Þetta er eins og að skeyta einni strengjaaðferð fyrir aftan aðra eins og við sáum í lok síðasta kafla. Það sem gerist er að fyrst skoðar vélin hvað er í 5 sæti í breytunni `nyr_listi`, sem er listinn `[5]`, þá nær vélin í það sem er í núllta sæti í þeim lista sem er heiltalan 5.

Þetta getum við svo sannreynt með því að skoða úttakið og gera okkar eigin tilraunir.

Ef við hugsum okkur töflureikni eins og Calc eða Excel þá getum við ímyndað okkur að ein lína þar sé eins og einn listi hér, að hver dálkur þar innihaldi gögn sem væri stak í listanum hér. Þá getum við líka ímyndað okkur að ef við erum með margar raðir séu þær geymdar á einni örk eða einu skjali. Sjáum hvernig það myndi líta út í kóðabút 4.2 þar sem við viljum halda utan um starfsfólk í fyrirtæki. Ef við ættum skjal í töflureikni sem héldi utan um allt starfsfólk í fyrirtæki gæti hausinn á því litið svona út: Nafn Tölvupóstur Deild Símanúmer

Svo er hver röð fyrir neðan það útfyllt með upplýsingum um einhvað tiltekið starfsman.

Kóðabútur 4.2: Listar af listum

```
1 starfsfolk = [["Jóna Jónsdóttir", "jona@fyrirtaeki.is", "Póstur", "4445555"],
2               ["Kristján Kristjánsson", "kristjan@fyrirtaki.is","Laun","4445589"],
3               ["Halldóra Halldórudóttir", "halldora@fyrirtaeki.is", "Skrifstofa", "4445500"]]
```

Við tökum eftir því að listinn `starfsfolk` í kóðabút 4.2 inniheldur þrjá aðra lista, og þeir eru aðgreindir með kommum alveg eins og stökin inni í hverjum innri lista fyrir sig eru líka aðgreind með kommum. Einnig tökum við eftir því að hér sjáum við í fyrsta sinn inndrátt, það er í raun bara aukalegt bil sem vélin hunsar við að skilgreina breytuna starfsfolk og auðveldar það okkur að lesa kóðann. Þetta er ekki eins og inndrátturinn sem við munum sjá og beita í næsta kafla, Segðir, skilyrðissetningar og sanngildi. Takið eftir því að gögnin eru einsleit, að fremsta stakið í öllum innri listum er af sömu týpu og svo koll af kolli. Þetta auðveldar gagnavinnslu því að við getum gert ráð fyrir því að lína númer 10.000 líti eins út án þess að þurfa að skoða hana.

4.2 Að vinna með gögn

Þegar við geymum gögn viljum við að þau séu aðgengileg og að við getum skoðað þau, breytt þeim og unnið með á máta sem hentar okkur. Listar gera okkur kleift að nálgast gögn eftir sætisvísum, við eigum eftir að sjá gagnagrindur sem geyma stökin á annan máta. Við náum í gögn upp úr lista eftir sætisvísi alveg eins og við sóttum tiltekið tákn úr streng, með því að nota hornklofa og það vísa sem við vildum. Sætisvísar eru frá 0 upp í lengdina á listanum að einum frádregnum, svo ef það eru þrjú stök í listanum eins og í kóðabút 4.2, þá er listinn af lengd 3 og vísarnir í honum er 0,1 og 2. Einnig megum við nota neikvæða vísa, eins og í strengjum, þar sem síðasta stakið er í vísi -1 og fremsta stakið er í vísi sem er jafn neikvæðri lengd listans.

Hér þurfum við að athuga að við viljum ekki ruglast á því að skilgreina lista með hornklofum og að sækja gögn úr lista eða streng með hornklofum. Í fyrra tilfellinu standa hornklofarnir einir og sér, þar sem við erum að skilgreina nýjan lista. Í seinna tilfellinu standa hornklofarnir fyrir aftan þá breytu sem á að sækja gögn upp úr með ákveðnum sætisvísum. Sjáum hvernig við getum fengið upplýsingar sem eru skráðar um tiltekið starfsman úr listanum.

Kóðabútur 4.3: Unnið með gögn úr lista

```
1 print(starfsfolk[0])
2 print(starfsfolk[0][0])
3 print(starfsfolk[0][1][4])
```

```
['Jóna Jónsdóttir', 'jona@fyrirtaeki.is', 'Póstur', '4445555']
Jóna Jónsdóttir
@
```

4.2.1 Listar eru breytanlegir

Nú allt í einu munum við að Jóna er ekki Jónsdóttir heldur Alfreðsdóttir og við þurfum að laga það, við þurfum ekki að skilgreina listann allann upp á nýtt (sem við hefðum þurft að gera ef við værum með streng) heldur þurfum við bara að setja nýtt gildi inn fyrir það sem heldur utan um nafnið hennar Jónu. Við vitum að nafnið hennar er í listanum okkar sem heldur utan um starfsfólk, við vitum að hún er í núllta innri listanum og að nafnið hennar er núllta stakið í þeim lista, við sáum það í kóðabút 4.3. Þá það sem við gerum er að endurskilgreina þann stað í listanum í stað þess að endurskilgreina allan listann. Hugsið þetta eins og 100.000.000 línur í gagnagrunni, væri ekki þægilegt að geta breytt bara einni línu í stað þess að þurfa að gera afrit af öllum grunninum til þess að breyta einu litlu nafni?

Kóðabútur 4.4: Unnið með gögn úr lista

```
starfsfolk[0][0] = "Jóna Alfreðsdóttir"
print(starfsfolk)
```

```
[['Jóna Alfreðsdóttir', 'jona@fyrirtaeki.is', 'Póstur', '4445555'], ['Kristján
    Kristjánsson', 'kristjan@fyrirtaki.is', 'Laun', '4445589'], ['Halldóra
    Halldórudóttir', 'halldora@fyrirtaeki.is', 'Skrifstofa', '4445500']]
```

4.3 Gagnlegar aðferðir á lista

Eins og tekið var fram í kaflanum um strengi þá er ekki ætlunin að fara yfir allar þær innbyggðu aðferðir sem til eru fyrir lista heldur draga fram nokkrar sem eru mjög gagnlegar til að auka skilning á notkun á aðferðum.

Gefum okkur að við eigum listann `[0,2,1,3]` sem er geymdur í breytunni `listinn_minn`, við gefum okkur einnig að aðferðir séu keyrðar á hann án þess að aðferðin á undan hafi breytt honum neitt.

- **pop** virkar eins og við séum með stafla af diskum og við poppum einum disknum af. `listinn_minn.pop()`
  - það sem þetta gerir er að breyta listanum og skila staki.
  - gildið sem það skilar er aftasta stakið úr listinn_minn.
  - 3 er gildið sem það skilar í okkar tilfelli svo listinn_minn verður að [0,2,1].
  - hægt er að geyma það með því að gera x = listinn_minn.pop() og þá inniheldur x töluna 3.
  - einnig er hægt að setja inn sætisnúmer sem viðfang og þá er stakið í því sæti fjarlægt og listinn dregst saman, sjá kóðabút 4.5.
- **append** þýðir að skeyta aftan við og það er nákvæmlega það sem aðferðin gerir. `listinn_minn.append(x)`
  - það sem þetta gerir er að breyta listanum þannig að búið er að bæta breytunni x aftast í listann.
  - þessi aðferð skilar engu til baka til okkar svo það er ekkert vit í því að skrifa `listi = listinn_minn.append(4)`
  - segjum að x hafi verið stillt sem talan 4 þá lítur listinn núna svona út [0,2,1,3,4].
  - þessi aðferð verður að fá eitt viðfang og nákvæmlega eitt viðfang, sem er af hvaða gagnatýpu sem er, svo við gætum sett inn einn lista sem inniheldur 100.000 stök en það

er nákvæmlega einn listi.
  - sjá notkun í kóðabút 4.6
- **sort** þýðir að raða og þessi aðferð raðar listanum ef það er mögulegt. `listinn_minn.sort()`
  - það sem þetta gerir er að raða listanum í röð með samanburðarvirkjum (þeir verða kynntir í kafla Segðir, skilyrðissetningar og sanngildi), og stökin í listanum þurfa þá að vera samanburðarhæf.
  - aðferðin raðar listanum í röð frá lægsta gildi til hæsta gildis, það er okkur tamt þegar við skoðum talna lista en í því tilfelli að listinn innihaldi bara strengi þýðir það að listanum er raðað í stafrófsröð sem er skilgreind eftir því táknakerfi sem Python notar.
  - listinn_minn.sort() myndi gera það að verkum að hann sé nú geymdur sem [0,1,2,3].
  - aðferðin skilar engu svo það er ekkert vit í því að gera `x = listinn_minn.sort()`
  - sjá notkun í kóðabút 4.7.

Kóðabútur 4.5: .pop() aðferðin

```
1 test = [1,2,3]
2
3 x = test.pop()
4 y = test.pop(0)
5 print(x, y)
6 print(test)
```

```
1 3 1
2 [2]
```

Kóðabútur 4.6: .append() aðferðin

```
1 test = []
2
3 test.append(1)
4 test.append("nú bætum við streng aftast í listann")
5 test.append(["hér er heill listi", "með nokkrum stökum", "en hann er samt einn stakur
      listi", "og telst því sem að bæta við einu staki"])
6 test[2].append("hér var bætt aftast í innri listann, ekki er komið nýtt stak í test")
7
8 print(test)
```

```
1 [1, 'nú bætum við streng aftast í listann', ['hér er heill listi', 'með nokkrum stökum',
      'en hann er samt einn stakur listi', 'og telst því sem að bæta við einu staki', 'hér
      var bætt aftast í innri listann, ekki er komið nýtt stak í test']]
```

Sjáum að í línu 4 í kóðabút 4.5 þá er einhver tala sett inn í aðferðina, sem segir til um sætisnúmerið sem við viljum fjarlægja en í línunni á undan þá var það aftasta stakið. Athugum einnig úttakið að listinn `test` hefur snarminnkað. Getiði núna fjarlægt stak í stæði 1? Af hverju ekki?

Athugum í kóðabút 4.6 er verið að bæta aftan í lista, í línu 5 er heilum lista bætt við og í línu 6 er bætt við þann lista. Skoðið þetta og prófið ykkur áfram með það. Hvað gerist ef þið setjið tvo strengi inn sem viðfang með kommu á milli? Getiði sett inn streng sem er með strengjaaðferð hangandi á sér inn í svigana?

Þá síðast skoðum við að raða, raða má einsleitum eða sambærilegum stökum. Ef listinn inniheldur innri lista er raðað eftir fremsta, núllta, staki hvers lista. Gerið nú tilraun á þessu með því að setja inn gögn af mismunandi týpum inn í lista og raða svo, eða breyta `starfsfolk` listanum þannig að fremsta stakið sé einhversstaðar tala en annarsstaðar strengur. Sjáið hvaða villu þið fáið.

Kóðabútur 4.7: .sort() aðferðin

```
test = [1,6,3,1]
test.sort()
print(test)

test = ["b", "a", "m", "z"]
test.sort()
print(test)

starfsfolk = [["Jóna Jónsdóttir", "jona@fyrirtaeki.is", "Póstur", "4445555"],
            ["Kristján Kristjánsson", "kristjan@fyrirtaki.is","Laun","4445589"],
            ["Halldóra Halldórudóttir", "halldora@fyrirtaeki", "Skrifstofa", "4445500"]]
starfsfolk.sort()
print(starfsfolk)
```

```
[1, 1, 3, 6]
['a', 'b', 'm', 'z']
[['Halldóra Halldórudóttir', 'halldora@fyrirtaeki', 'Skrifstofa', '4445500'], ['Jóna
    Jónsdóttir', 'jona@fyrirtaeki.is', 'Póstur', '4445555'], ['Kristján Kristjánsson',
    'kristjan@fyrirtaki.is', 'Laun', '4445589']]
```

4.4 Æfingar

Æfing 4.1 Búðu til breytu sem inniheldur lista ■

Æfing 4.2 Verkefnið er tvíþætt:

1. Búðu til lista sem inniheldur 4 stök sem öll eru af mismunandi týpum.
2. Vitandi hvar strengurinn er í listanum, skaltu svo breyta stakinu í listanum sem inniheldur strenginn og setja einhvern annan streng í staðinn.

■

Æfing 4.3 Gefin er eftirfarandi kóði. Það sem við viljum gera er að fletta upp heimavist og netfangi nemanda 1 og 2. Við viljum búa til strengjabreytu sem inniheldur þessi gögn fyrir hvorn nemanda fyrir sig. Án þess að vita hvernig nem1 og nem2 breyturnar líta út (eins og við höfum fengið þær gefnar að nafninu til og sjáum ekki hvað þær geyma) við fáum að vita að breyturnar séu listar sem séu eins uppbyggðir og header listinn.

Við þurfum því að beita index aðferðinni til að finna gögnin.

```
1 header = ["nemandi", "sími", "heimavist", "netfang", "lykilorð", "áfangar"]
2 nem1 = ["Valborg", "9999999", "vestur", "valborg@netfang.is", "best_practice",
      "FORR2**"]
3 nem2 = ["Sturludóttir", '00000000', 'austur', "valborg@example.com", "1234", "FORR1**"]
```

■

Æfing 4.4 Gefinn er eftirfarandi kóðabútur, náðu í strenginn "valli"þannig að hann sé geymdur í breytu og type fallið af breytunni skili niðurstöðunni str. Náðu einnig í töluna 0 innan listans og geymdu í breytu þannig að type fallið af henni skilið int.

```
1 nested_list = [[[[0],1],2],["hvar"],["er"],[["valli"],"?"]]
```

■

Æfing 4.5 Hvers vegna prentast hér **tómur** listi?

```
1 listed = ["stak í núllta stæði", "fyrsta", "öðru", "þriðja", "fjórða"]
2 print(listed[2:2])
```

■

Æfing 4.6 Leystu eftirfarandi verkefni í röð:

1. Búðu til lista sem inniheldur nokkur nöfn, þetta á að vera tengiliðalisti.
2. Raðaðu listanum í stafrófsröð, eins og Python gefur kost á fyrir íslensku.
3. Það gleymdist að setja Agnesi í listann, hvernig bætirðu henni við aftast í listann?
4. Það kemur alveg hrikalega út að hafa Agnesi aftast eftir að hafa raðað listanum, raðaðu listanum aftur í stafrófsröð
5. Æ, nú er listinn orðinn of langur, nú skaltu fjarlægja eitthvað nafn úr listanum, þó ekki það aftasta.

■

Æfing 4.7 Leystu eftirfarandi verkefni í röð:

1. Búðu til lista sem inniheldur nokkra innri lista, innri listarnir eru upplýsingar um gæludýr

(nafn, aldur og tegund).
2. Kemur í ljós að Askja, 13 ára gömul border collie tík gleymdist, bættu henni í listann.
3. Askja á sér það áhugamál að elta laufblöð, bættu því aftast í listann sem inniheldur gögnin um Öskju.
4. Nú skaltu raða ytri listanum.
5. Kemur í ljós að það hentar ekki að geyma gögn um áhugamálið hennar Öskju, fjarlægðu þau gögn úr listanum um Öskju.
6. Æ, nú er Askja dáin og það þarf að fjarlægja hana úr ytri listanum.

■

5. Segðir, skilyrðissetningar og sanngildi

Kóða má skipta í *segðir* (e. expressions) og *yrðingar* (e. statements). Segð er eitthvað sem krefst svars, „er rigning?”. Svarið er metið sem gildi og við fáum útkomu. Yrðing er eitthvað sem er sett fram sem staðreynd „það er rigning”.

Núna ætlum við að velta fyrir okkur segðum, í þessum kafla ætlum við að einbeita okkur að því að meta útkomu og fá í hendurnar svör sem við getum svo gert eitthvað við.

Til þess að gera það þurfum við að læra á nýja týpu sem heitir `boolean` og hefur lykilorðið **bool**, boolean gildi eru kölluð sanngildi (stundum búlsk gildi). Boolean týpan er frábrugðin þeim týpum sem við höfum séð hingað til því að það eru eingöngu tvö möguleg gildi sem Boolean getur verið, **True** og **False** sem þýðast sem 1 og 0, satt og ósatt. Þau eru upprunin úr búlískri algebru [1] (e. Boolean algebra). Nú er það flestum kunnug staðreynd að tölvur vinna með 0 og 1 í grunninn, en hvernig það er notað í almennri forritun í æðri forritunarmálum er viðfangsefnið okkar núna.

Í þessum kafla verður farið yfir sanngildi, *samanburð* (e. comparison) og *samanburðarvirkja* (e. comparison operators), *rökvirkja* (e. logical operators) og svo *skilyrðissetningar* (e. conditional statements). Það er mikilvægt að ná góðum tökum á þessum kafla ef halda á lengra inn í námsefnið, ef ekki er skilingur fyrir hendi á því hvernig segðir virka eða hvernig á að setja upp skilyrðissetningu er erfitt að ætla að halda mikið áfram. Því er gott að gefa sér nægan tíma í þetta efni og gera ítarlegar tilraunir.

5.1 Sanngildi

Eins og kom fram í inngangi kaflans eru sanngildi einungis tvö, True og False. Hægt er að geyma þau í breytum eins og gögn af öðrum týpum sem við höfum séð. Sanngildi eru einnig metin sem 1 eða 0, fyrir True annars vegar og False hinsevegar.

Vitandi að gildin geta verið 0 eða 1 (aldrei bæði í einu) þá er þess virði að nefna hérna sanntöflur. Látum p vera yrðinguna „það er rigning” og látum q vera yrðinguna „mér er kalt”. Þá gætum við,

[1] Ekki verður farið yfir búlíska algebru af neinu ráði í þessari bók en þau fræði eru gífurlega góður grunnur til að skilja betur hvernig segðir og rökvirkjar virka, endilega kíkið á ensku Wikipedia síðuna.

Tafla 5.1: Sanntafla

p	q	s1	s2
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

með því að skoða mismunandi aðstæður, fengið rökrétt svar við t.d. spurningunni „er rigning og er mér kalt?" sem við getum skrifað sem spurning1 (eða s1) og svo annarri spurningu sem er „er rigning eða er mér kalt?" sem við getum kallað spurning2 (eða s2).

tta ótrúlega
láss sem er á
xtans og töflunn-

Ef við horfum á töflu 5.1 þá sjáum við að yrðingarnar okkar um rigningu og kulda eru uppsettar þannig að hver lína í töflunni er einstakt ástand, og allar mögulegar samsetningar koma fram [2]. Báðar yrðingar eru ósannar í fyrstu línunni, svo eru þær sannar sitt á hvað, og í fjórðu línu eru þær báðar sannar. Þá eru dálkarnir fyrir s1 og s2 svörin við spurningunum hér að ofan miðað við sanngildi yrðinganna í þeim tilteknu aðstæðum. Í þeim aðstæðum þar sem er hvorki rigning né mér er kalt þá er svarið við báðum spurningum einnig neitandi (0). Í þeim aðstæðum þar sem er bæði rigning og mér er kalt þá er svarið við báðum spurningum játandi (1). Þannig að til þess að svarið við spurningu 1 sé játandi þá þarf mér bæði að vera kalt og það þarf að vera rigning, svo þegar yrðingarnar eru ekki sannar á sama tíma þá skiptir ekki máli hvor sé sönn því að önnur er ósönn og því er svarið neitandi. En spurning 2 er þannig orðuð að það sé nóg að annað hvort sé mér kalt eða það sé rigning úti til þess að svarið sé játandi, svo þegar yrðingarnar eru sannar á víxl þá er svarið alltaf játandi.

Kóðabútur 5.1: Sanngildi geymd sem breytur

```
1 satt = True
2 strengur = "True"
3 osatt = False
```

Akkúrat núna þurfum við bara að vita að týpan Boolean sé til og hvernig eigi að nota hana, með hástaf fremst. Sjáum svo í seinni köflum hvernig hún gagnast okkur.

5.2 Segðir

Eins og kom fram í inngangi kaflans má líta svo á að segðir séu sá hluti af kóðans sem er metinn sem eitthvað gildi, eins og 4 + 5 er segð en x = 5 er yrðing. Nú ætlum við þó að einblína á búlskar segðir, horfa á spurningar sem hafa svar sem er annað hvort satt eða ósatt. Er rigning? Þá horfum við út og sjáum að miðað við aðstæður þá er svarið annað hvort satt eða ósatt og það breytist eftir því hvenær við horfum.

5.2.1 Samanburður

Hvað er samanburður? Það er þegar eitthvað er metið miðað við eitthvað annað, eins og er þetta stærra en hitt? Er þetta þyngra? Er þetta jafngilt? Athugið hér að aðalatriðið er að bera saman eitthvað tvennt, ekki er hægt að nota samanburð nema vera með tvennt í höndunum. Þið hafið

[2] Fjöldi lína í sanntöflu byggir á fjölda yrðinga sem á að skoða. Ef það er bara ein yrðing þá er fjöldi lína 2, það er satt eða ósatt. Fjöldi lína er 2 í veldi fjölda staðhæfinga, eins og í töflu 5.1 þá eru yrðingarnar tvær svo línurnar eru 2^2, og ef þær væru þrjár þá væri línufjöldinn 2^3 og svo framvegis.

kannski lent á spjalli við barn sem segir „er ég stærri?”, stærri en hvað er ekki ljóst og við eigum því erfitt með að svara spurningunni.

Nú þurfum við nýtt hugtak, við erum búin að kynnast reiknivirkjum eins og + og - í kafla 2.2. Nýja hugtakið okkar eru **samanburðarvirkjar**. Samanburðarvirkjar eru notaðir til að spyrja hvort að ákveðin tengsl gilda á milli einhverja tveggja hluta. Eins og í daglegu tali þegar við segjum „er þetta epli stærra en þessi appelsína?” og erum þannig að bera saman epli og appelsínur, samanburðarvirkjar eru til þess að gera slíka setningu formlega svo að tölva geti svarað spurningunni.

Samanburðarvirkjar eru nokkir í Python:

== þá er spurt hvort að hlutirnir sitt hvoru megin við virkjann séu jafngildir

!= þá er spurt hvort að hlutirnir sitt hvoru megin við virkjann séu ólíkir

< þá er spurt hvort að það sem er vinstra megin sé strangt minna en það sem hægra megin (3 er ekki minna en 3 t.d.)

> þá er spurt hvort að það sem er vinstra megin sé strangt stærra en það sem er hægra megin

<= þá er spurt hvort að það sem er vinstra megin sé minna eða jafnt því sem er hægra megin

>= þá er spurt hvort að það sem er vinstra megin sé stærra eða jafnt því sem er hægra megin

Skoðum kóðabút þar sem þessir samanburðarvirkjar eru nýttir til þess annars vegar að fá niðurstöður með tölur og hinsvegar strengi.

Kóðabútur 5.2: Samanburðarvirkjar

```
strengur1 = "abc"
strengur2 = "bcd"
strengur3 = "3"
tala1 = 3
tala2 = 3.0
tala3 = 4
print('jafngildissamanburður')
print(tala1 == tala3)
print(strengur1 == strengur2)
print(tala1 == tala2)
print(strengur3 == tala1)

print('minna en')
print(strengur1 < strengur2)

print('minna eða jafnt')
print(tala1 <= tala2)
```

```
jafngildissamanburður
False
False
True
False
minna en
True
minna eða jafnt
True
```

Í kóðabút 5.2 er ekki verið að nota alla samanburðarvirkjana heldur einungis sýna hvernig er hægt að prófa sig áfram með þá.

5.2.2 Rökvirkjar

Rökvirkjar (e. logical operators) í Python eru þrír, þeir eru **og**, **eða** og **ekki** táknað með `and`, `or` og `not`. Nöfnin þeirra eru lykilorð í Python eins og nöfnin á týpunum sem við höfum séð (**str**, **int**,

float, **list**) en rökvirkjar eru ekki gögn af einhverri týpu heldur eru meira eins og reiknivirkjarnir (+, -, *, **, //, %). Það sem þessir virkjar gera fyrir okkur er að taka tvær búlskar segðir og egja okkur eitthvað um samsetningu þeirra. Tökum dæmi; „Kaffið er heitt og það eru til sítrónur.” Hægt er að meta hvort að kaffið sé heitt eða ekki, og fá þannig út sanngildi fyrir þá segð, það er hægt að gera það sama fyrir segðina um sítrónurnar. En tökum eftir að á milli þessara tveggja segða er rökvirkinn *og*, sem segir okkur að til þess að meta gildi allrar setningarinnar þurfa báðar segðirnar sitthvoru megin við rökvirkjann að vera sannar til þess að setningin í heild sinni skili sönnu annars er hún ósönn.

and til þess að segð með þessum rökvirkja sé sönn þurfa báðar hliðar að vera sannar, annars er hún ósönn
- Það má líta á *og* rökvirkjann eins og margföldun, hann hefur forgang umfram eða.
- Þar sem satt er 1 og ósatt 0 þá ef við margöldum með 0 fáum við alltaf 0 út.
- „það er heitt úti” og „það er kalt úti” myndi skila okkur ósönnu því ekki getur bæði verið satt.
- „það er heitt úti” og „klukkan er fimm” myndi skila okkur sönnu eftir aðstæðum.

or til þess að segð með þessum rökvirkja sé sönn þarf önnur hvor hliðin að vera sönn, annars er hún ósönn.
- Það má líta á *eða* rökvirkjann eins og samlagningu.
- Þar sem satt er 1 og ósatt 0, þá þurfum við bara að sjá 1 einu sinni til þess að útkoman í heild sinni verði sönn.
- „það er heitt úti” eða „það er kalt úti” myndi skila okkur sönnu ef þetta væru þau einu tvö hitastig sem væru í boði.
- „það er heitt úti” eða „klukkan er fimm” myndi skila sönnu eftir aðstæðum.

not snýr við sanngildi segðar, not er ekki sett á milli segða heldur fyrir framan eins segð.
- Það má líta á rökvirkjan *ekki* eins og mínus, hann snýr við sanngildi eins og formerki
- Ekki satt yrði ósatt, ekki ósatt yrði satt.
- **ekki** „það er heitt úti” yrði að yrðingunni „það er ekki heitt úti”.

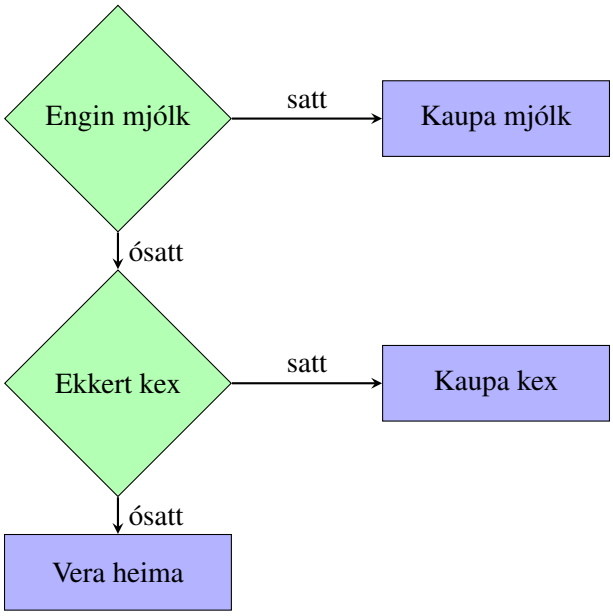
Ítarefni 5.1 Rökvirkjar sem reikniaðgerðir
Til að halda áfram með þessa samlíkingu með margföldun, samlagningu og mínus skulum við skoða eftirfarandi reikningsdæmi: 1·1·1·0 + 1·0 + (-1). Hér gerum við ráð fyrir að hver hluti af þessu reikningsdæmi sé yrðing sem búið er að meta sem sanna eða ósanna eftir þeim aðstæðum sem við erum í (kaffið er heitt, það er kalt úti og þess háttar). Þegar við reiknum þetta dæmi sjáum við að margfaldað er með 0 í báðum þáttunum þar sem margföldun kemur fyrir svo útkoman í hvorum fyrir sig ætti að vera núll. Þessi síðasti liður er okkur ekki eins eðlislægur en við munum að það eru bara til 0 eða 1 og mínus skiptir gildinu okkar svo við hljótum að enda með 0. Þannig að við endum í 0 + 0 + 0 sem gefur okkur 0 og því er öll segðin metin sem ósönn.

5.3 Skilyrðissetningar

Nú viljum við vita til hvers í ósköpunum við vorum eiginlega að leggja það á okkur að skilja hvenær eitthvað er satt eða ósatt. Það er einmitt heilmikið tölvunarfræðilegt gagn í því að geta spurt svona já eða nei spurninga sem tölvan getur svarað. Til dæmis viljum við geta framkvæmt einhverja aðgerð í forritinu okkar **ef** einhver skilyrði eru fyrir hendi. Segjum að við séum með vekjaraklukku sem við forritum til að hringja þegar klukkan er orðin 8. Þá viljum við geta spurt tölvuna hvort að það sé satt eða ósatt að klukkan sé orðin 8. Ef klukkan er ekki orðin 8 viljum við ekki gera neitt, en

ef hún er orðin átta þá viljum við að hún spili einhvern hljóm eða titri. Við gætum líka verið að forrita einfaldan tölvuleik eins og hengimann, ef spilarinn er ekki búinn að giska á alla stafina í orðinu okkar viljum við geta beðið viðkomandi að spyrja aftur. **Annars** viljum við að notandinn fái verðlaun fyrir að hafa giskað á rétt orð [3]. Einnig gætum við viljað gera eitthvað tiltekið þá og því aðeins að eitthvað annað var ósatt. Ef við notum okkar eigin máltilfinningu til að leggja skilning í eftirfarandi setningu: Ef við eigum ekki mjólk vil ég kaupa mjólk, ef svo er ekki vil ég athuga hvort að við eigum kex og ef við eigum ekki kex vil ég kaupa það, annars fer ég ekkert í búðina. Hér er aðaláherslan lögð á mjólkurstöðuna okkar, ef við eigum ekki mjólk viljum við laga það, en ef við eigum mjólk þá getum við gert eitthvað annað.

Þarna eru komnar aðstæður þar sem við athugum mólkurstöðuna og fyllum á ef þarf, en ef við eigum nóg af mjólk þá viljum við samt athuga hvort að við eigum nóg af safa því að við gætum þurft að fylla á þar. Hér er kannski ekki augljóst en ef það vantar mjólk þá skiptir ekki máli hvort það vanti kex eða ekki, við förum í búðina og kaupum mjólk, við kaupum ekki kex. Þetta skilst kannski frekar á flæðiriti sem sést á mynd 5.1. Flæðiritið líkir eftir uppsetningu á skilyrðissetningum þannig að það sem er inni í gænum þríhyrningum eru spurningar sem þarf að svara, bláu ferhyrningarnir eru svo niðurstöður sem fást í málið.



Mynd 5.1: Hér sést hvernig setningin: „Ef við eigum ekki mjólk vil ég kaupa mjólk, ef svo er ekki vil ég athuga hvort að við eigum kex og ef við eigum ekki kex vil ég kaupa það, annars fer ég ekkert í búðina.” má setja fram sem fæðirit. Ef það er engin mjólk þá förum við og kaupum mjólk, en ef það er til mjólk þá athugum við hvort að það sé til kex og kaupum það ef það vantar, hins vegar ef við eigum bæði kex og mjólk er engin ástæða til að fara í búðina.

Vegna þess að áherslan er lögð á „við eigum ekki mjólk” þá er vitlegast að setja inn segð sem er með neitun. Ef yrðingin m stendur fyrir setninguna „ við eigum mjólk” þá er yrðingin ekki m (not m) „við eigum ekki mjólk”. Skoðum þetta í töflu 5.2, sambærilegri þeirri sem við sáum áður (tafla 5.1), nema í staðinn fyrir p og q notum við yrðinguna „það er til mjólk”.

[3] Hérna er gert ráð fyrir að mega giska óendanlega oft rangt.

Tafla 5.2: Sanntafla með ákveðnum yrðingum

m = það er til mjólk	ekki m = það er ekki til mjólk	
0	0	Bæði ósatt, getur ekki verið
0	1	
1	0	
1	1	Bæði satt á sama tíma, getur ekki verið

Við viljum að aðalatriðið komi fram í inngangspunktinum í skilyrðissetningunni okkar til að hún sé skýrt upp sett og skiljanleg, til þess gætum við þurft að nota neitun. Við sjáum betur í næstu þremur undirköflum hvað ætti að fara á hvaða stað, en eins og með góðar nafnavenjur þegar við nefnum breyturnar okkar skulum við venja okkur á strax í upphafi að skilyrðissetningarnar okkar eru skýrar.

Nú höfum við séð í inngangi þessa undirkafla orðunum ef og annars slengt fram. Við þekkjum þessi orð og skiljum hvernig á að nota þau í setningu til að kalla fram útkomu. En það sem við þurfum að gera núna er að átta okkur á því að þessi orð eru mun formlegri í forritun heldur en í daglegu tali. Eins og til dæmis: „Ertu ekki að hugsa um Jamie Lee Curtis?" Í íslensku er hægt að svara þessari spurningu með „já ég er ekki að hugsa um hana" eða „nei ég er ekki að hugsa um hana" og bæði skilst, einnig er hægt að segja „jú ég er að hugsa um hana". Forritunarmál eru ekki tungumál, þau eru formleg og því er engin tvíræðni í boði.

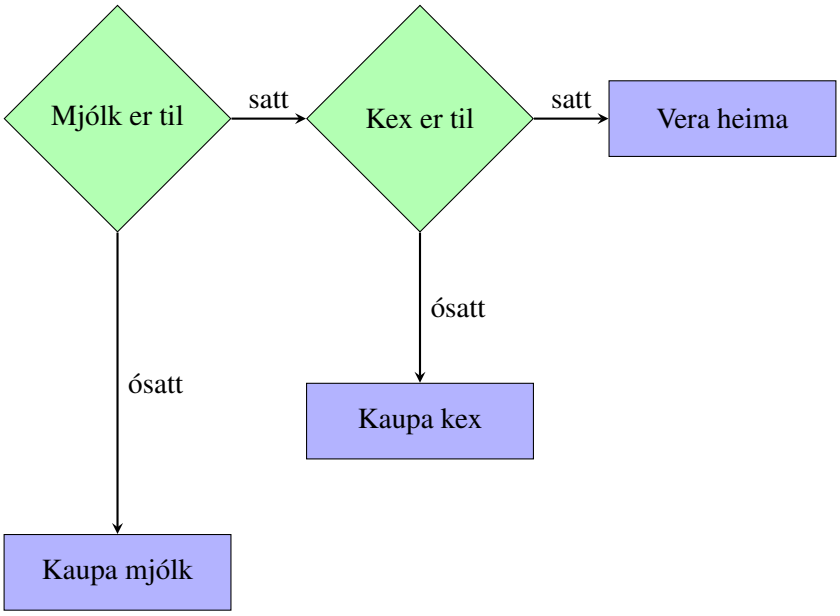
Skoðum því nú hvað það þýðir að nota skilyrðissetningar (e. conditional statements) í Python með lykilorðunum **if - elif - else**, sem verða þó kynnt annarri röð.

5.3.1 if

Fyrsta lykilorðið sem við tökum fyrir er **if**, þar sem ekki er hægt að búa til skilyrðissetningu án þess. Og nú þurfum við að huga að því hvernig kóðinn okkar er uppsettur. Það sem á að framkvæma undir ef setningunni/if yrðingunni er inndregið um fjögur bil eða einu sinni á „tab" takkann[4]. Eina sem ræður því hvað fer mikið af kóða undir hverja yrðingu er hóf og skynsemi. Við sjáum svo í kafla 5.3.4 um hreiðrun hvers vegna það er mikilvægt að skilyrðissetningar séu skýrar.

Góð venja er að búa til skilyrðissetningar þar sem aðalvirknin á sér stað inni í if yrðingunni, þannig að segðin sem fer þar inn passi við hvað eigi að framkvæma. Ef við tökum aftur dæmið um mjólkina og búðarferðina í mynd 5.1 og skoðum hvernig flæðiritið breytist eftir því hvernig við orðum skilyrðin. Það er að við skoðum hvernig uppsetningin á flæðiritinu verður bjöguð ef við orðum spurninguna með játun en ekki neitun: Ef það er til mjólk vil ég athuga hvort það sé til kex ef svo er vil ég vera heima, annars kaupi ég kex ef það er til mjólk en ekki kex og annars kaupi ég mjólk ef það er ekki til mjólk. Þetta er kannski ekki nógu flókin setning til þess að valda þeim hugrhrifum sem ætlast er til, en við sjáum að til þess að komast að þeim endapunkti sem aðaláherslan er á „vera heima" þar sem hún er fyrsti endapunkturinn okkar þá þurfum við að fara í gegnum tvær spurningar. Svo með því að orða spurninguna öðruvísi erum við búin að setja upp skilyrðissetninguna upp þannig að mólkurstaðan er núna ekki lengur í forgrunni, við virðumst frekar vera að reyna að halda okkur heima.

[4]Hann er fyrir ofan „CapsLock" takkann.



Mynd 5.2: Hér sést hvernig setningin: „Ef það er til mjólk vil ég athuga hvort það sé til kex ef svo er vil ég vera heima, annars kaupi ég kex ef það er til mjólk en ekki kex og annars kaupi ég mjólk ef það er ekki til mjólk.” má setja fram sem fæðirit. Þetta veldur því að aðaláherslan virðist nú vera að komast að því hvort eigi að kaupa kex eða vera heima og mjólkurstaðan er athuguð fyrst af einhverri ástæðu. Setningin í heild er frekar ruglingsleg og hún kom mun betur út í flæðiritinu á mynd 5.1. Þó áherslan sé önnur er niðurstaðan sú sama, það er á ábyrgð forritara að skrifa kóða sem er læsilegur og skiljanlegur.

Skoðum nú kóðabút 5.3 og hvað er átt við með réttum inndrætti, hér er einungis sýnt if setning ein og stök. Í kóðabútnum tökum við fyrir segðina „er til mjólk?” og veitum henni gildið `True` svo við erum stödd í þeim aðstæðum að við eigum vissulega til mjólk, skoðið töflu 5.2 til að sannfærast og það sem er að gerast. Við sjáum í næsta undirkafla hvað við getum gert ef við förum framhjá if setningunni okkar og viljum gera eitthvað í því tilfelli.

Kóðabútur 5.3: if notað

```
1 mjolk = True
2
3 if(not m):
4    print('við fórum í búðina og keyptum mjólk')
```

```
1 # við áttum mjólk svo ekkert prentast
```

5.3.2 else

Lykilorðið **else** má fylgja **if**, en það er ekki nauðsynlegt. Hinsvegar verður að vera eitthvað *ef* til þess að það geti verið eitthvað *annars*. Setningin „annars kaupi ég mjólk” er ekki sérlega vitræn því að okkur vantar alveg fyrri hlutann. Einnig er ekki sérlega vitrænt að segja „ég kaupi mjólk ef vantar annars kaupi ég kex annars kaupi ég te annars...”. Því er einungis hægt að setja eitt annars við hvert ef, sjáum kóðabút 5.4. Sú klausa keyrist einungis þegar ef setningin sem hún hangir fyrir neðan keyrist ekki, það er eina skilyrðið. Það þarf ekki að spyrja neinnar spurningar sem er metið sem boolean gildi til að keyra else, það mun alltaf keyrast þegar segðin í ef yrðingunni var ósönn.

ing

Kóðabútur 5.4: else notað

```
1  mjolk = True
2
3  if(not mjolk):
4     print('við fórum í búðina og keyptum mjólk')
5  else:
6     print('vera heima')
7
8  if(3 < 4):
9     print("þrír er minna en fjórir")
10 else:
11    print('ég fer ekki hingað inn, því 3 er vissulega minna en fjórir, en það er gott að
       vera við öllu búin')
```

```
1  vera heima
```

Að geta sett svona annars-klausu er mikilvægt því að við viljum geta brugðist við ef upphaflega skilyrðið okkar er ósatt. Við viljum geta tekið á öðrum tilfellum heldur en bara upphafsskilyrðinu okkar.

5.3.3 elif

En hvað ef við viljum geta tekið á einhverju sérstöku tilfelli, sem kemur einungis upp í ákveðnum aðstæðum? Við viljum ekki bara grípa það að inngangspunkturinn okkar var ósannur heldur viljum við einnig skoða eitthvað fleira? Þarna kemur setningin um mjólkina, kexið og búðarferðirnar aftur inn. Þá getum við sagt „ef það er ekki til mjólk, fer ég í búð, **annars ef** það er ekki til kex fer ég í búð og kaupi kex, nú annars er engin ástæða til að fara í búðina og ég verð bara heima". Við viljum bara nota þetta seinna ef í ákveðnu tilfelli, við höfum ekkert að gera við kexið ef það er engin mjólk svo það er aðeins keyrt ef við vissulega eigum hana. Sjáum þetta forritað í kóðabút 5.5.

Skilyrðissetningar eru settar upp þannig að það verður að vera eitt **if** svo mega koma núll eða fleiri **elif** og að lokum má setja 0 eða 1 **else**. Þetta er eins og málfræðilegur skilningur okkar er á tungumálinu, við megum hengja endalaust af annars ef klausum inn í setningarnar okkar, þær verða þá bara erfiðari að skilja (kóðinn sömuleiðis).

Kóðabútur 5.5: elif notað

```
1  mjolk = True
2  kex = False
3
4  if(not m):
5     print('við fórum í búðina og keyptum mjólk')
6  elif(not k):
7     print('við fórum í búðina og keyptum kex')
8  else:
9     print('vera heima')
10
11 if(5 < 4):
12    print("fimm er minna en fjórir")
13 elif(4 < 4):
14    print("fjórir er minna en fjórir!")
15 elif(3 < 4):
16    print("þrír er minna en fjórir")
17 elif(2 < 4):
18    print("en 2 er líka minna en fjórir!")
19 else:
20    print('eitthvað er að')
```

```
við fórum í búðina og keyptum kex
þrír er minna en fjórir
```

5.3.4 Hreiðrun

Hreiðrun (e. nesting) þýðir að setja eitthvað endurtekið undir eitthvað annað, eins og babúska dúkkur eða að pakka gjöf inn í mörg lög af gjafapappír. Í forritun þýðir hreiðrun að yrðing af einhverri tegund tilheyri og sé keyrð innan í yrðingu af sömu tegund. Við getum hugsað þetta í samhengi við skilyrðissetningar að við séum með innri skilyrðissetningar sem þarf einnig að meta til þess að komast að niðurstöðu. Skoðum þetta aftur í samhengi við mjólkurkaupin nema nú bætum við því við að við eigum bara ákveðið mikinn pening sjá kóðabút 5.6.

Kóðabútur 5.6: Hreiðrun

```
mjolk = True
kex = False
peningar = 100

if(not mjolk):
   if(p > 200):
      print('við fórum í búðina og keyptum mjólk því við vorum með nógu mikinn pening')
   else:
      print('okkur vantaði mjólk en við vorum ekki með nógu mikinn pening')
elif(not kex):
   if(p > 99):
      print('við fórum í búðina og keyptum kex því við vorum með nægan pening')
   else:
      print('okkur vantaði kex en við vorum ekki með nægan pening')
else:
   print('vera heima og geyma allan peninginn')
```

```
við fórum í búðina og keyptum kex því við vorum með nægan pening
```

Hreiðrun er gagnleg því að við viljum skoða eitthvað innra skilyrði aðeins ef ytra skilyrðinu er mætt.

5.4 Inntak

Nú höfum við verið að skoða spurningar og svör við þeim sem við skráðum sjálf. Það sem við viljum geta gert er að spyrja notandann að einhverju og geta gert eitthvað byggt á því svari. Við þurfum að fá inntak (e. input) frá notandanum. Þá lærum við um nýtt innbyggt fall í Python sem heitir `input()`. Það sem input() gerir er að taka við streng frá notanda, notandi skrifar eitthvað inn í þartilgert svæði, og við getum notað það í forritinu okkar[5].

Skoðum kóðadæmi í kóðabút 5.7, þar sem við geymum svarið frá notanda í breytunni `svar` og viðfangið sem við settum inn í input fallið er strengur sem inniheldur spurninguna sem notandinn sér. Þar sést þó ekki þegar inntak glugginn var notaður.

Fallið input() skilar okkur alltaf streng. Ef við viljum geta spurt notandann um tölustafi þurfum við að kunna að kasta á milli gagnataga sjálf og við sjáum hvernig það er gert í kóðabút 5.9

[5] Að nota input() í skipanalínu gefur okkur nýja línu til að svara, að nota input() í Jupyter Notebooks gefur okkur lítinn glugga til að skrifa svarið okkar í fyrir neðan selluna þar sem input() skipunin er keyrð.

Kóðabútur 5.7: input() fallið notað

```
1 svar = input('skrifaðu nafnið þitt')
2 print('halló', svar)
```

```
1  Valborg
2 halló Valborg
```

Þetta er mikilvægt fall fyrir okkur að skilja og nota á þessu stigi málsins, því að við verðum að átta okkur á því að þegar við forritum þá erum við miklu meira að vinna með breytunöfn heldur en gögn sem við getum horft á. Í kóðabút 5.7 þá kemur hvergi fram í kóðanum að nafnið sé Valborg, og það getur verið hvað sem er, við prentum bara út það sem notandinn gaf okkur án þess að vera eitthvað að skoða hvað það er. Oft vilja byrjendur horfa á gögnin sín og setja inn niðurstöður fyrir tölvuna, til dæmis verkefnið „búðu til breytu sem inniheldur nafn og prentaðu út breytuna ásamt strengnum "halló:“' endar í kóða eins og sést í kóðabút 5.8. Eða að finna miðju í streng sem hægt væri að gera með `len(strengur)//2`, en byrjendum finnst eðlislægara að finna lengdina, og finna svo helminginn af því og nota svo loks þá tölu

Kóðabútur 5.8: Oft forðast byrjendur að nota breytur og treysta meira á að sjá hvað ætti að koma út

```
1  nafn = 'Valborg'
2  print('Halló: Valborg')
3
4  strengur = "þessi strengur hefur 31 tákn!!!"
5  # þetta þyrfti að gera í skrefum
6  print(len(strengur))
7  print(31/2)
8  print(strengur[15])
9
10 print(strengur[len(strengur)//2])
```

```
1 Halló: Valborg
2 31
3 15.5
4 h
5 h
```

Eins og sést í kóðabút 5.8 að til að komast að því að setja vísi 15 inn og fá táknið þarf að keyra fyrst línuna `len(strengur)` til þess að sjá þá tölu og svo þarf að deila þeirra tölu með tveimur til að finna miðjuna og svo þarf handvirkt að setja þá tölu inn eftir að hafa breytt henni í næstu heilu tölu. Það sem er að gerast er ekkert rangt, það er hinsvegar mikil vannýting á því sem tölvan getur gert fyrir okkur og eykur vinnuna fyrir okkur sjálf umtalsvert því að það þarf að keyra hvert skref í kóðabútnum fyrir sig til að komast að því hvað eigi að gera í næsta skrefi, í stað þess að gera það í einni línu eins og í línu 10.

5.4.1 Kastað á milli gagnataga

Til þess að geta unnið með gögn eins og þá týpu sem við viljum þurfum við að læra að *kasta* á milli taga/týpna (e. typecasting). Þetta þýðir að við látum vélina umrita gögnin okkar yfir í annað gagnatag, sem er einungis hægt ef að gögnin eru sambærileg týpunni sem á að kasta í.

Þá koma lykilorðin sem við höfum lært fyrir týpurnar okkar að gagni. Við þekkjum núna strengi með lykilorðið **str**, heiltölur með lykilorðið **int**, fleytitölur með lykilorðið **float** og lista með lykilorðið **list**. Þá notum við lykilorðið eins og fall og setjum inn í fallið sem viðfang það sem á að verða að því gagnatagi sem lykilorðið segir til um. Sjáum í kóðabút 5.9 hvernig á að fara að þessu.

Nú höfum við séð að input() fallið skilar alltaf til okkur gögnum af taginu/týpunni strengur. En við viljum geta kannski unnið með inntakið frá notandanum sem tölu. Ef að strengurinn inniheldur

einungis tölur á bilinu 0-9 er hægt að geyma hann sem heiltölu eða fleytitölu, ef hann inniheldur einungis tölur á bilinu 0-9 og nákvæmlega einn punkt er hægt að geyma hann sem fleytitölu.

Kóðabútur 5.9: Hvernig á að kasta á milli gagnataga

```
talnastrengur = "123"
heiltala = int(talnastrengur)
fleytitala = float(talnastrengur)

fleytitolustrengur = "3.1415"
talan_pi = float(fleytitolustrengur)
```

Nú þar sem við vitum að við getum fengið streng í hendurnar frá notanda, vitandi það að við báðum um tölu, getum við leyft okkur að kasta strengnum í það talnatag sem okkur hentar.

Við sjáum svo í seinni hluta bókarinnar hvernig á að taka á mismunandi tilfellum og reyna á eitthvað sem gæti valdið villu án þess að það skemmi fyrir okkur, en núna ætlum við að láta sem að við getum treyst notendum til að gefa okkur inntak sem samræmist því sem við báðum um. Prófið ykkur nú áfram með að kasta á milli taga, þið munið fá einhverjar villur og það er frábært. Gerið tilraunir og prófanir, kastið á milli allra þeirra taga sem þið þekkið í öll þau tög sem þið þekkið, hvað má hvað má ekki?

Ástæðan fyrir því að vilja kasta á milli taga er til að geta beitt þeim aðgerðum og aðferðum sem eru í boði fyrir það gagnatag sem við sækjumst eftir að nota, til dæmis er ekki hægt að sækja þriðja tölustafinn í heiltölu en ef við köstum henni í streng getum við sótt táknið í sætisnúmeri 2 og fengið þannig þriðja tölustafinn.

Ítarefni 5.2 Dæmi um notkun á kasti milli taga

Seinna munum við sjá gagnatýpuna mengi, ein gagnleg notkun á kasti milli taga væri að kasta lista í mengi til að losna við tvítekningar og kasta menginu svo í lista aftur.

```
a = [1, 2, 2]
b = set(a)
a = list(b)
```

Nú er listinn `a` orðinn að `[1,2]`, sjá meira um það í kafla 9.

6. Lykkjur

Til þess að keyra kóða endurtekið án þess að afrita og líma eða handvirkt keyra hann oft, þá notum við lykkjur. Lykkjur eru kóðabútar sem keyrist endurtekið, eða ítrar, eftir ákveðnum reglum. Þær lykkjur sem eru til í Python eru **for** lykkjur og **while** lykkjur. For lykkjur keyra fyrir hvert stak í ítranlegum hlut eða hverja tölu á bili (keyra ákveðið oft, í mesta lagi). While lykkjur keyra á meðan skilyrðið fyrir keyrslu þeirra er satt (geta keyrt að „eilífu"). Nöfnin á for og while verða ekki þýdd sérstaklega í þessari bók, en við segjum t.a.m. „að gera eitthvað á meðan" eða „gera eitthvað fyrir hvert stak". Við ætlum að kynnast því til hvers þær eru ætlaðar og hvers þær eru megnugar, í hvaða tilfellum á að nota hvora fyrir sig og lykilorð sem gera notkun þeirra öflugri. Byrjum á því að skoða til hvers „að lykkja" og hvað það eiginlega þýðir. Það að nota lykkju þýðir að skrifa forritsbút sem keyrir endurtekið.

Tökum dæmi úr daglegu lífi; ef við viljum framkvæma einhverja aðgerð eins og að vaska upp búum við til reglu eins og að setja fyrst upp uppþvotta-hanska, láta vatnið renna og stafla öllu sem er óhreint við hliðina á vaskinum. Svo viljum við endurtaka aðgerðina að þrífa hvern hlut sem er öðru megin við vaskinn, þar til þeir eru allir komnir hreinir hinu megin. Endurtekningin þarna er að taka upp hvern óhreinan hlut og þrífa hann. Þá gætum við sagt að fyrir hvern hlut sem er hægra megin, viljum við þrífa hann og setja svo vinstra megin (fer eftir því hvernig vaskurinn snýr) og hætta þegar engir hlutir eru eftir hægra megin. Þetta ferli að framkvæma sömu aðgerð á stök í mengi er einmitt það sem for lykkja getur gert.

Tökum annað dæmi úr daglegu lífi; ef við ætlum að bíða eftir einhverjum og framkvæma svo einhverja aðgerð þegar viðkomandi kemur þá myndum við væntanlega bíða þangað til að viðkomandi kemur. Svo á meðan viðkomandi er ekki enn kominn þá höldum við áfram að bíða. En þar sem við erum ekki tölvur þá myndum við ekki bíða endalaust, við myndum gefast upp. Þetta ferli að halda áfram að framkvæma einhverja aðgerð þangað til að eitthvað skilyrði á ekki við er það sem while lykkja getur gert.

6.1 For

For lykkkjur nota lykilorðið **for** ásamt lykilorðinu **in**. Það sem in gerir þegar það er notað eitt og sér er að spyrja hvort að *eitthvað* sé „í" *einhverju öðru* (sjá kóðabút 6.1) en í sem hluti af for lykkju þá er það in sem úthlutar lykkjunni næsta staki úr menginu til að skoða. Þannig að þetta býr til segð sem skilar sanngildi eða einu tilteknu tákni eða staki úr hlut. Nú skulum við líta á kóðabút **??** til að átta okkur á því hvernig lykkjan er notuð, hvernig við beitum inndrætti til að skilgreina stef lykkjunnar (það sem tilheyrir henni) og hvernig skilyrðissetningar bætast við þetta.

Kóðabútur 6.1: Lykilorðið in

```
print('er táknið a í strengum Valborg?')
print("a" in "Valborg")

print('er táknið x í Valborg?')
print("x" in "Valborg")
```

```
er táknið a í strengum Valborg?
True
er táknið x í Valborg?
False
```

Eins og sést í kóðabút 6.1 þá virkar in nokkuð svipað því sem orðið í gerir í setningu. Ekki gleyma þessu lykilorði við lykkjugerðina. Sjá ítarefni í lok kaflans um önnur lykilorð sem gagnast við forritun á lykkjum. Í næstu kóðabútum verður sýnd grunnvirkni for-lykkjunnar.

Ítarefni 6.1 Nánar um `in` og vísa

Þegar þetta orð er notað í for lykkjum er þó ekki verið að setja fram segð heldur er verið að úthluta einhverri hlaupandi breytu tilteknu gildi úr ítranlegum hlut. Það að hlutur sé ítranlegur þýðir að við getum horft á hann stak fyrir stak, skoðað eitt gildi úr honum í einu. Eins og strengur hefur vísa þá getum við horft á hvert tákn fyrir sig með því að rúlla í gegnum vísana frá 0 og út í enda (eða einhverri annarri röð). Listar eru einnig ítranlegir þar sem stökin í listum hafa vísa og því má horfa á hvert stak fyrir sig í heild sinni, hvort sem það er annar listi eða ein stök tala. Heiltölur, fleytitölur og sanngildi eru ekki ítranleg og því ekki hægt að rúlla í gegnum þau með for lykkju. Hægt er að komast framhjá því með því að kasta þeim í strengi.

formatting

Kóðabútur 6.2: For lykkjur kynntar

```
1  # við byrjum á að skilgreina streng
2  strengur = "Valborg"
3  print(strengur, "til viðmiðunar")
4
5  for stafur in strengur:
6     print(stafur)
7
8  print()
9  print('lykkjan er búin')
```

```
1  Valborg til viðmiðunar
2  V
3  a
4  l
5  b
6  o
7  r
8  g
9
10 lykkjan er búin
```

Í kóðabút 6.2 hvernig rúllað er í gegnum strenginn `Valborg` með breytunni `stafur`. Sú breyta er búin til í línu 5 þegar lykkjan er búin til, það þurfti ekki að skilgreina hana áður. Það er vegna þess að breytan er skilgreind inni í lykkjunni fyrir okkur, hún er áfram aðgengileg en er ósköp gagnslaus eftir keyrsluna svo okkur er alveg sama um hana, hún er svokölluð *tímabundin* (e. temporary) breyta sem hættir að skipta máli eftir notkun innan lykkjunnar.

Allt það sem tilheyrir svo lykkjunni eða á að gerast í hverri keyrslu hennar er inndregið undir henni. Línur 9 og 10 eru ekki hluti af stefi lykkjunnar og keyrast því ekki nema einu sinni, eins og lína 3 prentast bara einu sinni.

Það sem prentast á úttakið úr línu 6, breytan `stafur`, er hvert tákn fyrir sig í þeirri röð sem það kemur fyrir í strengnum sem verið er að ítra í gegnum. Lesið yfir þennan kóðabút og gerið tilraunir á eigin spýtur til að átta ykkur á því hvað það er sem er að gerast.

Hvað gerist ef inndrátturinn breytist? Hvað prentast þá út? Hvað gerist ef eitthvað annað orð er sett í staðinn fyrir `stafur`? En `strengur`? Haldið áfram að gera tilraunir með þetta þangað til að þið áttið ykkur betur á því hvernig þetta hangir saman.

Sjáum nú hvernig megi flétta skilyrðissetningar inn í þetta.

Kóðabútur 6.3: For lykkja og skilyrðissetningar

```
1  for stafur in strengur:
2     # við vitum að það er a í Valborg svo þetta mun einhvern tímann gerast
3     if(stafur == 'a'):
4        print(stafur)
```

```
1  a
```

Í kóðabútum 6.2 og 6.3 vorum við að vinna með sama strenginn, en í fyrra skiptið prentaðist hann allur út en í seinna skiptið fengum við bara eitt stakt tákn út. Munurinn er sá að í kóðabút 6.3 þá þegar við vorum komin með táknið í hendurnar vildum við gera eitthvað við það svo við spurðum er þetta tákn jafngilt `a`, einungis í því tilfelli vildum við prenta eitthvað út. Við ákváðum að prenta út táknið sem við vorum að skoða sem er geymt í breytunni `stafur` en hefðu hæglega getað gert eitthvað annað.

Við sjáum einnig að þegar við settum inn skilyrðissetningu þá bættist við annar inndráttur, það er vegna þess að inndráttarnotkunin breytist ekki sama hvar við erum að nota kóða sem krefst inndráttar heldur dregst kóðinn bara lengst til hægri eftir því sem við förum innar. Því getur verið ágætt að takmarka hreiðrun til þess að kóðinn sé sem læsilegastur.

Gerið nú tilraunir til að átta ykkur betur á því hvernig megi skoða ítranlegan hlut en framkvæma einungis aðgerð ef eitthvað skilyrði á við. Hvað gerist ef við breytum skilyrðissetningunni? En ef við breytum því sem er undir henni? Hvað gerist ef við bætum við *annars* klausu? Má setja eitthvað inn í stef lykkjunnar á eftir skilyrðissetningunni?

Næst skoðum við annan ítranlegan hlut með for lykkju, það er listi.

Kóðabútur 6.4: For lykkja með lista

```
1 listinn_minn = [0, "strengur", [0, 1, 2]]
2
3 for x in listinn_minn:
4    # nú hleypur x í gegnum stökin í breytunni listinn_minn
5    print(x)
```

```
1 0
2 strengur
3 [0, 1, 2]
```

Tökum eftir í kóðabút 6.4 þá fær `x` það gildi sem er næst í röðinni í listanum, og það skiptir ekki máli af hvaða týpu gögnin eru. Nafnið á breytunni er ekki lýsandi, ekki eins og `stafur` eða `strengur` í kóðabút 6.3, ástæðan fyrir því að nafnið x varð fyrir valinu er til að sýna lesendum að þetta er breytuheiti eins og hvert annað og lútir sömu venjum og við sáum í kafla 2. Listinn í línu 1 inniheldur gögn af þremur týpum, breytan `x` kippir sér ekkert upp við það og birtir gögnin í þeirri röð sem þau bárust.

Prófið ykkur nú aðeins áfram og reynið í staðinn fyrir `listinn_minn` að setja einhvern annan lista, sem er ekki geymdur í breytu. Prófið nú að setja inn skilyrðissetningu þarna og nota `type()` með skilyrðissetningu til að prenta einungis þau x sem eru strengir.

For lykkjur eru því helst gagnlegar þegar við vitum hversu mörg stök lykkjan á mörgulega að skoða, því til stuðnings ætlum við að skoða innbyggða fallið `range()` sem gefur okkur hlut af tölum á ákveðnu bili. Fallið `range()` tekur við sömu viðföngum eins og hornklofarnir[1] þegar við sóttum nokkur tákn upp úr streng eða lista, þau eru þó aðgreind með kommum[2].

Viðföngin í `range(a,b,c)` fallið eru heilartölur og þeim er raðað svona:

1. **a**: talan sem á að byrja að nota (hér má sleppa því að setja þetta inn því að sjálfgefið gildi er 0)
2. **b**: talan sem á að hætta fyrir framan (þetta verður að setja inn, því að þetta er aðalatriðið)
3. **c**: tala sem segir til um skrefastærðina (sjálfgefið gildi er 1 og þessu má sleppa), ef skrefastærð er tekin með þarf að velja upphafsstað (annars myndast tvíræðni)

Ef við viljum leysa verkefni sem felst í því að finna odda tölur frá 0 og upp að 1000 hvernig myndum við fara að því? En ef við viljum vera viss um að eitthvað gerist ákveðið oft? Skoðum kóðabút 6.5 þar sem fyrra verkefnið er leyst á tvo mismunandi vegu.

Kóðabútur 6.5: range() fallið kynnt með for lykkju

```
1 for tala in range(6):
2    if(tala%2 != 0):
3       print(tala)
4
```

[1] það eru í raun ekki viðföng, í "strengur"[1:5:2] eru 1, 5, og 2 ekki viðvöng beinlínis heldur vísar.

[2] Við höfum áður séð viðföng notuð í falli eins og print() fallið, þar sem við getum prentað margt út svo lengi sem við setjum kommur á milli þess.

```
5  print()
6
7  for tala in range(0,5,2):
8     print(tala)
9
10 print()
11
12 for x in range(3):
13    print('bílalúgudýraspítali', x)
```

```
1  1
2  3
3  5
4
5  0
6  2
7  4
8
9  bílalúgudýraspítali 0
10 bílalúgudýraspítali 1
11 bílalúgudýraspítali 2
```

Aðalatriðið sem þarf að hafa í huga þarna í kóðabút 6.5 er að for lykkjan hleypur í gegnum lista af tölum svo að við vitum alltaf hvar við erum stödd og við vitum hvað við keyrum lykkjuna oft. Sjáum í línu 13 þá er `x` prentað og á úttakinu (línur 9-11) sést að það er hlaupandi númer sem byrjar í 0 og hættir í 2 sem er talan fyrir framan 3 og þar vildum við hætta. Nú getið þið breytt þessum lykkjum til að skoða til dæmis sléttar tölur undir 1000 eða tölur deilanlegar með 17 undir 100.

En við þurfum ekki endilega að byrja í núll, sjáum í kóðabút 6.6 hvernig hægt er að velja afmarkaðra talnabil og svo sjáum við í kóðabút 6.7 að hægt er að telja afturábak.

Allt er þetta þó spurning um að finna það sem hentar því verkefni sem við erum að reyna að leysa. Næstu sýnidæmi eru meira til þess fallin að sýna virkni `range()` fallsins og for lykkja yfirhöfuð án þess þó að vera að leysa einhver flókin vandamál. Eftir að hafa séð þetta, gætuð þið prófað ykkur áfram og náð þannig tökum á þessu.

Það sem skiptir mestu máli til að ná ákveðinni leikni er að prófa sig áfram, gera tilraunir og þora að mistakast.

Kóðabútur 6.6: for lykkja og range() fallið með skilyrðissetningu

```
1 for tala in range(10, 20):
2    if(tala%3 == 0):
3       print(tala)
```

```
1 12
2 15
3 18
```

Eins og áður kom fram þá þarf að taka fram upphafspunkt ef nota á skrefastærð svo að í línu 1 í kóðabút 6.7 fer ekki milli mála að byrja á fyrir framan töluna 10 og enda fyrir aftan töluna 20. Svo það sem gerist í skilyrðissetningunni er að tölunni er kastað í streng og spurt er hvort að síðasta táknið í strengnum sé talan 9. Ef svo er þá prentum við út töluna ásamt textanum `endar á 9`, svo þegar keyrslu lykkjunnar lýkur fáum við að sjá hvað breytan `tala` inniheldur.

Kóðabútur 6.7: for lykkja range() fallið notað til að telja aftur á bak

```
1 for tala in range(100, 70, -1):
2    if(str(tala)[-1] == '9'):
3       print(tala, 'endar á 9')
4 print('lykkjan er búin, hvað er tala?', tala)
```

```
99 endar á 9
89 endar á 9
79 endar á 9
lykkjan er búin, hvað er tala? 71
```

6.1.1 Gagnleg lykilorð

Áður en lengra er haldið í hvernig á að beita lykkjum er ágætt að nefna nokkur grunn lykilorð sem hjálpa okkur gríðarlega. Þau eru **pass**, **continue** og, **break**.

Kóðabútur 6.8: Lykilorðið pass notað með for lykkju

```
for x in range(15):
    if(x % 3 != 0):
        pass
    else:
        print('þetta gerðist fyrir töluna', x)

```

```
þetta gerðist fyrir töluna 0
þetta gerðist fyrir töluna 3
þetta gerðist fyrir töluna 6
þetta gerðist fyrir töluna 9
þetta gerðist fyrir töluna 12
```

Kóðabútur 6.9: Lykilorðið continue notað með for lykkju

```
for x in [1, 2, 59, 9, 53, 2]:
    if (x < 50):
        continue
    print(x)
```

```
59
53
```

Kóðabútur 6.10: Lykilorðið continue notað með for lykkju

```
listi_af_tolum = [1,5,7,9,13,15,17,18]
for tala in listi_af_tolum:
    if(tala == 13):
        print("það er þrettán í listanum")
        break
    elif(tala != 13):
        continue
    else:
        print("þetta prentast aldrei")

```

```
"það er þrettán í listanum"
```

Í kóðabútum 6.8, 6.9 og, 6.10 sjáum við að lykilorðin geta gefið okkur möguleika á að hætta keyrslu, nota bara hluta úr kóða eða gefa okkur kost á að nýta staðhaldara þegar við vitum ekki hvaða kóði á að koma þangað. Án þess að fara meira út í hvernig kóðinn fyrir þessi lykilorð virka þá er þess virði að nefna að þau eru ekki nauðsynleg í hverri lykkju sem við forritum hér eftir, þau eru gagnleg þegar þau eiga við og við þurfum að átta okkur á hvernær svo er.

Ítarefni 6.2 Nánar um lykkju lykilorðin

- **pass** er lykilorð sem gerir ekkert, tölvan heldur áfram keyrslu sinni eins og ekkert hafi verið gert, nema að þarna er kóði sem er rétt inndreginn og gerir það að verkum að tölvan kvartar ekki yfir því að hafa búist við einhverju inndregnu en fengið ekkert. Þetta notum við þegar við erum ekki viss hvað á að vera í lykkjunni og við setjum þetta orð inn svo að við getum haldið áfram með annað sem átti að forrita. pass er gagnlegt sem staðhaldari (e. placeholder) þegar við erum ekki viss hvernig á að halda áfram en verðum að setja eitthvað því að annars fengjum við málskipanar villu (e. syntax error). Þetta lykilorð má nota annarsstaðar en í lykkjum og er einnig gagnlegt sem staðhaldari í föllum.
- **continue** er lykilorð sem lætur vélina stoppa þar sem hún er í lykkjunni, hunsa allt sem kemur á eftir því og fara efst í lykkjuna. Continue er gagnlegt þegar kemur að því að það er bara ákveðin virkni sem á að framkvæma undir vissum aðstæðum og við viljum ekki að vélin geri allar aðgerðir sem koma fram í lykkjunni okkar. Þetta lykilorð má einungis nota inni í lykkjum.
- **break** hættir keyrslu lykkjunnar, ólíkt continue þá förum við alfarið út úr lykkjunni þegar kallað er í þetta lykilorð og keyrir vélin næst kóða sem er ekki inndreginn undir lykkjunni. Þetta lykilorð má einungis nota inni í lykkjum. Þetta lykilorð getur reynst ómetanlegt þegar við skoðum while lykkjur.

6.2 While

While lykkjur nota lykilorðið **while** og keyra „á meðan" eitthvað skilyrði er satt. Þær eru helst gagnlegar þegar við vitum ekki hvað við viljum að lykkjan keyri lengi eða þegar við viljum að hún keyri endalaust nema annað sé tekið fram (t.d. með break).

Skilyrðið fyrir keyrslunni er metið sem sanngildi, annað hvort með sanngildinu sjálfu eða segð sem skilar sanngildi. Þá gefst okkur tækifæri á að forrita lausn á vanda eins og „ef það er enginn eftir í stofunni á að slökkva ljósið" og forritið keyrir á meðan „einhver er eftir í stofunni". Þarna þurfum við ekki að gera annað en að fylgjast með aðstæðum. While lykkjur eru vandmeðfarnar og harla líklegt að lenda í því að skrifa lykkju sem keyrir endalaust við fyrstu notkun. Þær eru jafnframt öflugar til að leysa ýmsan vanda sem krefst þess að aðstæður hverju sinni séu skoðaðar.

Skoðum kóðabút 6.11 til þess að sjá hvernig má auðveldlega lenda í vandræðum við gerð slíkra lykkja og hvernig uppsetning þeirra lítur út.

Kóðabútur 6.11: while lykkja sem keyrir að eilífu

```
while(True):
    # inndreginn kóði sem tilheyrir lykkjunni - stef lykkjunnar
    pass
print('kemst ekki hingað því lykkjan er enn að keyra')
```

```
# ef lykkjan okkar gerði eitthvað væri þessi bútur troðfullur
```

Lykkjan í kóðabút 6.11 keyrir að eilífu vegna þess að skilyrðið fyrir henni er `True` og ekkert breytir því í stefi hennar. Hægt er að stöðva vélina handvirkt í þessum aðstæðum[3]. En nú viljum við að þetta gerist ekki aftur svo við notum break lykilorðið.

[3] Í Jupyter Notebooks er það gert með Kernel -> Restart Kernel

Kóðabútur 6.12: while lykkja sem keyrir ekki að eilífu en hún gerir ekkert

```
1 while(True):
2    # nú ætlum við að reyna að komast út
3    break
4 print('vei við komumst út, en hvað kostaði það?')
```

```
1 vei við komumst út, en hvað kostaði það?
```

Við komumst út úr lykkjunni, hún keyrði einu sinni og hætti strax keyrslu, ekki mjög gagnleg lykkja en hún keyrði allavega ekki að eilífu. Annað sem við þurfum líka að hugsa um er að skilyrðið okkar sé alveg örugglega rétt skilgreint, að við séum að ná að fanga þær aðstæður sem við vildum halda í. Skoðum næsta kóðabút þar sem skilyrðið mun aldrei verða satt og því mun stef lykkjunnar aldrei keyrast og breytan sem er þar skilgreind aldrei fá stað í minni, sem veldur villu þegar á að nota breytuna á eftir lykkjunni.

Kóðabútur 6.13: while lykkja sem keyrir aldrei

```
1 while(3 < 2):
2    print('þetta mun aldrei prentast því að stef lykkjunnar mun aldrei keyrast')
3    x = 5
4 print(x)
```

```
1 NameError                                 Traceback (most recent call last)
2 <ipython-input-21-8ba2a8c60ab2> in <module>
3 2         print('þetta mun aldrei prentast því að stef lykkjunnar mun aldrei keyrast')
4 3         x = 5
5 ----> 4 print(x)
6
7 NameError: name 'x' is not defined
```

Takið sérstaklega eftir því hvað villuskilaboðin eru skýr í úttakinu á kóðabút 6.13, að villan er nafnavilla, hún á sér stað í línu fjögur í kóðanum og að það er vegna þess að 'x' er ekki skilgreint þegar það er notað í línu 4.

Skoðum nú einhverja gagnlega lykkju. Segjum að við skuldum 10.000 krónur og við ætlum að borga inn á skuldina okkar 1.000 krónur í einu. Við viljum að sjálfsögðu hætta að borga þegar við skuldum ekkert lengur og auðvitað viljum við að skuldin okkar lækki.

Kóðabútur 6.14: while lykkja sem eitthvað vit er í

```
1 skuld = 10000
2 innborgun = 1000
3 while(skuld > 0):
4    skuld = skuld - innborgun
5    print('nú er skuldin', skuld)
```

```
1 nú er skuldin 9000
2 nú er skuldin 8000
3 nú er skuldin 7000
4 nú er skuldin 6000
5 nú er skuldin 5000
6 nú er skuldin 4000
7 nú er skuldin 3000
8 nú er skuldin 2000
9 nú er skuldin 1000
10 nú er skuldin 0
```

Nú þegar við höfum skoðað haldbært dæmi um eitthvað sem vit er í skulum við skoða óhlutbundið dæmi þar sem við erum að vinna með hugmyndina um að slökkva ljósin í stofunni ef allir

eru farnir.

```
while(True):
   if(fjöldi nemenda er 0):
      slökkva ljós
      break

   fjöldi nemenda talinn aftur
```

Þetta er ekki alvöru Python kóði, heldur *sauðakóði* (e. psudocode) sem kemur þó merkingunni til skila, að aðalatriðið er að vera í sífellu að skoða það hvort að engir nemendur séu eftir og telja þá aftur. Þar kemur while lykkjan sterk inn, að við viljum gera eitthvað á meðan eitthvað ástand varir. Takið eftir að talning nemenda fer fram inni í lykkjunni, ef sá hluti yrði færður einum inndrætti innar væri það ekki lengur hluti af stefi lykkjunnar og keyrðist þegar henni væri lokið (en henni myndi aldrei ljúka því að aldrei yrði komist inn í skilyrðissetninguna því að ekkert breytir fjölda nemenda innan lykkjunnar).

Hugsum okkur nú að nota segð fyrir eitthvað flóknara skilyrði en við sáum í kóðabút 6.13. Eins og við sáum á myndum 5.1 og 5.2 þá skiptir máli hvernig við orðum skilyrðin okkar, eins og setningin „á meðan það er óuppvaskaður diskur við hliðina á vaskinum eða við eldhúsborðið þá ætla ég að vaska upp" hvernig yrði hún forrituð sem skilyrði inn í while lykkju? Athugum að þarna erum við með rökvirkjann *eða* og því þarf annað hvort að vera skítugur diskur við vaskinn eða á borðinu.

Kóðabútur 6.15: while lykkja óhlutbundin til að sýna rökvirkja

```
while(það er skítugur diskur við vaskinn eða það er skítugur diskur við borðið):
   vaska upp disk
print(allir diskar eru hreinir)
```

Hér sjáum við eitt sem vefst fyrir mörgum, það er að skilyrðið í línu 1 virðist vera óþarflega nákvæmt, til hvers að taka fram skítugur diskur tvisvar? Það er vegna þess að segðinni „það er skítugur diskur við vaskinn" er hægt að svara með já eða nei og sömuleiðis „það er skítugur diskur við borðið". En ef skilyrðið okkar hefði einungis verið „það er skítugur diskur við vaskinn eða borðið" þá lendir vélin í því að fá í hendurnar segð sem hægt er að svara hægra megin við eða og svo „borðið" hinu megin. Hvernig á að svara „eða borðið"? Það er ekki hægt, því að það er ekki skiljanleg spurning. Því þarf að muna að hafa alltaf heila skýra segð sem hægt er að meta sem sanna eða ósanna.

Tökum dæmi um rökvirkjanotkun í skilyrði í lykkju í kóðabút. Þar sem við viljum vita hvort að við séum með líkamshita á eðlilegu bili, eftir að hafa mælt það einu sinni í upphafi og svo mælum við reglulega eftir það.

Kóðabútur 6.16: while lykkja með og rökvirkjanum

```
hiti = 37.0
while(hiti < 37.6 and hiti > 36.0):
   # mælum hitann með þessari óvísindalegu aðferð
   hiti = hiti + 0.5
   print(hiti)
```

```
37.5
38.0
```

Við sjáum að skilyrðið í kóðabút 6.16 er ekki með *eða* heldur *og*, það sem er verið að spyrja er „er hitinn á milli talnanna 36.0 og 37.6?". Þannig að fyrst er spurt er hitinn lægri en 37.6, svo er spurt hvort hann sé hærri en 36.0 og ef svarið við báðum spurningum er já þá hlýtur hitinn að vera á milli þessara talna.

Annað sem má gera við while lykkjur er að koma fram við þær sem skilyrðissetningu sem má fá else klausu aftan við sig sem keyrist þegar skilyrði lykkjunnar verður ósatt.

Kóðabútur 6.17: Að nota else með while

```
x = 5
while(x > 1):
   print("talan er", x, "sem er stærra en 1")
   x -= 1
else:
   print("nú er talan orðin 1 því 1 er ekki stærri en 1 -->", x)

```

```
talan er 5 sem er stærra en 1
talan er 4 sem er stærra en 1
talan er 3 sem er stærra en 1
talan er 2 sem er stærra en 1
nú er talan orðin 1 því 1 er ekki stærri en 1 --> 1
```

6.3 Æfingar

Æfing 6.1 Búið til lykkju sem keyrir 100 sinnum og prentar út númer keyrslunnar. ■

Æfing 6.2 Búið til lista sem inniheldur einungis tölur, lykkjið í gegnum allan listann og leggið saman tölurnar. Prentið út summu listans að keyrslu lokinni. ■

Æfing 6.3 Þetta er sama æfing og 6.2 nema í stað þess að búa til ykkar eigin talnalista eigið þið að finna summu talna frá 0 upp að 1000. Prentið svo út summuna þegar keyrslu lykkjunnar lýkur. ■

Æfing 6.4 Síðasta talnaæfingin með for-lykkjur. Nú ætlið þið að prenta allar þær tölur sem eru á bilinu 0-100 sem eru með þversummu[a] (e. transverse sum) hærri en sex. Þar sem þessi æfing er töluvert flóknari en aðrar verður hún leyst í skrefum og hægt er að kíkja á svörin til að fá fyrst vísbendingu. ■

[a] https://is.wikipedia.org/wiki/%C3%9Eversumma

Æfing 6.5 Síðasta for-lykkju æfingin. Búið til lista með nokkrum strengjum, prentið út alla þá strengi sem innihalda táknið `a`. Ábending, athugið að skoða kóðabút 6.1. ■

Æfing 6.6 Búið til while-lykkju sem keyrir alltaf en er brotin í fyrstu keyrslu. ■

Æfing 6.7 Búið til breytu sem inniheldur einhverja tölu sem er á bilinu 0-10. Búið svo til while-lykkju sem keyrir á meðan sú tala er lægri en 20. Innan lykkjunnar ætlið þið að prenta út töluna og hækka hana svo um 1. ■

Æfing 6.8 Búið til lista sem inniheldur nokkra strengi, en nokkur stök eru strengurinn `"popp"` þ.e. hann kemur nokkrum sinnum fyrir víðsvegar um listann. Nú ætlið þið að búa til while-lykkju sem keyrir á meðan orðið `popp` er enn í listanum (eitt og sér til að einfala málin). Það sem þið gerið svo innan í lykkjunni er að fjarlægja orðið `popp` úr listanum og prenta út breytta útgáfu. Rifjið upp `.pop()` aðferðina úr kafla 4 ásamt `.index()` úr kafla 3 eða flettið upp notkun á `.remove()` á netinu. ■

7. N-dir

Nú ætlum við að kynnast nýrri týpu, hún heitir **n-d** (lesist ennd) eða n-und (e. tuple) [1]. Lykilorð þessarar týpu er **tuple**. Nafnið er komið frá hugmyndinni um tvenndir og þrenndir, nema við vitum ekki hversu mörg stök er verið að hópa saman, þau gætu verið af n fjölda svo við köllum týpuna n-d eða n-und (þá frá tvíund og þríund). Líklega eina orðið í íslensku sem inniheldur ekki sérhljóða. Hér eftir verður týpan kölluð *nd*, þó margir noti n-und.

Hún líkist listum að því leitinu til að margar af sömu aðgerðum sem má gera á lista má gera á ndir. Hún líkist strengjum því að hún er óbreytanleg. Það má ekki bæta við, breyta eða taka út stök eftir að ndin er skilgreind.

Ástæður fyrir því að nota ndir í stað lista er sú að það getur verið hagkvæmara, ndir nota ekki eins mikið minni, eða okkur er umhugað um gagnaheilindi, og svo sjáum við í kafla 10 um hvernig megi fá eina nd í stað margra skilagilda.

7.1 Skilgreining

Við skilgreinum nd með svigum. Athugið að hingað til höfum við notað sviga til að aðgreina segðir og það er vandmeðfarið að átta sig á því hvenær er sviginn stærðfræðilegur (þ.e. einungis fyrir forritarann til að aðgreina samhengi) og hins vegar skilgreining á gögnum af týpunni nd. Aðgreiningin verður augljós þegar við áttum okkur á því að til þess að skilgreina nd þá þurfum við, líkt og með lista, að aðgreina stökin innan ndinnar með kommum. Sjáum í kóðabút 7.1 hvernig má skilgreina ndir og hvernig svigar gera það ekki nema við notum kommur. Þar sjáum við einnig að það eru einungis tvær aðferðir til fyrir týpuna, `.count()` og `.index()`. Hvernig má það vera að týpan líkist listum þegar það eru bara til tvær aðferðir? Var ekki verið að taka fram að það mætti gera margt það sama? Jú, aðgerðir og aðferðir er ekki það sama. Við getum ítrað í gegnum nd, við getum skeytt einni nd aftan við aðra (fáum þá nýja nd en breytum henni ekki), við getum náð í hluta úr ndinni (með hornklofum eins og hlutstrengi eða hluta úr lista).

vísa í kóðabút ú
um eða tölum þa
segðir eru aðgre
með svigum til
ingar, mögulega
að búa það til

Skoðum nú í kóðabút 7.1 hvernig svigar geta annars vegar verið til að afmarka stærfræðilegan

[1] Skoðið endilega orðasafn stærðfræðifélagssins

forgang og hins vegar til að skilgreina nýju týpuna okkar. Tökum sérstaklega eftir notkuninni á innbyggða fallinu `type()` sem auðveldar okkur að skilja hvenær nd verður til.

Kóðabútur 7.1: Ndir skilgreindar

```
a = (3+4)*2
print("a er af taginu", type(a))

b = (1)
print("b er af taginu", type(b))

c = () # þetta verður tóm nd
d = (1,) # þetta verður nd sem inniheldur eitt stak, athugið kommunotkunina
e = (1, 1, 2, 2, 5) # þetta verður nd sem inniheldur 5 stök
```

```
a er af taginu <class 'int'>
b er af taginu <class 'int'>
c er <class 'tuple'> d er <class 'tuple'> og e er <class 'tuple'>
```

Nú gerum við ráð fyrir að eiga ennþá ndirnar `c`,`d` og `e` úr kóðabút 7.1, skoðum þá aðferðirnar tvær sem eru innbyggðar í kóðabút 7.2, ásamt því hvað gerist þegar við skeytum einni nd aftan við aðra, hvernig ítrun með for lykkju er lík því sem við þekkjum með lista og að lokum hvernig megi sækja hlut-nd.

Kóðabútur 7.2: Ndir aðgerðir og aðferðir

```
print(d.index(1))
print(e.count(1))
print(e + d)
print()

for tala in e:
   print(tala)

# Athugum að við megum ekki breyta nd, svo eftirfarandi kóði veldur villu
# c[4] = 3

print(e[1:3])
```

```
0
2
(1, 1, 2, 2, 5, 1)

1
1
2
2
5
(1, 2)
```

7.2 Notkun

Þar sem ndir eru óbreytanlegar er gagnlegt að nota þær til að halda utan um ástand sem við viljum ekki að sé hróflað við. Segjum að það séu ákveðin tengsl á milli tveggja gilda og við viljum halda heilindum þeirra þá væri gott að nota nd. Við getum líka notaðað þær til að spara minni þegar við þurfum litla lista sem þarf bara að nota tímabundið og breytast ekki. Einnig geta þær nýst til að halda utan um breytur sem á svo að nota hverja í sínu lagi seinna. Tökum eftir að vissulega megi útfæra fyrri þrjú, af þessum fjórum atriðum nefndum, með listum.

Skoðum kóðabút 7.3 þar sem við sjáum dæmi um nd sem við viljum að haldist óbreytt þó hún sé skoðuð en við viljum getað úthlutað hverju staki í einhverja breytu (e. unpack) til að nota og skoða án þess að það hafi áhrif á ndina. Við viljum ekki keyra aðferðir á borð við `.sort()` á ndina því að þá breytist hún, við viljum heldur ekki geta haft áhrif á einstaka sætisvísi (skoðið hvaða villu fæst við þá aðgerð með því að keyra línu 10 í kóðabút 7.2), það sem við viljum er létt gagnagrind sem passar upp á gögnin.

Kóðabútur 7.3: Ndir notaðar fyrir það sem þær eru gagnlegar

```
1  notanda_upplysingar = ("valborg", "rosalega gott lykilorð", "netfang@internet.is")
2
3  notandanafn = notanda_upplysingar[0]
4  lykilord = notenda_upplysingar[1]
5  netfang = notenda_upplysingar[2]
6
7  notandanafn, lykilord, netfang = notenda_upplysingar
8
9  print(notandanafn)
10 notandanafn = notandanafn.upper()
11 print(notenda_upplysingar)
```

```
1  valborg
2  ('valborg', 'rosalega gott lykilorð', 'netfang@internet.is')
```

Línur 3-5 og lína 7 eru jafngildar í kóðabút 7.3. Þessi „afpökkun" er læsileg og þægileg leið til að vinna með nd, við sjáum það svo betur þegar við skoðum skilagildi í kafla 10.4 hversu mikilvægt er að kunna á þetta. Takið einnig eftir í úttakinu og þrátt fyrir að breytan `notandanafn` hafi verið uppfærð þá hafði það engin áhrif á ndina. Reynið nú að uppfæra það gildi í ndinni, reynið að setja í staðinn fyrir fremsta stakið þessa nýju breytu og sjáið hvaða villu þið fáið.

Að sjálfsögðu er markmiðið okkar ekki enn sem komið er orðið að því að skrifa kóða í sem fæstum línum mögulegum, en það sem við viljum þó geta gert er að gera kóðann okkar eins læsilegan og mögulegt er með því að nota þær aðgerðir sem Python býður upp á. Jafnvel þó að eini ávinningurinn er að við sjálf skiljum kóðann ennþá þegar við skoðum hann seinna.

7.3 Æfingar

Æfing 7.1 Búið til nd sem inniheldur 3 stök og setjið svo aftasta stakið í breytu. ■

Æfing 7.2 Búið til nd sem inniheldur eingöngu tölur, ítrið í gegnum ndina og prentið út þær tölur sem eru stærri en 100. ■

Æfing 7.3 Búið til nd sem inniheldur tvær tölur, úthlutið svo þeim tveimur tölum í tvær breytur með afpökkun. Geymið svo útkomuna úr því hvort að fremri talan sé stærri en sú seinni og búið til nýja nd þar sem útkomunni er skeytt aftan við upphaflegu ndina. ■

Æfing 7.4 Búið til tóma nd. Búið til lykkju sem keyrir fjórum sinnum og í hvert sinn spyr hún notandann um uppáhaldslitinn sinn. Í hvert sinn sem notandinn er búinn að svara skal endurskilgreina ndina sem það sem hún var áður að viðskeyttu nýja svarinu. Þegar lykkjan hefur lokið keyrslu sinni skulið þið prenta út ndina. ■

8. Orðabækur

Ný týpa sem vil ætlum að nú að fást við heitir **orðabók** (e. dictionary) og lykilorðið hennar er **dict**. Orðabók er orð sem hentar fyrir þýðingu á týpunni í Python en hún er einnig þekkt sem hakkatafla (e. hash table / hash map) í öðrum forritunarmálum. Til þess að búa til orðabók eru notaðir slaufusvigar {}.

Orðabækur eru gagnagrindur eins og listar og ndir, það er þær geyma fyrir okkur gögn af einhverjum týpum. Orðabækur eru þó frábrugnar listum að því leitinu til að þær eru *óraðaðar*, sem þýðir að þær hafa enga sætisvísa sem hægt er að nota[1]. Við getum því ekki sótt gögn í orðabækur með því að vita *hvar* þau eru við þurfum að vita *hver* þau eru.

Þetta er vegna þess að orðabækur eru skipulagðar sem lykla og gildis pör, við finnum þau gildi sem við viljum með því að vita hvaða lykill gengur að þeim. Þetta er ekki ósvipað því að horfa á lyklakippurnar okkar. Lyklarnir eru allir ólíkir. Við getum alltaf reitt okkur á það að sama hvar einhver ákveðinn lykill er þá gengur hann alltaf að sama lásnum, svo ef við þekkjum lyklana okkar getum við auðveldlega náð í þann sem við viljum til þess að opna þann lás sem við viljum hverju sinni.

Lyklarnir verða því að vera ólíkir hverjum öðrum, annars gætum við ekki þekkt þá í sundur og tveir eins lyklar gætu ekki gengið að tveimur mismunandi lásum. Svo lyklar verða að vera aðgreinanlegir.

Orðabókin er mjög öflugt fyrirbæri, og því þess virði að kynna sér vel hvernig þessi týpa virkar. Einnig skoðum við í þessum kafla hvernig má ítra í gegnum orðabækur og hvers vegna það var ágætt að vera búin að skoða ndir áður en við komum að þessari mikilvægu týpu.

8.1 Orðabækur skilgreindar og notaðar

Eins og kom fram í inngangi er gögnum í orðabókum skipt niður á lyklana sem ganga að þeim (sjá ítarefni um aðgreinanleika um hvað má vera lykill og hvað aðgreinanleiki þýðir). Sjáum fyrir okkur lyklakippuna okkar þar sem við erum með stóran ASSA lykil að útidyrahurðinni, lítinn lykil með

[1] Í Python >3.6 eru þær ekki alveg óraðar, stök eru sett inn í ákveðinni röð og helst sú „röðun" þegar orðabókin er notuð fyrir það voru stökin aðgenileg með handahófskenndri röð fyrir minnisbestun.

svörtu plasti að hjólalásnum, kassalaga lykil að útidyrahurðinni hennar ömmu og einn pínulítinn lykil að geymslunni. Við eigum auðvelt með að halda utan um þetta litla lyklasafn og við vitum að hverju allir lyklarnir ganga. En ef við værum nú með 10.000 lykla? Skoðum kóðabút 8.1 til að sjá hvernig megi búa til orðabók sem heldur utan um lyklakippuna sem var lýst hér að ofan. Takið eftir að einungis er unnið með strengi innan orðabókarinnar.

Kóðabútur 8.1: Orðabók kynnt með lyklakippusamlíkingu

```
1 tom_ordabok = {}
2 kippa = {'ASSA': 'útidyrahurðin heima ', 'lítill svartur': 'hjólið', 'kassalaga': 'heima
      hjá ömmu', 'pínulítill': 'geymslulykillinn'}
3
4 print(kippa['lítill svartur'])
```

```
1 hjólið
```

Takið eftir hvernig stökin eru aðgreind, með kommum alveg eins og áður. Nema núna eru stökin tvenndir sem hanga saman með tvípunkti. Við sjáum einnig að til þess að nálgast gögn þá notum við hornklofa eins og áður en við gerum það ekki með sætisvísi heldur gerum við það með lyklinum sem við viljum finna gögnin að. En ef lykill er heiltala þá náum við vissulega í gögnin á þeim lykli með því að nota heiltölu innan hornklofanna (sjá línu 3 í kóðabút 8.2) Í kóðabút 8.1 þá sjáum við hvernig á að búa til tóma bók í fyrstu línu, við gerum ekkert frekar með þessa orðabók í þessum kóðabút en í kóðabút 8.2 sjáum við hvernig má setja stök (lykla og gildis pör) inn í orðabók eftir að hún er skilgreind og við sjáum í kóðabút 8.3 hvaða aðferðir eru til á þessa týpu.

Kóðabútur 8.2: Gögnum bætt við og þau tekin út

```
1 ordabok2 = {1: 'gildi á lykli 1 sem er heiltala', 8: 'gildi sem er á lykli 8', 5: 'takið
      eftir að pörin eru aðgreind með kommu'}
2
3 print(ordabok2[5])
4
5 ordabok2[1] = 'nýtt gildi'
6 ordabok2['nýr lykill'] = 'nýtt gildi'
7 print(ordabok2)
```

```
1 takið eftir að pörin eru aðgreind með kommu
2 {1: 'nýtt gildi', 8: 'gildi sem er á lykli 8', 5: 'takið eftir að pörin eru aðgreind með
      kommu', 'nýr lykill': 'nýtt gildi'}
```

Í kóðabút 8.2 sjáum við hvernig heilartölur geta verið lyklarnir í orðabókinni en við höfum þó aðeins verið að vinna með strengi sem gildi. Í raun eru skorður á því hvað geta verið lyklar en ekki hvað geta verið gildi, við getum geymt hvað sem er sem gildi og við sjáum í kóðabút **??** þegar við notum lista sem gildi.

Ítarefni 8.1 Aðgreinanleiki

Í Python er til `hash()` fall sem skilar „hakki" af því sem því er gefið sem viðfang. Við sáum í kafla 2 að 1 og 1.0 var hægt að reikna með þó þær væru af mismunandi tagi og í kafla 5 sáum við að 1 og 1.0 var jafngilt. Það sem `hash()` gerir er að skila heiltölugildi fyrir viðfangið og þegar við viljum athuga hvort að eitthvað sé jafngilt með rökvirkjanum == erum við að spyrja hvort að fallið skili mismunandi heilum tölum fyrir það sem er sitthvoru megin við rökvirkjann. Í tilfellinu 1, 1.0 og True þá eru þau ekki aðgreinanleg og því ekki hægt að nota þau sem þrjá mismunandi lykla í sömu orðabókinni. Þetta er ástæðan fyrir því að orðabók er oft kölluð hakkatafla.

Prófið ykkur áfram með `hash()` fallið og sjáið hvaða tögum gögnin eru sem þið megið hakka.

Nú höfum við séð grunnvirknina við það að búa til orðabók og ná í gögn á lykil en hvaða aðferðir eru til á þær? Þær eru ekki margar og þess virði að taka nokkrar fyrir sérstaklega vegna þess hve gagnlegar þær eru strax fyrir byrjendur.

`.get()` leyfir okkur að athuga hvort að lykill sé til í orðabók án þess að valda villu, og ef við viljum skila stöðlu gildi ef lykillinn fannst ekki.

`.popitem()` fjarlægir það stak sem síðast var sett inn (fjarlægir eitthvað stak í Python <3.6).

`.pop()` fjarlægir nákvæmlega það stak sem við viljum með því að við gefum upp lykil.

`.items()`, `.keys()` og `.values()` skilar okkur ítranlegum hlut af því sem við viljum geta unnið með, items eru lykla og gildis pör sem ndir, keys eru bara lyklarnir og values eru gildin.

Skoðum aðeins nánar hvernig `items()`, `keys()` og `values()` virka því að við viljum geta ítrað í gegnum orðabækur.

Kóðabútur 8.3: Aðferðir á orðabækur

```
ordabok3 = {1: [1,2,3,4,5], 2: ["strengir", "í", "lista"], "þrír": [-1,-2,-3]}
print("gildin:", ordabok3.values())
print("lyklarnir", ordabok3.keys())
print("ndir af pörum", ordabok3.items())
```

```
gildin: dict_values([[1, 2, 3, 4, 5], ['strengir', 'í', 'lista'], [-1, -2, -3]])
lyklarnir dict_keys([1, 2, 'þrír'])
ndir af pörum dict_items([(1, [1, 2, 3, 4, 5]), (2, ['strengir', 'í', 'lista']), ('þrír',
    [-1, -2, -3])])
```

Höfum ekki óþarfa áhyggjur af úttakinu þar sem stendur dict_ eitthvað. Það sem við þurfum að átta okkur á er að við fáum nokkurs konar lista (e. view) í hvert sinn og að stökin í listunum fást upp úr orðabókinni okkar með útreiknanlegum hætti.

Skoðum nú í næsta undirkafla hvernig megi vinna með þetta.

8.2 Ítrað í gegnum orðabækur

Nú höfum við séð for lykkjur í kafla 6 og hvernig mátti lykkja í gegnum lista í kóðabút **??**. Nú hins vegar þurfum við að fara yfir hvernig í ósköpunum á eiginlega að skoða stak í orðabók á kerfisbundinn máta þegar eitt stak er bæði lykill og gildi.

Þetta útfærsluatriði var gert þannig í Python að þegar óskað er eftir að ítra í gegnum orðabók eru lyklarnir hennar eingöngu teknir fyrir. Hins vegar er hægt að ítra yfir lykla og gildi eða einungis gildin með því að kalla í aðferðirnar sem teknar voru fyrir í lok síðasta undirkafla. Nú eru nöfnin á þessum aðferðum ágætlega lýsandi:

- `.keys()`: við fáum í hendurnar ítranlegan hlut sem inniheldur alla lyklana.
- `.values()` við fáum í hendurnar ítranlegan hlut sem inniheldur öll gildin.
- `.items()` og við fáum í hendurnar ítranlegan hlut sem inniheldur lista af tvenndum (nd með tveimur stökum) þar sem fyrra stakið er alltaf lykilinn og seinna stakið er alltaf gildi hans.

Þannig að til þess að sækja það sem við viljum skoða þurfum við að nota þá aðferð á orðabókina okkar sem okkur hentar hverju sinni. Ef við vildum til dæmis halda utan um bókasafnið okkar með orðabók og vinna með þær upplýsingar úr bókasafninu sem henta hverju sinni gætum við gert það eins og kemur fram í kóðabút 8.4. Við viljum að höfundur sé lykillinn og að gildið sé listi af bókum sem við eigum eftir þann höfund. Svo viljum við geta prentað út nöfn þeirra höfunda ásamt upplýsingum um hversu oft þeir koma fyrir á bókasafninu.

Kóðabútur 8.4: Skoðum hvernig megi ítra í gegnum orðabækur

```
bokasafn = {'Beazley': ['Python Essential Reference'],'Halldór Laxness':
    ['Íslandsklulkka', 'Salka Valka], 'Auður Haralds': ['Hlustið þér á Mozart',
    'Læknamafían', 'Hvunndagshetjan'] }

for hofundur in bokasafn:
   if len(bokasafn[hofundur]) > 5:
      print('Á bókasafninu eru til fleiri en fimm bækur eftir höfundinn', hofundur)
   elif(len(bokasafn[hofundur]) > 2):
      print('Á bókasafninu eru til fleiri en tvær bækur en þó innan við sex, eftir
     höfundinn', hofundur)
   elif(len(bokasafn[hofundur]) > 1):
      print('Á bókasafninu eru til tvær bækur eftir höfundinn', hofundur)
   elif(len(bokasafn[hofundur]) > 0):
      print('Á bókasafninu er til ein bók eftir höfundinn', hofundur)
   else:
      print('Á bókasafninu er ekki til nein bók eftir höfundinn', hofundur)
```

```
Á bókasafninu er til ein bók eftir höfundinn Beazley
Á bókasafninu eru til tvær bækur eftir höfundinn Halldór Laxness
Á bókasafninu eru til fleiri en tvær bækur en þó innan við sex, eftir höfundinn Auður
    Haralds
```

Í kóðabút 8.4 var engum aðferðum beitt svo að við fengum það sem er staðlað að vinna með, einungis lyklana. Takið eftir að þegar kallað er í `bokasafn[hofundur]` er verið að biðja um gildið sem tilheyrir þessum tiltekna höfundi, breytan `hofundur` er hlaupandi breytan sem rúllar í gegnum lyklana úr orðabókinni. Við sjáum á úttakinu að fyrsta stakið sem `hofundur` fær úthlutað er Beazley og `bokasafn['Beazley']` er metið sem `['Python Essential Reference']` og svo er kallað á `len` (innbyggt fall sem skilar okkur lengd hluta eða fjölda sætisvísa) sem segir okkur að það sé 1 stak í listanum sem er gildið á lyklinum. Þá rúllum við í næsta hluta skilyrðissetningarinnar því að 1 er vissulega ekki stærra en 5, við fáum sanngildi þegar spurt er hvort að 1 sé stærra en 0 og því fáum við úttakið: Á bókasafninu er til ein bók eftir höfundinn Beazley. Svo gerist það sama aftur fyrir næsta höfund í röð lykla.

Í næsta kóðabút skoðum við svo hvernig eigi að fara að því að skoða bara bókalistana burt séð frá því hverjir höfundarnir eru, þetta gerum við með sömu `bokasafn` breytunni. Svo við gerum ráð fyrir að hún sé enn aðgengileg í kóðabút 8.5. Markmiðið þar er ekki að skoða bækur eftir höfundum heldur prenta út nöfnin á öllum bókum sem eru af nægilega löng. Vegna þess að gildin eru listar af bókum þá getum við ítrað í gegnum hvern fyrir sig og þá hreiðrað aðra for lykkju inn í þá lykkju sem sér um að ítra í gegnum bókasafnið okkar.

Kóðabútur 8.5: Ítrun í gegnum orðabækur með .values()

```
for bokalisti in bokasafn.values():
   for bok in bokalisti:
      if(len(bok) > 15):
         print(bok)
```

```
Python Essential Reference
Hlustið þér á Mozart
```

Nú er ekki ýkja frábært að vera með hreiðraðar for lykkjur, þær eru gífurlega tímafrekar og það sem þær gera væri oft hægt að leysa á betri máta. En eins og fram hefur komið áður erum við að reyna að átta okkur á því hvernig hlutir virka, við erum ekki að reyna að besta (e. optimize).

Prófið ykkur áfram með kóðann í kóðabút 8.5, sjáið hvar breytunar eru aðgengilegar með því að prenta þær út og sjáið hvað breyturnar innihalda hverju sinni með útprentunum. Prófið einnig að breyta til, og sjá hvort þið áttið ykkur á því hvað kemur út.

Í næsta kóðabút sjáum við svo hvað við gerum til að geta unnið með bæði lykil og gildi. Notkun á `.items()` hefði mögulega sparað okkur smá hausverk í kóðabút 8.4 og gert þann kóða læsilegri. Tökum eftir að þar sem `.items()` skilar nd þá getum við annað hvort notað eitt breytuheiti til að taka við allri ndinni eða við getum úthlutað hverju staki úr ndinni í sína eigin breytu. Sem er það sem er gert í línu 1 í kóðabút 8.6, við munum að aðferðin skilar nd þar sem lykillinn kemur fyrst og svo kemur gildið. Því er breytan `hofundur` á undan í röðinni, breyturnar eru svo aðgreindar með kommu og þá kemur `bokalisti`. Aftur gerum við ráð fyrir að sama bókasafnið sé okkur aðgengilegt.

Kóðabútur 8.6: Ítrun í gegnum orðabækur með .items()

```
1  for hofundur, bokalisti in bokasafn.items():
2     # ef bókalistinn er ákveðið langur þá langar okkur að prenta út nafnið á höfundinum
3     if(len(bokalisti) > 5):
4        print(hofundur, "er mjög vinsæll höfundur")
5     elif(len(bokalisti) > 2):
6        print(hofundur, "er frekar vinsæll höfundur")
7     elif(len(bokalisti) > 1):
8        print(hofundur, "gæti verið vinsælli")
9     elif(len(bokalisti) == 1):
10       print(hofundur, "er vissulega til staðar")
11    else:
12       print(hofundur, "á ekki tiltall til einnar bókar í þessu bókasafni")
```

```
1  Beazley er vissulega til staðar
2  Halldór Laxness gæti verið vinsælli
3  Auður Haralds er frekar vinsæll höfundur
```

Þetta eru mjög einföld dæmi en þau sýna það helsta sem þarf til þess að geta gert frekari tilraunir og leyst hin ýmsu verkefni. Eins og áður þá næst árangur í forritun með því að gera tilraunir.

Skoðum nú næst kóðabút 8.7 þar sem rennt er í gegnum orðabók þar sem gildin eru orðabækur, takið sérstaklega eftir breytunni `upplysingar` og hvað hún gerir mikið fyrir okkur. Við sjáum einnig í línu 15 að þar er innri lykkja sem er einungis keyrð fyrir þá höfunda sem eru með nógu háa meðaleinkunn.

Kóðabútur 8.7: Orðabók sem inniheldur orðabók sem gildi

```
1  itarlegt_bokasafn = {"Beazley": {'lesnar': 1, 'olesnar': 0, 'medaleinkunn': 5,'baekur':
       ['Python Essential Reference 4th ed'], 'besta bok': 'Python Essential Reference 5th
       ed' },
2     'Halldór Laxness': {'lesnar': 1, 'olesnar': 1, 'medaleinkunn': 3,'baekur':
       ['Íslandsklulkka', 'Salka Valka'], 'besta bok': 'Vefarinn mikli frá Kasmír'},
3     'Auður Haralds': {'lesnar': 3, 'olesnar': 0, 'medaleinkunn': 4,'baekur': ['Hlustið þér
       á Mozart', 'Læknamafían', 'Hvunndagshetjan'], 'besta bok': "Hvunndagshetjan"}
4  }
5  for hofundur, upplysingar in itarlegt_bokasafn.items():
6     print(hofundur)
7     if(upplysingar['olesnar'] > 0):
8        print('Þú átt eftir að lesa einhverja af eftirfarandi bókum', upplysingar['baekur'])
9     if len(upplysingar['baekur']) < 2:
10       print('Það virðist vanta fleiri bækur eftir', hofundur)
11    if upplysingar['besta bok'] not in upplysingar['baekur']:
12       print('Þig vantar bestu bókina eftir höfundinn', hofundur, "sem er",
       upplysingar['besta bok'])
13    if(upplysingar['medaleinkunn'] > 3):
14       print(hofundur, 'er í miklu uppáhaldi og þú átt eftirfarandi bækur eftir
       viðkomandi:')
15       for bok in upplysingar['baekur']:
16          print (bok)
```

```
17      print()
```

```
Beazley
Það virðist vanta fleiri bækur eftir Beazley
Þig vantar bestu bókina eftir höfundinn Beazley sem er Python Essential Reference 5th ed
Beazley er í miklu uppáhaldi og þú átt eftirfarandi bækur eftir viðkomandi:
Python Essential Reference 4th ed

Halldór Laxness
Þú átt eftir að lesa einhverja af eftirfarandi bókum ['Íslandsklulkka', 'Salka Valka']
Þig vantar bestu bókina eftir höfundinn Halldór Laxness sem er Vefarinn mikli frá Kasmír

Auður Haralds
Auður Haralds er í miklu uppáhaldi og þú átt eftirfarandi bækur eftir viðkomandi:
Hlustið þér á Mozart
Læknamafían
Hvunndagshetjan
```

Í þessum síðasta kóðabút er nóg um að vera sem ætti að vera gott veganesti í æfingar þessa kafla.

8.3 Æfingar

Æfing 8.1 Búið til tóma orðabók, bætið svo við lykli og gildi. ■

Æfing 8.2 Búið til tóma orðabók. Skrifið svo for lykkju þannig að þið bætið tölum frá 0 og upp að n (að eigin vali) sem lykla og sömu tölur í öðru veldi sem gildi, t.d ef n er 5 þá liti orðabókin svona út: 0: 0, 1: 1, 2: 4, 3: 9, 4: 16 ■

Æfing 8.3 Búið til orðabók með þremur stökum, fjarlægið einhver tvö þeirra. ■

Æfing 8.4 Búið til orðabók með þremur stökum þar sem gildin eru listar af tölum, ítrið í gegnum orðabókina og prentið út það stak sem er minnst af öllum (einungis eina tölu) ef það er þó minna en talan 0, annars skal prenta út töluna 0. Athugið að nota min() fallið. ■

Æfing 8.5 Búið til orðabók þar sem gildin eru listar af strengjum. Ítrið í gegnum orðabókina og bætið 'x' aftan við alla strengi í listunum sem eru í gildum orðabókarinnar. Athugið hér að nota `type()` fallið. ■

Æfing 8.6 Krefjandi æfing
Búið til orðabók sem inniheldur spurningar sem lykla og rétt svör við spurningunum sem gildi. Semjið að minnsta kosti 5 spurningar, svörin þurfa að vera rétt eða rangt (til að einfalda hlutina töluvert). Búið til breytu sem heldur utan um stig notandans sem byrja í 0, hækkið þessa tölu þegar notandinn svarar rétt en breytið henni ekki annars. Búið til lykkju sem fer í gegnum allar spurningarnar og spyr notandann að þeim. Þegar notandinn hefur svarað þá athugið þið hvort svarið sé rétt. Ef það er rétt hækkiði einkunnina og látið notandann vita að svarið var rétt ef það er rangt látiði notandann vita að svarið var rangt.

Athugið að það er gott að staðla svar notandans t.d. með .lower() eða annarri sambærilegri aðferð. Þegar spurningarnar eru búnar þá prentiði út einkunn notandans og ef öll svörin voru rétt þá prentiði og þú fékkst hæstu einkunn". ■

Æfing 8.7 Krefjandi æfing
Búið til orðabók sem heldur utan um sveitarfélög á höfuðborgarsvæðinu sem lykla og íbúafjölda þeirra sem gildi. Spyrjið notandann um tvö mismunandi sveitarfélög á höfuðborgarsvæðinu, gefið notandanum upp hversu mikill fjöldi býr þar samanlagt. ■

9. Mengi

Mengi (stundum kölluð sett) (e. set) eru týpa sem geymir óraðað safn af gögnum án tvítekninga, þau eru ein af fjórum innbyggðum gagnagrindum í Python (listar, ndir, orðabækur eru hinar) og geta þau geymt gögn af hvaða týpu sem er. Lykilorðið þeirra er **set**. Mengi eru skilgreind með slaufusvigum og eru stök þeirra aðgreind með kommum, ólíkt orðabókum þá eru engin lykla og gildispör sem hanga saman með tvípunkti og því ruglast vélin ekki á þessum tveimur týpum. Eins og orðabækur eru óraðaðar, þá er ekki hægt að nota vísa til þess að segja hvar eitthvað stak er í mengi.

Mengi þessi eru eins og mengi sem við könnumst við í stærðfræði, þar sem hvert stak kemur þó aðeins fyrir einu sinni. Við getum framkvæmt ýmsar stærðfræðilegar aðgerðir á þau ásamt hefðbundnum aðgerðum til að bæta við eða fjarlægja stök, hins vegar er ekki hægt að breyta staki sem er nú þegar komið í mengið.

Skoðum kóðabút 9.1 til þess að sjá hvernig mengi eru skilgreind og hvernig megi nota lykilorðið til að búa til mengi fyrir okkur úr gögnum.

Kóðabútur 9.1: Mengi skilgreind

```
# Fyrsta mengið okkar inniheldur nokkrar tölur
mengid_mitt = {1,2,3,4}
print(mengid_mitt)
# úttakið verður
# {1, 2, 3, 4}

# en til þess að búa til tómt mengi þarf að nota lykilorðið
tomt_mengi = set()

# því að þetta er tóm orðabók:
ekki_mengi = {}
```

```
    hjólið
```

9.1 Tvítekning

Tvítekning í mengjum er ekki leyfileg og því ágætt að nota mengi til þess að fjarlægja tvítekningar úr gögnunum okkar. Ef við tökum fyrir orðið 'halló' og gerum mengi úr því með `set('halló')` þá fengjum við mengi sem innihéldi `'h', 'a', 'l', 'ó'`. Stafurinn 'l' kemur tvisvar fyrir í strengnum en hann kemur einu sinni fyrir í menginu af strengnum. Sjáum kóðabút 9.2 hvernig við fáum ekki út tvítekningar sama hvernig við reynum. Takið eftir úttalinu þar sem stafirnir koma í einhverri röð, sú röð er ekki heilög þar sem þetta er óraðað gagnatag og þessi röðun verður ekki endilega eins við aðra keyrslu. En hvernig sem þessi röðun er þá kemur sama táknið ekki fyrir tvisvar. Það er eitt bil, eitt litla v og eitt stóra V, og svo framvegis.

Kóðabútur 9.2: Mengi skilgreind

```
1 mengid_mitt = {1,2,3,4, 1, 2, 3, 4}
2 print(mengid_mitt)
3
4 strengur = "Valborg Sturludóttir vinsamlegast"
5 print(set(strengur))
```

```
1 {1, 2, 3, 4}
2 {'r', 'ó', 'n', 'm', 'v', 'S', 't', 'b', 'u', ' ', 'd', 'l', 'g', 'a', 'i', 'V', 'e',
      'o', 's'}
```

9.2 Aðgerðir

Aðferðir sem hægt er að beita á mengi er að bæta við staki, **add**(), fjarlægja stak, **remove**() og uppfæra mengið með lista eða mengi til þess að geta sett inn mörg stök í einu, **update**(). Engin þessara aðferða gerir okkur kleyft að eiga tvö eins stök í menginu. Tvítekning er ekki liðin, sama hvernig við reynum að komast framhjá henni.

Kóðabútur 9.3: Mengja aðferðir

```
1 mengid_mitt = {1,2,3,4, 1, 2, 3, 4}
2 mengid_mitt.add(3)
3 print(mengid_mitt)
4 mengid_mitt.remove(4)
5 print(mengid_mitt)
6 mengid_mitt.update([1,2,3,3,2])
7 print(mengid_mitt)
8 mengid_mitt.remove(5)
```

```
1 {1, 2, 3, 4}
2 {1, 2, 3}
3 {1, 2, 3}
4
5 ---------------------------------------------------------------------------
6 KeyError                                  Traceback (most recent call last)
7 <ipython-input-99-690bf73c0733> in <module>
8 6 mengid_mitt.update([1,2,3,3,2])
9 7 print(mengid_mitt)
10 ----> 8 mengid_mitt.remove(5)
11
12 KeyError: 5
```

Hér sjáum við lyklavillu þar sem við reyndum að fjarlægja stakið 5 úr mengi sem innihélt það ekki, við sjáum að villan á sér stað í línu 8 og að lyklavillan verður útaf 5, svo við getum auðveldlega lagað þessa villu. Reynið nú að laga villuna og gerið tilraunir með þessar þrjár aðferðir.

Ítarefni 9.1 Mengjafræði og tákn
Stærðfræðilegar aðgerðir sem hægt er að gera er t.d. að finna sniðmengi eða sammengi[a], eða einhverja aðra sniðuga blöndu. Þetta er hægt að gera með innbyggðum aðferðum sem taka við tveimur mengjum og skila einu mengi til baka, en það er einnig hægt að gera með því að beita reiknivirkjum á milli tveggja mengja sem má sjá í opinberri skjölun Python. Þar sést nöfnin á innbyggðu aðferðunum fyrir mengjaaðferðirnar ásamt þeim reiknivirkjum sem má nota í staðinn.

[a] Mengi á ensku wikipedia

Mengi reynast vel við að vinna með gögn eins og símaskrár eða tölvupóstföng því við viljum ekki tvítekningar og þegar á að sameina símaskrár eða tölvupóstföng með ákveðnum reglum er gott að vita að hægt sé að beita þessari týpu. Hún er einföld og þægileg, en ræður við grífurlega leiðigjarna útreikninga og því skynsamlegt að kynna sér hana til að geta auðveldlega leyst verkefni sem annars væru mikil handavinna.

9.3 Æfingar

Æfing 9.1 Búið til mengi sem er tómt. ■

Æfing 9.2 Búið til lista af strengjum, búið svo til mengi út frá þeim lista. ■

Æfing 9.3 Búið til lista af tölunum 1, 2, 3, 4, 5 og einn tóman lista. Ítrið þá í genum talnalistann með for lykkju þannig að þið rúllið frá 0 upp í töluna 3 með range fallinu svo að þið getið skoðað eitt stak hægra megin við þá tölu sem við eruð að skoða.

Það sem þið gerið í lykkjunni ykkar er að skoða töluna í staki *i* (hlaupandi stakið) og töluna í staki *i+1* og leggið þær saman og setjið inn í tóma listann sem var skilgreindur fyrir ofan lykkjuna.

Þegar lykkjan er búin að keyra eruð þið komin með tvo talnalista. Fyrri listann gerið þið að mengi sem þið uppfærið svo og setjið uppfærða listann inn sem viðfang.

Athugið svo hvort að mengið sé tíu stök að lengd. ■

Æfing 9.4 Búið til tvö mengi sem innihalda einhverjar tölur og finnið sniðmengið þeirra með því að nota reiknivirkjann `&` ■

10. Föll

Föll (e. functions), eins og lykkjur, eru kóðabútar sem má keyra oft. Þau líkjast hins vegar frekar skilgreiningum eða uppskriftum frekar en lykkjum þar sem það þarf að *nota* þau til þess að þau geri eitthvað ekki bara búa þau til.

Ágætar samlíkingar fyrir föll eru til dæmis stærðfræðilegar skilreiningar, uppskriftir eða réttir á matseðli.

- Stærðfræðilega skilgreining á hring er eftirfarandi: „Hringur er safn punkta í gefinni fjarlægð frá ákveðinni miðju" sem þýðir að til þess að búa til hring þarf einhverja miðju og teikna svo punkta í einhverjum tilteknum radíus frá þeirri miðju. Skilgreining þessi réttir okkur ekki hring með miðju í punkti (0,0) og radíus 3 þegar við setjum hana fram. En vegna þess að við eigum hana getum við notað hana til þess að búa til alla þá hringi sem okkur henta.
- Uppskrift í matreiðslubók er ákveðin runa af aðgerðum sem þarf að framkvæma, eins og kom fram í kafla 1.4 um hnetusmjörssamlokuna, það að skrifa niður röð aðgerðanna og það sem þarf til er ekki jafngilt því að framkvæma aðgerðirnar og enda með matinn í höndunum
- Réttur á matseðli er skilreindur á ákveðinn hátt, með ákveðnu meðlæti og þess háttar. Við gætum þó viljað gera breytingu á þessari skilgreiningu til þess að fá mat sem er okkur meira að skapi en það sem kokknum datt í hug. Það gerum við með því að biðja um skilgreininguna á réttinum nema með breytingum.

Höfum þessar samlíkingar í huga þegar kemur að því að skrifa og beita föllum, því það hjálpar að átta sig á því strax að þegar við skilgreinum föll þá búum við til uppskrift sem hægt er að fylgja án þess þó að biðja vélina um að fylgja þeim.

Við höfum séð föll áður, eins og:

- print("fyrsta viðfangið", "næsta viðfang sem fallið tekur", "print er sér á báti, því það tekur við svo mörgum viðföngum")
- len("breyta sem ég vil vita lengdina á")
- range(0,50)
- type("breyta sem viðfang sem ég vil vita týpuna á")

Þetta eru innbyggð föll[1], það að þau séu innbyggð þýðir að við getum notað þau án þess að ná í

[1] Python skjölunin inniheldur lista og upplýsingar um öll innbyggð föll

einhvern annan kóða (sjá kafla 11) og eru þau flest svo gagnleg að ákveðið var að gefa notendum auðveldara aðgengi að þeim.

Við höfum þó séð fleiri föll sem eru aðferðir (e. methods), munurinn liggur í því að aðferð er hengd aftan á hlut með punkti og er fall sem keyrir á þann hlut en fall keyrir þegar kallað er í það og óþarfi er að hengja það við eitthvað annað. Allar aðferðir eru föll, ekki öll föll eru aðferðir.

Til dæmis sáum við aðferðirnar .capitalize() á strengi og .sort() á lista.

Föll (e.function) og aðferðir (e. methods) eru aðgreinanlegar að þessu leiti, annað er skilgreint og virkar eins og það á að gera fyrir þau gögn sem þau eiga að virka á en aðferðir er eitthvað sem er fast við hlut og eiga að verka á þann hlut.

Við sjáum svo í öðrum hluta bókarinnar hvernig við skilgreinum aðferðir.

10.1 Tilgangur falla

Eins og með allt annað sem við lærum er gott að vita hvers vegna við erum á annað borð að læra um það. Ástæðan fyrir því að við viljum læra um föll er að þau eru eitthvað það öflugasta sem við beitum í forritun, skoðum eftirfarandi lista til að skilja hvers megnug þau eru:

- Við getum endurnýtt föll í stað þess að skrifa upp sama kóðann á bakvið þau á mörgum stöðum.
- Við getum gert föll aðgengileg út fyrir skjalið þar sem við skilgreindum þau.
- Við getum unnið með inntak frá notanda á skilvirkan máta.
- Föll halda utan um einhverja tiltekna virkni sem við viljum hafa aðgang að og eru skilvirk leið til að afmarka virkni.

Við höfum hingað til ekki fengist við meira en að átta okkur á grunnvirkni í forritun með hjálp Python en nú erum við komin á þann stað að við getum farið að leysa flókin vandamál.

10.2 Að skilgreina föll

Til þess að skrifa föll þurfum við að læra nýtt lykilorð, **def**. Það stendur fyrir *define* eða að skilgreina, þar sem við erum með því að búa til ákveðna skilgreiningu sem vélin getur svo notað til þess að framkvæma aðgerðir.

Það næsta sem þarf er að búa til nafn á fallið, nafnið er það sem vð notum til þess að beita fallinu okkar eftir að hafa skilgreint það. Alveg eins og með aðrar breytur þá megum við ekki nota föll fyrr en búið er að skilgreina þau.

Þegar það er komið getum við byrjað að forrita virkni fallsins okkar, allt sem er a.m.k. einum inndrætti innar en def lykilorðið tilheyrir fallinu okkar.

Í kóðabút 10.1 sjáum við hvernig á að búa til skilgreininguna og svo hvernig á að beita fallinu sem við bjuggum til, við ætlum að skoða minni úr forritun[2] og prenta út einfalda setningu.

Takið eftir að fallið er skilgreint og svo er það sem er inndregið undir því það sem fallið gerir, fyrir neðan skilgreininguna (ekki lengur inndregið undir henni) er svo kallað í fallið með því að skrifa nafnið á því og tóma sviga fyrir aftan nafnið. Svigarnir eru nauðsynlegir því að þannig segjum við vélinni að við séum að kalla í fall.

Kóðabútur 10.1: Föll skilgreind

```
1 def prentunarfall():
2     print("Halló Heimur!")
3
4 prentunarfall()
```

[2] Venjulega það fyrsta sem gert er í nýju forritunarmáli er að prenta út á staðalúttak "Halló Heimur!", þetta er skemmtileg hefð sem óþarfi er að gera miklar breytingar á.

```
Halló Heimur!
```

Ástæðan fyrir því að svigarnir eru tómir í línu 4 er vegna þess að þeir eru tómir í línu 1 í skilgreiningunni. Prófið að setja eitthvað á annan staðinn en ekki hinn og keyra svo kóðann.

10.3 Viðföng

Nú höfum við séð að hægt er að búa til skilgreiningar á föllum, en tökum eftir að í kóðabút 10.1 þá eru tómir svigar fyrir aftan nafnið á fallinu. Þessir svigar eru ekki þarna að ástæðulausu og þeir eru ekki tómir í þessum kóðabút að ástæðulausu heldur.

Það sem fer inn í svigana eru svo kölluð *viðföng* (e. arguments), viðföngin skiptast í tvær tegundir **stöðubundin** (e. postitional) og **sjálfgefin** (e. named). Hægt er að nota bæði í bland og eina viðmiðið er að gefa kost á þeim viðföngum sem notandinn ætti að fá eitthvað um að segja.

Kóðabútur 10.2: Föll með viðföngum

```
def prentunarfall(vidfang):
    print("Halló", vidfang, "!")

prentunarfall("Valborg")
```

```
Halló Valborg!
```

Annað sem mikilvægt er að átta sig á er að viðföng fá breytuheiti sem eru aðeins aðgengileg innan fallsins en ekki utan þess. Eins og breytan `vidfang` er aldrei formlega skilgreind svo við sjáum, allt í einu er hún bara komin þarna inn í svigann í línu 1 og strax notuð í línu 2. Það er í rauninni ekki keyrsluröðin, það sem gerist er að vélin fær skilgreininguna á `prentunarfall` og að þeirri skilgreiningu fylgir einhver staðhaldari sem mun seinna fá eitthvað gildi. Sem er það sem gerist í línu 4, við gefum breytunni gildið `"Valborg"`.

Við ætlum þá að skoða nýtt hugtak áður en við skoðum stöðubundin og sjálfgefin viðföng til þess að átta okkur á hvar þessi staðhaldari er í raun til.

10.3.1 Gildissvið

Gildissvið (e. scope) skiptis í staðvært (e. local) og víðvært (e. global). Það sem gildissvið þýðir er hvar eitthvað sé aðgengilegt. Ef við búum til Jupyter vinnubók eða .py skjal þar sem við skilgreinum breytuna x er sú breyta ekki aðgengileg í öðru skjali. Hins vegar ef við búum til breytu í vinnubók í einhverri sellu, með engum inndrætti, er sú breyta hluti af viðværu gildissviði og aðgengileg öllum sellum og allri virkni sem við viljum beita þessari breytu í. En þegar við skilgreinum föll þá förum við inn á staðvært gildissvið sem þýðir að þegar kallað er í breytu byrjar vélin á að skoða hvort að breytan sé skilgreind innan þess sviðs, ef ekki þá notar hún víðværa gildissviðið. En ef okkur langar að nota breytu sem var skilgreind innan einhvers falls (einhvers staðværs gildissviðs) þá höfum við ekki aðgang að henni í hinu víðværa gildisssviði.

Þetta virðist ótrúlega flókið í svona tæknilegu og löngu máli en skoðið kóðabút 10.3 og skoðið hvaða x er verið að vísa í hverju sinni. Það x sem er skilgreint í línu eitt er hluti af víðværu gildissviði og það sem er skilgreint innan fallsins í línu 3 er hluti af staðværu gildissviði og þetta eru því tvö mismunandi x sem hafa ekki áhrif hvort á annað.

Kóðabútur 10.3: Gildissvið

```
x = 20
def gildissvids_prufa(x):
    x = x + 20
    print(x)
```

```
5
6  gildissvids_prufa(10)
7
8  def prufa_tvo():
9     print(x)
10
11 prufa_tvo()
```

```
1  30
2  20
```

Það má líta á þetta eins og að horfa á gosbrunn, þar sem víðværa gildissviðið er þar sem vatnið kemur upp efst í brunninum og svo fellur það niður í staðværa gildissviðið sem er neðri hluti brunnsins. Allt sem er til í efri hlutanum getur neðri hlutinn fengið en það sem er í neðri hlutanum fer ekki upp (í þessari samlíkingu ætlum við að horfa framhjá innri virkni gosbrunnsins og sjá bara fyrir okkur hvernig er að horfa á fallegan gosbrunn sem er á tveimur eða fleiri hæðum).

Ástæða þess að það er mikilvægt að nefna gilssvið að svo stöddu er vegna þess að byrjendur vilja oft ruglast á breytunotkun með þessum hætti og halda að viðföng séu skilreindar breytur sem hægt sé að láta fallið hafa aftur, þegar það er líkara hugmyndinni um breytuna en ekki breytan sjálf.

Þetta skýrist þegar við skoðum stöðubundin viðföng.

10.3.2 Stöðubundin viðföng

Stöðubundin viðföng (e. postitional arguements) fá nafn og röðun þegar þau eru sett í skilgreiningu á falli. Nöfnin á þeim lúta sömu lögmálum og nafnavenjur sem við höfum séð á breytum og er best að hafa nöfnin lýsandi fyrir virkni þeirra.

Þegar við köllum í fall sem skilgreint er með einu viðfangi og við gefum því engin viðföng þá fáum við villu, villumeldingin sem við fáum upp segir vantar eitt stöðubundið viðfang. Í kóðabút 10.4 sjáum við þessa villu og í kóðabút 10.5 sjáum við hvernig má komast hjá þessari villu og þá hvernig viðföngin eru í raun stöðubundin, svo loks í kóðabút 10.6 sjáum við hvernig breytuheitin geta hjalpað okkur við notkun viðfanganna.

Kóðabútur 10.4: Villumelding fyrir ranga notkun á viðföngum

```
1  def fall_sem_tekur_vid_streng(strengur):
2     print(strengur.capitalize())
3  fall_sem_tekur_vid_streng()
```

```
1  ---------------------------------------------------------------------------
2  TypeError                                 Traceback (most recent call last)
3  <ipython-input-128-0e947c4208c2> in <module>
4  1 def fall_sem_tekur_vid_streng(strengur):
5  2         print(strengur.capitalize())
6  ----> 3 fall_sem_tekur_vid_streng()
7
8  TypeError: fall_sem_tekur_vid_streng() missing 1 required positional argument: 'strengur'
```

Hér sjáum við að breytuheitið okkar hjálpar okkur við að sjá hvað það er sem við eigum að setja inn, það er einhvers konar stengur. Það er ekki breytan strengur því að þá fengjum við nafnavillu (við eigum enga breytu sem heitir strengur).

Við þurfum þá að kalla í fallið og setja eitthvað inn í svigann þegar við köllum í það, eins og 'Halló Heimur!' sem myndi þá prenta út "Halló heimur!".

Nú viljum við skoða röðunina á viðföngunum ef fallið tekur við nokkrum. Athugum svo að það skiptir máli í hvað röð viðföngin eru sett inn, í línum 4-6 í kóðabút 10.5 er verið að setja inn fyrir `a`, `b` og `c` en aldrei í sömu röð og því er úttakið alltaf mismunandi.

Kóðabútur 10.5: Stöðubundin viðföng kynnt

```
def fall(a,b,c):
   print(a**b/c)

fall(1,2,3)
fall(2,3,1)
fall(3,1,2)
```

```
0.3333333333333333
8.0
1.5
```

Athugum að í kóðabút 10.5 þá er hvorki nafnið á fallinu né viðfanga þess sérlega lýsandi. Nafnið á fallinu gefur ekki til kynna hvað það gerir og nöfnin á viðföngunum segja ekkert til um hvernig þau verða notuð eða af hvaða týpu þau eiga að vera. Þetta fall væri mögulega nothæft fyrir okkur sjálf, en um leið og annað fólk á að fara að nota kóðann okkar þá er eins gott að venja sig af því að nota svona ógegnsæjar nafnavenjur. Betra væri, fyrir þessa tilteknu formúlu að finna eitthvað nafn á hana eða nefna fallið eftir nákvæmri virkni formúlunnar og nefna svo viðföngin eftir því hvar þau eru sett inn í formúluna.

Hingað til hafa öll föllin okkar verið að prenta út, við viljum geta gert eitthvað annað en það. Við sjáum í kafla 10.4 hvað hægt er að gera annað en að prenta bara út.

Lítum á kóðabút 10.6 til að sjá hvernig betur mætti fara með skilgreininguna úr kóðabút 10.5. Takið eftir að úttakið er að sjálfsögðu það sama því að kallað er í fallið með sömu tölum, það eina sem breyttist er að nú vitum við hvað er að fara inn fyrir hvað í útreikningunum.

Kóðabútur 10.6: Stöðubundin viðföng með skýrari breytuheitum

```
def hefja_i_veldi_og_deila(grunntala,veldisvisir,deiling):
   print(grunntala**veldisvisir/deiling)

hefja_i_veldi_og_deila(1,2,3)
hefja_i_veldi_og_deila(2,3,1)
hefja_i_veldi_og_deila(3,1,2)
```

```
0.3333333333333333
8.0
1.5
```

Ef við rifjum upp hugtakið gildissvið úr fyrri undirkafla og skoðum hvernig það á við um kóðabút 10.6 þá sjáum við að fallið heitir `hefja_i_veldi_og_deila` og það tekur við þremur viðföngum sem verða að vera sett inn í þeirri röð sem fallið kallar eftir þeim, fyrst grunntöluna svo töluna sem á að nota sem veldisvísi og svo loks töluna sem á að deila með.

Nú er við hæfi að taka fram hvers konar byrjenda mistök eru algeng hérna og ef meiri skilningur værir fyrir hendi á hvernig gildissvið virka myndu þessi mistök eiga sér sjaldnar stað. Það er að breyturnar grunntala, veldisvisir og deiling eru hluti af staðværu gildissviði þessa falls, neðri hluta brunnsins sem efri hlutinn getur ekki sótt vatn úr. Þess vegna getum við ekki kallað í fallið svona: `hefja_i_veldi_og_deila(grunntala,veldisvisir,deiling)` (það er skrifað inn nöfnin á viðföngunum eins og við eigum þau til sem breytur) því að þá erum við að biðja víðværa gildissviðið (efri hluta brunnsins) um að finna hjá sér einhverjar breytur sem heita þetta til þess að setja inn í staðinn fyrir þessar skilgreiningar.

Ef við horfum aftur til skilgreiningar hrings og myndum búa til fall sem lýsir því hvernig eigi að teikna hring með ákveðna miðju og tiltekinn radíus. Nú ef okkur langar til að fá einhvern hring í hendurnar getum við ekki sagt við vélina láttu okkur hafa hring með radíus `radius` nema að sú breyta hafi þegar fengið gildið sem við ætluðum að nota. Vélin, ef við manngerum hana örlítið,

myndi þá segja „það er það sem þú átt að segja mér, hvað radíus er, ég veit ekkert hvað það er!" Til þess að nota fallið þurfum við að gefa annað hvort upp gögn af týpunni sem um var beðið eða breytu sem er aðgengileg utan fallsins (annað hvort úr víðværu gildissviði eða víðara staðværu gildissviði) sem inniheldur gögn af týpunni sem um var beðið.

Tölvan reynir ekki að hafa vit fyrir okkur, hún lagar ekki inntakið þegar við setjum það í augljóslega ranga röð. Hún annað hvort vinnur með vitlausa inntakið okkar og við fáum í hausinn eitthvað úttak sem við skiljum ekki eða við fáum villu. Kíkið á kafla 14 til þess skoða hvernig megi taka á því þegar upp koma villur en við viljum að kóðinn okkar haldi samt áfram að keyra.

10.3.3 Sjálfgefin viðfögn

Ef við viljum vera viss um að við getum unnið með eitthvað viðfang án þess að neyða notandann til þess að gefa okkur það getum við notað sjálfgefin viðföng (e. named arguments, default arguments), einnig kölluð nefnd viðföng. Þá skilgreinum við fall og búum til skilgreiningu á viðföngunum þar sem við tökum þau fram. Skoðum þetta í kóðabút 10.7 þar sjáum við hvernig megi nota bæði stöðubundin og sjálfgefin saman og hvernig megi kalla í föll sem eru með bæði.

Þegar bæði er notað saman í bland þá þarf að setja stöðubundnu viðfögnin fremst og svo á eftir þeim koma sjálfgefnu viðföngin.

Athugum að bæði er í lagi að nota sjálfgefin viðföng sem stöðubundin, en þá verðum við líka að setja þau inn í réttri röð, og hins vegar að skrifa inn nafnið á viðfanginu og skilgreina það sem eitthvað (hvort sem það séu einhver gögn sem við setjum beint inn eða notum breytu), einnig er í lagi að sleppa því að taka þau fram.

Kóðabútur 10.7: Sjálfgefin viðföng kynnt

```
def hefja_i_veldi_og_deila(grunntala = 1,veldisvisir = 1,deiling = 1):
   print(grunntala**veldisvisir/deiling)

hefja_i_veldi_og_deila()
hefja_i_veldi_og_deila(2, 2, 2)
hefja_i_veldi_og_deila(deiling = 2)
hefja_i_veldi_og_deila(deiling = 2, veldisvisir = 1, grunntala = 4)

def hefja_i_veldi_og_deila_2(grunntala,veldisvisir,deiling = 1):
   print(grunntala**veldisvisir/deiling)

print()
hefja_i_veldi_og_deila_2(1,2)
hefja_i_veldi_og_deila_2(1, 2, 2)
hefja_i_veldi_og_deila_2(1, 2, deiling = 4)

print()
tala = 2
veldi = 2
deila = 2
hefja_i_veldi_og_deila_2(tala, veldi, deiling = deila)
```

```
1.0
2.0
0.5
2.0

1.0
0.5
0.25

2.0
```

Skoðið hér hvernig er kallað í fallið `hefja_i_veldi_og_deila` í línum 4-7 í kóðabút 10.7 og berið það saman við hvernig var kallað í föllin í kóðabútum 10.4 og 10.6. Þar þurfti alltaf að setja inn öll viðföng annars fékkst villa, en vegna þess að í skilgreiningunni kom fram að sjálfgefin gildi eru ákveðnar tölur sem hægt er að nota í útreikningum þá lendir vélin ekki í vandræðum þegar einhver viðföng vantar.

Að sjálfsögðu megum við ekki setja inn of mörg viðföng, þá lendum við í vandræðum, prófið ykkur áfram með það.

Takið líka eftir í línu 7 er kallað í viðföngin í „rangri" röð með því að nota nöfnin á þeim. Prófið að gera það með línu 21, að setja breyturnar inn fyrir viðföngin í annarri röð. Takið eftir því að stöðubundnu viðföngin verða að koma á undan.

Í línu 5 og 14 er kallað eins í föllin, þá er komið fram við sjálfgefnu viðföngin eins og stöðubundin. Þannig að sett er inn fyrir viðföngin í þeirri röð sem þau eru skilgreind í línum 1 og 9.

Eins og hefur komið fram er nauðsynlegt að gera tilraunir og prófanir til að ná árangri og skilningi á efninu.

10.4 Skilagildi

Nú höfum við séð hvernig á að búa til föll, við höfum séð hvernig á að láta föllin vinna með viðföng, það næsta sem við viljum skoða er hvernig á að láta föllin okkar skila útkomu sem megi nota áfram. Við höfum séð hvernig aðferðir á strengi skila oft til okkar öðrum streng sem byggir á strengnum sem við notuðum aðferðina á (sjá kóðabút **??**), til þess að geta notað úttkomuna þá getum við búið til breytu sem grípur það sem aðferðin *skilar*.

Í kóðabútum 10.6 og 10.7 þá er fallið `hefja_i_veldi` skilgreint, en aldrei er hægt að vinna eitthvað með úttkomuna úr útreikningnum,útkoman er bara prentuð. Það dugar okkur ekki, ef við þurfum að nota útreikninginn, að þurfa að horfa á það sem vélin skrifar út og skrifa það handvirkt inn sjálf. Það sem við viljum geta sagt er „hey reiknaðu þetta út og notaðu það svo hér, mér er alveg sama hvað það er því að ég treysti því að þú hafir gert það rétt." Því að við manngerum tölvuna að sjálfsögðu, hvað gæti farið úrskeiðis?

Til þess að geta nýtt þessa virkni þurfum við að læra nýtt lykilorð sem er **return** sem þýðir skila, mjög gagnsætt og gott lykilorð[3]. Það sem lykilorðið gerir er svipað *break* lykilorðinu, þegar vélin kemur að línu þar sem return kemur fram þá gerir fallið ekkert meira en að skila því sem beðið er um og vélin heldur áfram í næstu línu frá því kallað var í fallið. Sjáum í kóðabút 10.8

Kóðabútur 10.8: Hvernig á að láta fall skila gildi með return skipuninni

```
1  def hefja_i_veldi_og_deila(grunntala,veldisvisir,deiling = 1):
2      print('kallað var í fallið')
3      return grunntala**veldisvisir/deiling
4
5  print("í hvaða röð gerast hlutirnir?")
6  utkoma = hefja_i_veldi_og_deila(2, 2, 2)
7  print("útkoman var", utkoma, "og hún margfölduð með tveimur er", utkoma*2)
```

```
1  í hvaða röð gerast hlutirnir?
2  kallað var í fallið
3  útkoman var 2.0 og hún margfölduð með tveimur er 4.0
```

Nú höfum við skilað einu gildi sem er fleytitala og unnið með hana með þeim reiknivirkja sem okkur langaði til að prófa. Hægt er að skila gögnum af hvaða týpu sem er, og jafnvel fleiru en einu gildi í einu, og það af mismunandi týpum.

[3] eins og flest lykilorðin sem við höfum séð hingað til, að mati höfundar

Sjáum í kóðabút 10.9 hvernig hægt er að skila mörgum gildum og setja þau í breytu eða breytur. Það sem fallið tekur við er strengur, tala og listi. Fallið skilar tveimur tölum og listanum aftur óbreyttum Fyrri talan er hversu oft strengurinn í viðfanginu kom fyrir í listanum Seinni talan er hversu margar tölur í listanum eru stærri en talan í viðfanginu

Kóðabútur 10.9: Hvernig á að skila mörgum gildum

```
def skilum_morgum_gildum(strengur, tala, listi):

   skilatala = 0
   strengja_talning = 0
   for stak in listi:
      # x stendur það stak sem verið er skoða úr listanum listi
      if stak == strengur:
         strengja_talning +=1
      if(type(stak) == int or type(stak) == float ):
         if (tala < stak):
            skilatala += 1
   return strengja_talning, skilatala,  listi
```

Nú þegar við erum búin að skilgreina fallið þá getum við notað það og vegna þess að við viljum skoða nokkrar mismunandi leiðir brjótum við það upp í nokkra kóðabúta. Þetta heimfærist ágætlega á Jupyter vinnubækur og við getum litið svo á að hver þessara kóðabúta sé sér sella. Til þess að geta keyrt sellurnar fyrir neðan þarf að vera búið að keyra selluna fyrir ofan.

Kóðabútur 10.10: Fallið úr kóðabút 10.9 notað án þess að útkoman sé geymd

```
skilum_morgum_gildum("halló", 2, ["halló", "bless", 11, 6])
```

```
# ekkert gerist
```

Hér gerðum við ekkert við útkomuna, við kölluðum vissulega í fallið og útreikningarnir voru framkvæmdir en við höfum ekkert í höndunum um það. Reynum að setja útkomuna í breytu eins og við höfum gert ótal sinnum áður.

Kóðabútur 10.11: Fallið úr kóðabút 10.9 notað og útkoman geymd í einni breytu

```
utkoma = skilum_morgum_gildum("halló", 2, ["halló", "bless", 11, 6])
print(utkoma)
print(type(utkoma))
```

```
(1, 2, ['halló', 'bless', 11, 6])
<class 'tuple'>
```

Þannig að þegar mörgum skilagildum er komið fyrir í einni breytu er þeim komið fyrir sem nd. En við munum að það er hægt að úthluta stökum ndar í nokkrar breytur í einu skrefi.

Kóðabútur 10.12: Fallið úr kóðabút 10.9 notað og skilagildunum úthlutað í breytur rétt

```
talning_strengs, staerri_tolur, listinn = skilum_morgum_gildum("halló", 2, ["halló",
    "bless", 11, 6])

print("Hversu oft kom strengurinn fyrir:", talning_strengs)
print("Hversu margar tölur voru stærri en 2:", staerri_tolur)
print("listinn óbreyttur:", listinn)
```

```
Hversu oft kom strengurinn fyrir: 1
Hversu margar tölur voru stærri en 2: 2
listinn óbreyttur: ['halló', 'bless', 11, 6]
```

En við munum einnig að við þurfum að úthluta öllum stökum í eina ákveðna breytu og því virkar eftirfarandi kóðabútur ekki.

Kóðabútur 10.13: Fallið úr kóðabút 10.9 notað og skilagildunum úthlutað í breytur rangt

```
a, b = skilum_morgum_gildum("halló", 2, ["halló", "bless", 11, 6])
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-143-dc266568ff5c> in <module>
----> 1 a, b = skilum_morgum_gildum("halló", 2, ["halló", "bless", 11, 6])

ValueError: too many values to unpack (expected 2)
```

10.5 Innri föll

Föll mega innihalda önnur föll, athugum það sem við fórum yfir í kafla 10.3.1, þessi föll geta verið gagnleg til að útfæra útreikning sem er svo notaður oft innan fallsins. Sjáum dæmi í kóðabút 10.14, athugið sérstaklega gildissviðið því að í innra fallinu er vísað í viðfang sem heitir strengur og það er líka vísað í viðfang í ytra fallinu sem heitir strengur, en vegna þess að gildissviðið krefst þess að fyrst er athugað staðvært hvernig breytan er skilgreind þá skiptir ekki máli að breyturnar heita það sama.

Kóðabútur 10.14: Innri föll kynnt

```
def breyta_strengjum(strengur):
    def fremsti_stafur_er_nuna_aftastur(strengur):
        if(len(strengur) > 2):
            fremst = strengur[0]
            aftast = strengur[-1]
            strengur = aftast + strengur[1:-1] + fremst
            return strengur
        else:
            return strengur

    skilastrengur = fremsti_stafur_er_nuna_aftastur(strengur)
    return skilastrengur
```

Eins og áður er fallið skilgreint fyrst og svo skoðum við frekari notkun í næstu kóðabútum. Athugum sérstaklega að það er bara eitt innra fall þarna og það er ekki ýkja merkilegt, það tekur fremsta og aftasta tákn í streng og víxlar þeim. Við gætum verið með mörg önnur föll og kallað í þau handahófskennt byggt á einhverju eins og hvaða tákn er í þriðja vísi eða álíka. Aðalatriðið hér er að sýna að það er fall sem heitir `fremsti_stafur_er_nuna_aftastur` og það er einungis hægt að kalla í það innan fallsins `breyta_strengjum`. Takið vel eftir inndrætti og því að vissulega er kallað í innra fallið í línu 11.

Köllum nú í fallið með einhverju viðfangi og sjáum hvað gerist.

Kóðabútur 10.15: Innri föll kynnt

```
strengur = breyta_strengjum("halló")
print(strengur)
```

```
óallh
```

Og nú sjáum við að við getum ekki kallað í innra fallið því að við höfum ekki aðgang að því.

Kóðabútur 10.16: Innri föll kynnt

```
strengur =  fremsti_stafur_er_nuna_aftastur("halló")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-147-970f41b86e0c> in <module>
----> 1 strengur =  fremsti_stafur_er_nuna_aftastur("halló")

NameError: name 'fremsti_stafur_er_nuna_aftastur' is not defined
```

Nú höfum við séð hvernig megi skilgreina innri föll og það sem við ætlum að skoða næst er að það má skila föllum. Skipunin return er þá notuð alveg eins og ef við værum að skila einu gildi, eða fleirum. Sjáum í kóðabút 10.17 hvernig við skilum falli og hvernig á að nota skilagildið sem inniheldur fallið. Við sjáum í seinni hluta þessarar bókar, í umfjöllun um klasa, hvernig megi framkvæma sömu virkni en með því að sleppa klösum þá er vinnslutíminn umtalsvert minni. Svo ef það skiptir máli að gera eitthvað hratt sem má leysa með *lokun* (e. closure) þá ætti frekar að beita henni heldur en klösum.

Kóðabútur 10.17: Lokun kynnt

```
def prentunarfall(strengur):
   def prentum():
      return str(strengur) + " hér er viðbót af akademískri ástæðu"
   return prentum

a = prentunarfall('halló heimur')
print(a)
print(a())

b = prentunarfall('nýr strengur')
print(b())
```

```
<function prentunarfall.<locals>.prentum at 0x7fce9ed740d0>
halló heimur hér er viðbót af akademískri ástæðu
nýr strengur hér er viðbót af akademískri ástæðu
```

Úttakið hér minnir okkur á strengjaaðferðir (3.4) þar sem við þurftum að muna að nota sviga til að kalla í aðferðirnar okkar. Það er vegna þess að `a` er fall, það sem `prentunarfall` skilar er fall sem þarf að kalla í með svigum til að nota.

Það er ekki sérlega merkilegt fall, það hermir virkni sem við þekkjum vel úr `print()` fallinu, nema með þessari strengjaviðbót.

Skoðum nú að lokum fall sem er aðeins merkilegra. Við viljum geta búið til niðurteljara sem telur niður úr einhverri tölu en á einhverjum tilteknum tíma. Við leysum það með því að skila falli sem telur niður úr þeirri tölu sem fallinu var gefið.

Kóðabútur 10.18: Innri föll kynnt

```
def niðurtalning(n):
   def teljari():
      nonlocal n
      while(n>-1):
         print(n)
         n -= 1

   return teljari

teljum_fra_fimm = niðurtalning(5)
teljum_fra_tiu = niðurtalning(10)

```

```
13 teljum_fra_fimm()
```

```
1 5
2 4
3 3
4 2
5 1
6 0
```

Við höfum enn ekki fundið góðan tíma til að telja niður frá tíu svo við geymum þá breytu.

Tökum eftir að í skilgreiningum á a og b í kóðabút 10.17 þá erum við að nota ákveðinn streng sem þau eiga að prenta út. Þetta er á þessu stigi málsins eilítið óhlutstætt og ekki augljóst hvernig það nýtist okkur því dæmið í kóðabútnum er ekki sérlega nothæft fall. Niðurtalningarfallið hinsvegar er að framkvæma einhverja virkni sem við viljum hafa aðgang að þegar okkur hentar.

Athugum þó sérstaklega að gildissviðið sem innrafallið teljari() tilheyrir það hefur ekki aðgang að neinu staðværu n-i svo það skilar villu nema að við segjum því falli sérstaklega að nota ekki staðvært n heldur leita út fyrir gildissviðið með lykilorðinu **nonlocal**. Við munum ekki nota það orð af neinu viti í seinni hluta bókarinnar en það er þess virði að taka það fram að svo stöddu að þetta orð sé til og hvað það gerir.

10.6 Æfingar

Athugið að til þess að átta ykkur á því hvort að föllin ykkar séu rétt gerð þurfið þið að gera ítarlegar prófanir á þeim, með mismunandi inntaki. Kallið í föllin í öllum æfingum.

Æfing 10.1 Búið til fall sem tekur við tveimur stöðubundnum breytum og prentar þær út nema fyrri breytan er prentuð á eftir hinni breytunni. Prófið að kalla í fallið með streng og tölu, með tveimur strengjum, og með tveimur tölum. ■

Æfing 10.2 Búið til fall sem tekur við þremur nefndum viðföngum, sem eiga að vera strengir, ef ekkert er gefið upp ákveðið þið sjálf hvert gildi viðfanganna eigi að vera. Innan fallsins athugið þið svo hver strengjanna er aftastur í stafrófinu og prentið hann út. ■

Æfing 10.3 Búið til fall sem tekur við lista af tölum og skilar næsthæstu tölunni. ■

Æfing 10.4 Búið til fall sem að tekur við gildi og lista sem viðföng og skilar hversu oft gildið kemur fyrir í listanum. ■

Æfing 10.5 Búið til fall sem tekur við tveimur strengjum, ef strengirnir byrja báðir á sérhljóða eða á sama samhljóðanum þá skiliði True gildi annars False. ■

Æfing 10.6 Búið til fall sem tekur við tveimur tölum, ef tölurnar eru báðar sléttar skiliði þeirri tölu sem er lægri. Ef önnur eða hvorug er slétt skilarðu þeirri tölu sem er hærri. Nú er gott að benda á að `min()` skilar minnsta gildi alveg eins og `max()` skilar hæsta gildi. ■

Æfing 10.7 Hér fall sem tekur við lista og athugar hvort að stökin 0, 0, 7 komi fyrir í listanum í þessari röð (önnur stök mega vera á milli). Dæmi Listinn [1,2,3,0,1,20,0,7] myndi skila True en listinn [7,0,0,0,0,0,6] myndi skila False. Athugið eitthvað vantar í fallið og þið þurfið að fylla í eyðurnar. Breytan talning telur hversu mörg tákn við erum komin með af þeim þremur sem við þurfum, við erum fyrst með núll og svo hækkum við talninguna eftir því sem við sjáum táknin í þeirri röð sem við viljum sjá þau.

```
1  def fall6(listi):
2     talning = 0
3     for stak in listi:
4        if stak == 0 ?? talning == ??:
5           talning = 1
6        ?? stak == 0 ?? talning == ??:
7           talning = 2
8        ?? stak == 7 ?? talning == ??:
9           return True
10    ??
11
12 fall6([1,2,3,0,1,20,0,7])
13 fall6([1,7,0,0,0,0,0,0,0,0])
```

■

Æfing 10.8 Búið til fall sem tekur við einu viðfangi sem er tala, það sem fallið gerir er lausnin á

verkefni 7.4 en í stað þess að prenta út ndina á fallið að skila ndinni til baka. Áður en for lykkjan er keyrð skulið þið keyra skilyrðissetningu sem athugar hvort að viðfangið sé af týpunni heiltala (int), ef svo er ekki skulið þið skila tómri nd. Það er hægt að athuga með type() fallinu og int lykilorðinu, með != samanburðinum.

Ef viðfangið er heiltala skal halda áfram og sú tala fer inn í range fallið í for lykkjunni. Þannig erum við komin með fall sem getur búið til ndir sem spyrja x oft hver er uppáhalds litur notandans. ■

Seinni hluti
Hlutbundin forritun

11. Kóðasöfn

Kóðasafn (e. library)[1] er endurnýtanlegur kóði, sem sem útfærir ákveðna virkni og hefur ákveðið samhengi. Tilgangur þeirra er að spara forriturum vinnu við að útfæra ýmsa algenga virkni og reiða sig í staðinn á kóða sem er nú þegar til. Hjálpar okkur við að vera ekki að finna upp hjólið í sífellu. Ómögulegt er að ætla að forrita að einhverju viti án þess að nota kóðasöfn.

Við notum kóðasöfn með **import** skipuninni. Þegar import hefur verið sett inn einhvers staðar í skjal þá er óþarfi að setja það inn aftur, venjan er að öll import eru gerð efst í skjali burt séð frá því hvar í skjalinu þau eru notuð. Það gerir kóðann læsilegri og undirbýr okkur við lestur á kóða hvað er að fara að gerast. Sem dæmi ef efst í skjali stendur að kóðasöfnin math og random séu notuð þá vitum við strax að í þessum kóða sé líklega verið að vinna með einhverja handahófskennd og stærðfræði, en ef efst stæði að kóðasöfnin datetime og time væru notuð þá erum við líklega að skoða kóða sem er að vinna með tíma og dagsetningar, það ætti þá ekki að koma okkur á óvart að sjá dagsetningarvinnslu.

Venjan að setja öll „import" efst er því gagnleg fyrir þær sakir að kóðinn verður læsilegur og auðveldara verður að halda utan um kóðasöfnin sem við erum að nota.

11.1 Notkun kóðasafna

Eins og kom fram í kynningu þá viljum við geta einbeitt okkur að því að leysa okkar vandamál í stað þess að finna upp hjólið og því viljum við kynna okkur þau kóðasöfn sem eru í boði sem útfæra virkni sem við viljum beita.

Tilgangur þeirra er að létta okkur lífið og gera virkni aðgengilega. Í næsta undirkafla verða tekin fyrir nokkur gagnleg kóðasöfn en við getum varla talað um tilgang og gagnsemi kóðasafna án þess að taka eitthvert þeirra fyrir. Í inngangi voru kóðasöfnin time og random nefnd. Skoðum þau aðeins núna, sjá kóðabút **??** þar sem kóðasöfnin time og random eru tekin fyrir. Þau bjóða bæði upp á aragrúa aðferða og eiginda sem er út fyrir efni þessarar bókar að ræða í þaula en þó þess virði að taka fyrir ákveðna virkni sem búist er við að nota í æfingum í lok kaflans.

Athugið hér er t
um æfingar í lok
ans

[1] Hugtökin package og module ná einnig yfir kóðasöfn í Python vegna þess hve lauslega kóðasöfn eru skilgreind

Um kóðasafnið time:

- `time.time()` skilar okkur hversu margar sekúndur eru síðan tímatal í tölvum hófst 1.jan 1970.
- `time.sleep()` tekur við tölu og lætur vélina bíða það margar sekúndur áður en hún framkvæmdir aðgerðina í næstu línu fyrir neðan.
- `time.localtime()` skilar okkur nd sem inniheldur í minnkandi röð hver tíminn er, frá ári niður í sekúndur, ásamt deginum í vikunni og árinu og síðasta er gildi sem tekur mið af `isdaylightsavingstime` (**isdst**).

Um kóðasafnið random:Þetta kóðasafn gerir forriturum auðveldara fyrir með því að gera handahófskennd (e. randomness) aðgengilega, það að geta gert hluti af handahófi er mjög mikilvægt í tölvunarfræði og forritun.

- random.randint() nær í heiltölu á lokuðu bili, þar sem báðir endapunktar eru teknir með.
- random.random() nær í fleytitölu á bilinu 0 - 1
- random.choice() nær í stak af handahófi upp úr ítranlegum hlut
- random.shuffle() stokkar upp í raðanlegum hlut.

Kóðabútur 11.1: Notkun kóðasafna með time

```
import time

sekundur_adan = time.time()
time.sleep(3)
sekundur_3_sek_eftir_adan = time.time()

thrir = sekundur_3_sek_eftir_adan - sekundur_adan
print(thrir)
```

```
3.001107931137085
```

Þið fáið ekki nákvæmlega sömu tölu, og ekki einu sinni ef þið reynið að keyra þetta oft, það er líka ómögulegt að ná keyrslu upp á millisekúndu-nákvæmni með þessu móti.

Skoðum næst handahófskennd.

Kóðabútur 11.2: Notkun kóðasafna með random

```
import random

listi = [1,2,3,4,5,6,7,8,9]
einhver_tala = random.choice(listi)
random.shuffle(listi)
print(listi)
for i in range(einhver_tala):
   if random.randint(min(listi),max(listi)) > i:
      print("Þú vannst 10kr")
   else:
      print("Þú þarft að ydda blýantana þína")

fjoldi_folks = 100
hlutfall_folks_med_raudan_trefil = fjoldi_folks*random.random()
print(hlutfall_folks_med_raudan_trefil)
```

```
[9, 3, 2, 7, 5, 1, 4, 8, 6]
Þú vannst 10kr
Þú vannst 10kr
```

```
Þú vannst 10kr
Þú vannst 10kr
Þú þarft að ydda blýantana þína
Þú þarft að ydda blýantana þína
Þú þarft að ydda blýantana þína
Þú þarft að ydda blýantana þína
98.19660121258345 %
```

Sama upp á teningnum hér, úttakið verður ekki það sama þegar þið keyrið þennan kóða en ekki af sömu ástæðu. Hér er það beinlínis ætlunarverkið að úttakið verði óútreiknanlegt.

Eins og sést í kóðabút 11.1 þarf að nota nafnið á kóðasafninu til að ná í aðferðir og virkni. Sum kóðasöfn heita löngum nöfnum og ef fólk vill stytta nöfnin á þeim þá má nota lykilorðið **as** til þess að varpa nafninu á kóðasafninu í annað breytuheiti, sjá notkun í kóðabút 11.3. Vísunin kemur strax þegar kóðasafnið er flutt inn og þar eftir er kóðasafnið aðgengilegt með þessari vísun. Mikilvægt er þá að hafa í huga að nefna kóðasöfnin eittvað sem verður ekki óvart yfirskrifað í kóðanum, og gæti valdið ruglingi. Eins og að skipta út `import random` fyrir `import random as listinn_minn`. Það væri hrikalegt, illlæsilegt og myndi fyrirsjánlega valda vandamálum. En nú vitum við hvernig á að nota kóðasöfn.

Kóðabútur 11.3: Lykilorði as

```
import random as rnd
import time as t

timi = t.time()
tala = rnd.random()
print(timi * tala % tala)
```

```
0.4772097503672992
```

11.2 Nokkur gagnleg kóðasöfn

Tilgangur þessarar bókar er ekki að tiltaka hvert einasta kóðasafn sem er til, heldur að kynna til sögunnar hvernig notkun þeirra virkar og einhver þau algengustu eða skemmtilegustu kóðasöfn sem eru notuð í dag. Þetta er gert til þess að halda bókinni frá því að verða eins og símaskrá (ef einhver lesandi man eftir að hafa haldið á símaskrá) og passa að hún haldi í við þróun í fræðigreininni.

Að því sögðu er hér stutt kynning á nokkrum vinsælum kóðasöfnum.

- **math**, þetta kóðasafn gerir aðgengilegar allskonar stærðfræðilegar aðgerðir. Eins og hornaföll, logra, veldisföll og tölulega vinnslu. Ásamt því að gefa okkur aðgengi að mikilli nákvæmni á hinum ýmsu rauntölustærðum eins og pí.
- **numpy**, er kóðasafn sem gerir stærðfræðilega vinnslu aðgengilega, t.d. á fylkjum. Numpy er sérhæft fyrir flóknari vinnslu en math.
- **scipy**, er það sem hægt er að nota til þess að vinna með vélanám (e. machine learning). Scipy hefur oft verið kallað óþarflega flókið í notkun, en hugmyndirnar sem þar er verið að vinna með eru í grunninn mun flóknari en í numpy og math pökkunum svo það ætti ekki að koma á óvart.
- **pygame**, er kóðasafn sem vinnur með grafískt viðmót og inntak frá notanda oft með mjög skapandi útkomu. Margir litlir leikir hafa orðið til með þessu kóðasafni og er til mýgrútur af dæmum og leikjum á netinu til að vinna út frá.
- **datetime**, sem vinnur með dagsetingar því ef við ætlum einhvern tímann að skrifa hugbúnað sem vinnur með dagsetningar þá viljum við alls ekki finna upp hjólið. Þar er tekið á daylightsavings, hlaupárum og öðru slíku sem við viljum ekki þurfa að hafa áhyggjur af

- **matplotlib**, er safn sem var búið til í kringum tvívíða sýn á gögn. Til að sýna gröf og á fjölbreytilegan máta gera gögn sýnileg. Þetta er annað safn sem hefur slæmt orð á sér fyrir að vera óþarflega óaðgengilegt.

Að búa til kóðasöfn er út fyrir efni bókarinnar en það er lítið mál að kynna sér það á vefnum.

11.3 Æfingar

Æfing 11.1 Notið input skipun til þess að komast að því hvort að notandinn viti hversu lengi 10 sekúndur eru að líða. ■

Æfing 11.2 Gefinn er eftirfarandi listi: [’Lorem’,’ipsum’,’dolor’,’sit’,’amet,’,’consectetur’, ’adipiscing’,’elit,’,’sed’,’do’,’eiusmod’,’tempor’, ’incididunt’,’ut’,’labore’,’et’,’dolore’,’magna’,’aliqua.’, ’Ut’,’enim’,’ad’,’mini’,’veniam,’,’quis’,’nostrud’, ’exercitation’,’ullamco’,’laboris’,’nisi’,’ut’,’aliquip’, ’ex’,’e’,’commodo’,’consequat.’,’Duis’,’aute’,’irure’, ’dolor’,’in’,’reprehenderit’,’in’,’voluptate’,’velit’, ’esse’,’cillum’,’dolore’,’eu’,’fugiat’,’nulla’,’pariatur.’, ’Excepteur’,’sint’,’occaect’,’cupidatat’,’non’,’proident,’, ’sunt’,’in’,’culpa’,’qui’,’officia’,’deserunt’,’mollit’, ’ani’,’id’,’est’,’laborum.’]

Náið í orð af handahófi úr listanum og náið svo í tákn af handahófi úr því orði. ■

12. Skjalavinnsla

Að vinna með skjöl án þess að hafa þau opin í ritli, eins og MS Word, LibreOffice Write, Notepad eða álíka, er mjög ákjósanlegt ef t.d. þarf að gera litla breytingu í stóru skjali eða einhverja breytingu í mjög mörgum skjölum (skilgreiningin á mjög mörgum er sveiginaleg, sumum finnst það vera að gera eitthvað oftar en þrisvar). Segjum sem svo að við hefðum skrifað ritgerð og við yfirlestur á henni tækjum við eftir að við gerðum eina ákveðna villu alltaf. Villan væri að við hefðum gleymt að setja stóran staf í upphafi allra setinganna okkar! Ó nei, hvernig tókst okkur að gleyma þessu? Það á eftir að taka óratíma að lesa yfir og laga hvern einasta staf því ritgerðin er 20 blaðsíður. En vegna þess að við erum snjöll og kunnum að vinna með strengi þá getum við gert þetta með hjálp Python.

Hægt er að búa til skjöl, opna þau, lesa þau, skrifa inn í þau, yfirskrifa þau, loka þeim og henda þeim. Þetta er mikilvægt vegna þess að við viljum geta sagt tölvunni að nálgast skjöl og gera eitthvað við þau, við viljum ekki þurfa að handstýra tölvunni að óþörfu, til dæmis með því að lesa sjálf yfir 20 blaðsíður í leit að litlum staf þar sem á að vera stór. Til þess er tölvan.

Kóðabútur 12.1: Hér sjáum við hvernig má búa til skjöl

```
1 skjal = open('skjal1.txt', mode = 'w+')
2
3 skjal.write('hér kemur eitthvað sem við viljum setja inn í skjalið okkar')
4
5 skjal.close()
```

Nú getum við séð, á sama stað og þessi kóði er keyrður, í skrársafninu (e. filesystem) okkar skjal sem heitir `skjal1.txt` vegna þess að við völdum það nafn í línu 1 innan svigans, ástæðan fyrir því að þarna stendur *.txt* er sú að sú skráarending er fyrir einföld textaskjöl[1]. Þar kemur einnig fram `mode` sem er valið sem `w+`, sem leyfir okkur að lesa og skrifa eða yfirskrifa. Við veljum máta sem hentar okkur hverju sinni, þeir sem eru í boði eru:

- **r** : leyfir okkur aðeins að lesa (read).
- **w** : leyfir okkur aðeins að skrifa (write).

[1] svipað eins og .doc sem við könnumst flest við

- **a** : leyfir okkur aðeins að bæta við aftast (append).
- **r+** : leyfir okkur að lesa og skrifa (read +).
- **w+** : leyfir okkur að lesa og skrifa og yfirskrifar (write +).

Við tökum einnig eftir í kóðabút 12.1 að við notum þrjú föll, fallið `open()` er það sem býr til skjalið með w+ og nafninu sem við gefum því, þá er skjalið opið og aðgengilegt. Þá köllum við í `write` sem skrifar inn í skjalið, sú skipun er í boði vegna þess að við notuðum w+ (er ekki möguleg fyrir r t.d.). Við erum vissulega bara að gera einfalda hluti en þetta er til að sýna virknina í grunninnn ekki til að finna upp hjólið. Svo að lokum sjáum við eitthvað áhugavert, það er `close()`. Til hvers að gera það? Hafið þið lent í því að við það að reyna að henda skjali í ruslið af tölvunni ykkar fáið þið villu um að það sé ekki hægt því að skjalið er opið einhversstaðar? Það er það sem við erum að fyrirbyggja hérna með því að loka skjalinu þegar við erum búin að vinna með það. Við erum í rauninni bara að ganga frá eftir okkur svo að það sé ekki eitthvað opið sem er fyrir.

12.1 Unnið með skjöl

Til þess að geta unnið með skjöl þurfum við að geta vísað í þau, til þess notum við breytur. Eins og breytan `skjal` í kóðabúr 12.1. Breytan okkar stendur þá ekki fyrir einhverja grunntýpu í Python heldur er það vísun í heilt skjal á skráarsafninu okkar.

Í kóðabút 12.2 koma fram nokkrar aðferðir sem Python býður upp á fyrir skjöl sem búið er að opna. Til þess að sækja upplýsingar úr skrá þarf tölvan að lesa hana. Við tökum eftir því að þar er eitthvað til sem heitir *seek*, ástæðan fyrir því að við þurfum það er að tölvan les frá vinstri til hægri eins og henni er sagt en ef hún á að lesa eitthvað aftur frá byrjun eða öðrum stað þá þarf að segja henni að færa leshausinn sinn þangað. Nú er hætta á því að enginn lesandi hafi nokkurn tímann séð segulband en hugmyndin þar er sú sama, tölvan sem les segulbandið getur bara lesið bandið sem er undir leshausnum og sér ekkert annað. Ef tölvan á að lesa einhvern annan hluta af segulbandinu þarf að spóla fram eða til baka. Sama er upp á teningnum hérna, við þurfum að stilla leshausinn fyrir framan það gildi sem við viljum lesa hverju sinni. Ef vélin er búin að lesa skjalið er leshausinn kominn út í enda og við getum ekki lesið meira nema færa hann.

Þetta er svipað því eins og ef við settum puttann niður í bók og mættum bara lesa orðin fyrir ofan puttann og svo bara til hægri, þá til að lesa eitthvað aftur eða fara lengra inn í bókina þyrftum við að taka upp puttann og færa hann þangað.

Byrjum á að gera textaskjalið okkar aðeins bitastæðara með því að keyra eftirfarandi kóða í sér sellu í vinnubók[2]:

```
%%writefile skjal1.txt
hér kemur eitthvað sem við viljum setja inn í skjalið okkar
hér er næsta lína í skjalinu
úps engin lína endar á punkti og engin lína hefst á stórum staf
En nú lagast það.
æ, það gleymdist stór stafur.
Og nú gleymdist punktur
```

Skoðum svo hvað hægt er að gera við þennan texta[3].

Kóðabútur 12.2: Hér sjáum við einfalda skjalavinnslu

[2] virkar aðeins í Jupyter Notebooks og er ekki sér Python fyrirbæri heldur sérstætt fyrir þessar vinnubækur, þetta leyfir okkur að sleppa við \n eða newline character og æfingar tengdum því

[3] Þarna kemur fram annað viðfang sem heitir encoding, þarna er það valið sem utf-8 sem er það táknasafn sem nær yfir alla sériślenka stafi. Ef þið lendið í vandræðum við að íslenskir stafir eru í ruglinu er gott að vita til þess að stafakóðunin er þá mögulega önnur en utf-8. Þetta á alls ekki bara við um Python heldur er þetta alþjóðlegur staðall sem nýtist hvarvetna.

```
vinnuskjal = open('skjal1.txt', encoding = 'utf-8', mode = 'r')

linurnar = vinnuskjal.readlines()
vinnuskjal.seek(150)
print(vinnuskjal.read())

for lina in linur:
   if lina[0].islower() and lina[-1] != ".":
      print("línan byrjar á litlum staf og endar ekki á punkti")
   elif lina[0].islower() or lina[-1] != ".":
      print("línan endar ekki á punkti eða byrjar á litlum staf")
   else:
      print(lina)

vinnuskjal.close()
```

```
t á stórum staf
En nú lagast það.
æ, það gleymdist stór stafur.
Og nú gleymdist punktur

línan byrjar á litlum staf og endar ekki á punkti
línan byrjar á litlum staf og endar ekki á punkti
línan byrjar á litlum staf og endar ekki á punkti
línan endar ekki á punkti eða byrjar á litlum staf
línan byrjar á litlum staf og endar ekki á punkti
línan endar ekki á punkti eða byrjar á litlum staf
```

Í kóðabút 12.2 eru teknar fyrir þær helstu aðferðir sem eru í boði fyrir lestur á skjali `read()` og `readlines()`, annað gefur okkur streng en hitt gefur okkur lista af línum sem við getum ítrað í gegnum. Takið eftir því að leshausinn er settur á stað 150 og svo er útkoman prentuð, en það hefur ekki áhrif á `linur` því að sú breyta var skilgreind þegar leshausinn var á 0 og færðist út í enda við það að nota `readlines()`. Prófið ykkur áfram með röðunina á kóðalínunum.

Það gæti verið vesen að þurfa að muna eftir því að loka skjalinu og því viljum við skoða annan möguleika með nýju lykilorði **with**, við höfum séð **as** sem býr til *alias* eða annað heiti. Sjáum kóðabút 12.3, þar sem öll vinnslan í skjalinu tilheyrir inndrætti undir `with` og `as`. Þetta er eins og með föll, þegar við skrifum eitthvað í sama inndrætti og í línu 1 þá erum við komin út fyrir skjalavinnsluna okkar og skjalið er ekki lengur aðgengilegt því þá er verið að reyna að vinna með skrá sem er lokuð.

Kóðabútur 12.3: Hér sjáum við nýja leið til að opna skjal og loka því sjálfkrafa

```
with open('skjal1.txt', encoding='utf-8') as test:
   efni = test.read()
   test.seek(0)
   efni_i_listum = test.readlines()

with open('skjal1.txt', encoding='utf-8', mode= 'w+') as blergh:
   efni = blergh.write('Ég yfirskrifaði allt og á nú skjal sem heitir það sama en
    inniheldur bara þetta')
   blergh.seek(0)
   efni_i_listum2 = blergh.readlines()

print(efni_i_listum)
print(efni_i_listum2)
```

```
['hér kemur eitthvað sem við viljum setja inn í skjalið okkar\n', 'hér er næsta lína í
    skjalinu\n', 'úps engin lína endar á punkti og engin lína hefst á stórum staf\n', 'En
    nú lagast það.\n', 'æ, það gleymdist stór stafur.\n', 'Og nú gleymdist punktur\n']
```

```
['Ég yfirskrifaði allt og á nú skjal sem heitir það sama en inniheldur bara þetta']
```

Tökum eftir hér að við fáum villu við að reyna að vísa í breyturnar `test` og `blergh` en hinar breyturnar sem við búum til á meðan vinnslunni stendur eru enn aðgengilegar eins og sést í línum 12 og 13.

En þá skulum við skoða hvernig við eigum að fara að því að laga textann án þess að opna skjalið handvirkt og breyta táknunum sjálf. Það sem við vitum er að fremsti stafur á alltaf að vera stór og aftasta táknið á alltaf að vera punktur. Við getum svo breytt handvirkt þeim fáeinu setningum sem við viljum að endi á spurningarmerki eða upphrópunarmerki.

Kóðabútur 12.4: Leysum punkta og hástafa vandann okkar

```
with open('skjal1.txt', encoding='utf-8', mode= 'r') as lausn:
   lausn.seek(0)
   linur = lausn.readlines()
   for lina in linur:
      i = linur.index(lina)
      lina = lina[0].upper() + lina[1:]
      if lina[-2] != ".":
         lina = lina[:-1] + "." + lina[-1]
      linur[i] = lina

with open('skjal1.txt', encoding = 'utf-8', mode = 'w+') as laga:
   laga.writelines(linur)
   laga.seek(0)
   print(laga.read())
```

```
Hér kemur eitthvað sem við viljum setja inn í skjalið okkar.
Hér er næsta lína í skjalinu.
Úps engin lína endar á punkti og engin lína hefst á stórum staf.
En nú lagast það.
Æ, það gleymdist stór stafur.
Og nú gleymdist punktur.
```

Athugið sérstaklega notkun á -1 og -2, athugið hvað gerist ef þið breytið því, þetta er útaf því að síðasta táknið er „new line character" eða \n (þó þetta eru í raun tvö tákn þá er þetta saman eitt tákn). Þó það hefði mögulega verið minni hausverkur að laga þessar fáeinu setningar og að læra rétta stafsetningu þá er pælingin hérna var að við vorum búin að gera þetta síðustu tuttugu blaðsíðurnar og við vildum alls ekki gera einhverja villu í yfirferðinni með því að gleyma punkti einu sinni því að það fór framhjá okkur. Við ættum að temja okkur að láta tölvuna sjá um allt það sem við erum fær um að útskýra fyrir henni hvernig eigi að gera.

12.2 Æfingar

Æfing 12.1 Búiið til skjal með .txt endingu, setjið inn í það nokkrar mismunandi línur (annað hvort með %%writefile eða annarri leið og munið þá eftir \n til þess að fá mismunandi línur). Lykkjið svo í gegnum línurnar og prentið út þær sem byrja á sérhljóða. ■

Æfing 12.2 Búið til textaskjal sem inniheldur söngtexta úr einhverju lagi (þá er gott að nota %%writefile), lesið svo skjalið og geymið línurnar í lista. Prentið svo eina línu af handahófi úr listanum. ■

13. Klasar og hlutir

Forritun snýst um að meðhöndla gögn, hingað til höfum við kynnst nokkrum gagnatögum (t.d. strengir og listar), þær gegna mismunandi hlutverkum og bjóða upp á mismunandi aðgerðir til að vinna með gögnin. Þessar innbyggðu týpur duga þó ekki alltaf og því er mikilvægt að vita að þegar við forritum getum við smíðað okkar eigin. Þannig getum við aðlagað týpurnar okkar að þeim gögnum sem forritið okkar meðhöndlar og útfært okkar eigin aðferðir á þær. Klasar gera forriturum kleift að skilgreina sína eigin hluti í flestum hlutbundnum málum, Python er hlutbundið forritunarmál. Til þess að læra á hvernig eigi að búa til klasa þarf að átta sig á til hvers þeir eru nytsamlegir.

Gagnlegt er að hugsa sér klasa sem skilgreiningu eða uppskrift alveg eins og föll. Skilgreiningin ein og sér gerir ekki neitt, það er ekki fyrr en við búum okkur til ákveðna útgáfu sem við getum farið að vinna með hana. Gott dæmi um það er skilgreiningin á rétti á matseðli á veitingastað, textinn á matseðlinum er eingöngu hvað er í boði en er ekki útgáfa af matnum sjálfum.

Klasar eru hlutir sem hugsaðir eru til þess að búa til eintök af og geyma þannig eitthvert ástand og mögulega hafa áhrif á það. Hugmyndin er að eiga hlut eða *tilvik*, eina tiltekna útgáfu, sem má framkvæma aðgerðir á og eitthvað ástand hlutarins breytist eftir því hvað var gert, þannig er hægt að búa til mörg eintök af sama klasanum og láta hvert tilvik verða fyrir mismunandi áhrifum[1].

13.1 Klasar skilgreindir

Klasar nota lykilorðið **class** og eru skilgreindir með því orði, allt sem tilheyrir klasanum er inndregið undir honum. Nöfn klasa eru með stórum staf í Python og flestum hlutbundnum forritunarmálum.

Kóðabútur 13.1: Klasinn Bíll skilgreindur

```
1 class Bill:
2     tegund = "Citroen"
3
```

[1] Athuga þarf sérstaklega gildissvið þegar klasar eru annarsvegar, gildissvið í Python geta verið ögn ruglingsleg en við munum ekki beita klösum á það sérhæfðan máta að við lendum í miklum vandræðum. Hér er tilvalið að skoða „meta programming”

```
4 fyrsti_billinn = Bíll()
5 print(fyrsti_billinn.tegund)
```

```
1 Citroen
```

Hugsum okkur að við búum til skilgreiningu á bíl, hann þarf að vera af einhverri tegund, skoðum línur 1-2 í kóðabút 13.1. Svo viljum við fá tilvik af skilgreiningunni í hendurnar (lína 4), þá búum við til breytu sem fær gildi eins og við höfum gert hundrað sinnum áður, nema núna er gildið sem breytan fær nafnið á klasanum okkar ásamt svigum eins og við séum að kalla í hann. Prófið núna að búa til annað tilvik af klasanum `Bíll` án þess að nota svigana og prófið þá að prenta út það sem `type` skilar fyrir breyturnar tvær.

Kóðabútur 13.2: Klasinn Bíll skilgreindur og tvö tilvik búin til

```
1 class Bíll:
2    tegund = "Citroen"
3
4 fyrsti_billinn = Bíll()
5 print(fyrsti_billinn.tegund)
6 annar_bill = Bíll()
7 annar_bill.tegund = "Subaru"
8 print(annar_bill.tegund)
```

```
1 Citroen
2 Subaru
```

Breytan `fyrsti_bilinn` kemur ekki í veg fyrir það að við getum átt fleiri bíla, en hún heldur utan um ástandið á nákvæmlega þessum bíl okkar. Segjum að við fáum okkur svo annan bíl, þá getum við búið til aðra breytu (lína 6) í kóðabút 13.2 fyrir annað tilvik af klasanum. Bílarnir eru, fyrir okkur, óaðgreinanlegir í línu 6[2] en það breytist svo snarlega þegar við endurskilgreinum *klasabreytuna*[3] í línu 7, `tegund`.

Prófið nú að skipta um gildi á klasabreytunni `tegund` fyrir ykkar eigið tilvik af `Bíll`.

Þá skulum við skoða dálítið sérkennilegt fyrirbæri í Python, það er að við getum endurskilgreint klasabreyturnar okkar, sem hefur áhrif á öll tilvikin okkar. Til að skoða það skulum við nota aftur kóðann úr kóðabút 13.1.

Kóðabútur 13.3: Endurskilgreining á því sem klasinn býður upp á

```
1 class Bíll:
2    tegund = "Citroen"
3
4 fyrsti_billinn = Bíll()
5 print(fyrsti_billinn.tegund)
6 Bíll.tegund = "Volvo"
7 print(fyrsti_billinn.tegund)
```

```
1 Citroen
2 Volvo
```

Hér sjáum við hvernig tilvikið okkar, `fyrsti_billinn`, af bílaklasanum breytist. Í línu 5 er `tegund` "Citroenen í línu 7 er það orðið að "Volvo". Þetta gerist því að tilvikið okkar er af þessum klasa og klasinn breyttist í línu 6. Við endurskilgreindum klasann og því breytast öll tilvik af honum í samræmi.

[2] Þar sem ekki hefur verið útfærð __eq__ aðferðin þá er notast við id() fallið úr type klasanum sem klasinn okkar erfir frá bakvið tjöldin. Við skoðum erfðir betur seinna í kaflanum.

[3] Í öðrum hlutbundnum málum er venjulega talað um klasafasta en í Python er auðvelt að breyta þeim svo við hæfi að nota annað orð en klasa**fasti**

Nú eigum við tvær breytur sem við getum unnið með, kannski setja bensín á bílinn eða fylla á rúðuvökva og þá gerum við það við þá tilteknu breytu sem við ætlum að framkvæma þá aðgerð á. En þessi skilgreining innihélt engar aðferðir, við sjáum það í hluta 13.3.

13.2 Tilviksbreytur

Nú höfum við séð hvernig hægt er að búa til tilvik af klasa, en klasinn úr kóðabút **??** er sérstaklega ber og gagnlítill. En hvers eru klasar megnugir?

Athugum eftirfarandi samlíkingu áður en lengra er haldið. Þegar við förum á veitingastað þá er okkur boðinn ákveðinn matseðill, við fáum að vita að það séu þrír réttir á matseðlinum (þrír klasar) og í þeim réttum eru ákveðin hráefni (tilviksbreytur) og þegar við pöntum okkur mat fáum við í hendurnar eitt tiltekið tilvik af skilgreiningunni á matseðlinum (tilvik af klasa). Nú eru hráefnin kannski ekki okkur að skapi og við viljum fá að hafa áhrif á hvaða hráefni fara í réttinn okkar (okkar tiltekna tilvik) svo við gefum upp hvað við viljum fá (inntak) sem skilar sér í okkar tiltekna rétti (úttak).

Í þessari samlíkingu er matreiðslufólkið smiðurinn á bakvið klasann, í kóðabút 13.4 er aðferðin `__init__` sá smiður. Aðferðin smíðar fyrir okkur tilvik af klasanum með því inntaki sem hún fær.

Kóðabútur 13.4: Klasar skilgreindir með töfraaðferðinni __init__

```
1  class Samloka():
2     def __init__(self, sosa, alegg):
3        self.sosa = sosa
4        self.alegg = alegg
5
6  samlokan_min = Samloka('bbq', ['skinka', 'ostur', 'paprika'])
7
8  class Samloka_med_skinku():
9     def __init__(self, sosa = "", alegg = ['skinka']):
10       self.sosa = sosa
11       self.alegg = alegg
12
13 skinku_samloka = Samloka_med_skinku('bbq')
```

Í fyrri klasanum, `Samloka`, verðum við að gefa upp inntak fyrir `sosa` og `alegg` þegar við búum okkur til hlut því annars fáum við villu en í seinni klasanum, `Samloka_med_skinku` þá eru tilviksbreyturnar með sjálfgefin gildi. Þetta rímar ágætlega við raunheiminn, þar sem við verðum að tilgreina hvað við meinum með „samloka" en „skinkusamloka" er mun afmarkaðra.

Samlíkingin okkar með samlokur á veitingastað er ágæt en nú skulum við skoða hvað er eiginlega í gangi í kóðabút 13.4. Fyrir það fyrsta er klasinn núna skilgreindur sem `Samloka()` með svigum, það var ekki þannig í kóðabútum 13.1 - 13.3.

Ítarefni 13.1 Svigar eða ekki svigar?
Ástæðan fyrir því að svigar eru valkvæmir er svipuð og í kafla 5 þar sem mátti sleppa svigum utan um segðir fyrir skilyrðissetningar nema það væri þörf á þeim til útreiknings. Klasar eiga möguleika á að **erfa** (e. inherit) frá öðrum klösum, við munum tala um það í undirkafla 13.5, og þeir erfa í grunninn allir frá klasanum *Object*. Það sem tómur svigi þýðir (eða að sleppa sviganum alfarið) er að klasi erfi ekki frá öðrum klasa. Því er það upp á einstaklinginn komið að venja sig á að gera alltaf annað hvort, höfundur hefur vanið sig á tóma sviga en er það enginn heilagur sannleikur.

Næsta sem við þurfum að athuga er töfraaðferðin[4] `__init__` og orðið *self*. Orðið self eitt og sér er ekki lykilorð, það má skipta því út fyrir eitthvað annnað, hins vegar hefur komist ákveðin venja á að nota það orð og gerir það kóða læsilegri að halda sig við það. En hvað gerir orðið self? Þetta orð er breyta sem inniheldur tilviki af klasanum sjálfum, í okkar tilfelli er það `samlokan_min` í línu 6, þetta skýrist kannski þegar við skoðum endurkvæmni í kafla 15.

Í klasanum `Samloka` eru viðföngin `sosa` (sem við búumst við að sé strengur án þess að athuga það neitt sérstaklega, sjá kafla 14 um hvernig megi taka á því) og `alegg` (sem við búumst við að sé listi af strengjum). Ef notandinn gefur okkur ekkert inntak við gerð samlokunnar þá er ekki hægt að búa til tilvik af samlokunni, því `__init__` aðferðin, *smiðurinn*, býst við tveimur stöðubundnum viðföngum og getur ekkert gert án þeirra nema skila villu að svo stöddu. Þegar við skilgreindum `samlokan_min` þá sögðum við við smiðinn að við ætluðum að eiga eitt stykki samloku með bbq sósu, skinku, osti og papriku. Þannig að `sosa` inniheldur núna strenginn `bbq` fyrir þetta tiltekna tilvik af klasanum og `alegg` þennan tiltekna lista af áleggstegundum.

Það þriðja og kannski það erfiðasta að skilja er að `init` aðferðin í klasanum Samloka_med_skinku tekur við nefndum viðföngum, eins og við sáum í kafla 10.3.3, sem hafa einhver tiltekin gildi nú þegar skilgreind. Sem þýðir að við getum búið til einhverja óbreytta, staðlaða, sjálfgefna skinku samloku. Við þurfum ekki að gefa neitt upp til þess að fá tilvikið í hendurnar, hins vegar ef okkur langar til þess að fá samloku með einhverri sósu og einhverju öðru áleggi þurfum við að gefa það upp og við getum gert það alveg eins og þegar við notum föll með sjálfgefnum/nefndum viðföngum.

13.3 Aðferðir

Við þekkjum aðferðir, við höfum séð þær notaðar á týpurnar sem við þekkjum, eins og .capitalize() á strengi, .sort() á lista og .get("x", "y") á orðabækur. Aðferðir eru í raun föll sem eru skilgreind inni í klösum og verka á hlutinn sem klasinn skilgreinir (til upprifjunar sjá kafla 3.4).

Nú ætlum við að skilgreina okkar eigin aðferðir á hlutina okkar. Við ætlum að skoða aðferðir með tilliti til rafbíla. Það sem við viljum geta gert þegar við búum til tilvik af rafbíl er að segja hvaða tegund hann hefur, hvaða árgerð hann er af, hversu mikla drægni hann hefur á 100km, hversu margar kílówatt stundir rafhlaðan er og hversu marga kílómetra er búið að aka bílnum.

Kóðabútur 13.5: Klasa aðferðir á rafbílaklasa

```
1  class Rafbill():
2     def __init__(self, tegund, model, draegni = 16.7, kws = 40, akstur = 0):
3        self.tegund = tegund
4        self.argerd = argerd
5        self.eydsla = draegni/100     # hversu mörgum kw stundum bíllinn eyðir á 1 km
6        self.kws = kws                # hversu mikil hleðsla kemst fyrir
7        self.akstur = akstur          # km sem hafa verið eknir
8
9     def keyra_km(self, km):
10       self.akstur += km
11       self.kws -= self.eydsla * km
12
13    def hlada_bilinn(self, kw):
14       self.kws += kw
```

Við viljum að það að aka bílnum ákveðna kílómetra hafi áhrif á stöðu rafhlöðunnar. Við viljum líka geta hlaðið bílinn. En eins og sést í kóðabút 13.5 þá er hægt að hlaða bílinn endalaust og það er hægt að keyra hann endalaust líka. Við settum engin takmörk á það hvað má keyra marga kílómetra, við höldum bara áfram að lækka hleðsluna og við leyfðum okkur svo að hlaða bílinn langt umfram

[4] töfraaðferðir (e. magic methods, double underscore methods, dunder methods (þarna er orðunum double og under skeytt saman í dunder)) er hópur aðferða sem Python býður upp sem staðlað viðmót sem gerir forriturum kleift að nýta grunnvirkni eins og samanburður með samanburðarvirkja.

það hversu margar kílówattstundir komast fyrir í rafhlöðunni. Einnig er galli á þessum klasa að engin leið er til að halda utan um hvert er hámark hleðslu rafhlöðunnar.

En þetta dugar til að sýna fram á hvernig aðferðir eru skilgreindar, hvernig á að kalla í þær, hvernig þær hafa áhrif á tilveiksbreyturnar okkar og svo hvernig má kalla í tilviksbreyturnar til að sjá áhrifin. Takið sérstaklega eftir því í línu 14 að `kw` og `self.kw` er ekki það sama, eins og kom fram áður þá er self að vísa í tilvik af klasanum og klasi af þessu tagi hefur `kws` sem tilviksbreytu. Hér er þá verið að vísa í þær kw-stundir sem tilvikið bjó yfir þegar kallað var í aðferðina en seinna viðfangið, staka `kws` er fengið frá notandanum sem kallaði í aðferðina og í núna er það bara hugmynd að einhver muni á endanum gera það, svo sjáum við í kóðabút 13.6 hvernig það er gert.

Aðferðir þurfa þó ekki endilega að hafa áhrif á tilvikið okkar heldur geta skilað okkur til baka einhverri niðurstöðu, eins og flestar aðferðir á strengi (því við munum að strengir eru óbreytanlegir).

Engin aðferðanna í þessum klasa skilaði nokkurri niðurstöðu. En skoðum þó hvernig megi nota aðferðirnar á eitthvað tiltekið tilvik.

Kóðabútur 13.6: Tilvik af rafbílaklasanum búið til og notað

```
1 rafbill = Rafbill('Rafio', 2021)
2 rafbill.keyra_km(500)
3 print(rafbill.akstur)
4 rafbill.hlada_bilinn(900)
5 print(rafbill.kws)
6 rafbill.tegund = "Oiarf"
7 print('nú er bíllinn af tegundinni', rafbill.tegund)
8 rafbill.keyra_km(500)
9 print('nú er bíllinn búinn að keyra', rafbill.akstur)
```

```
1 500
2 856.5
3 nú er bíllinn af tegundinni Oiarf
4 nú er bíllinn búinn að keyra 1000
```

Hérna er kallað í báðar aðferðirnar sem við skilgreindum í kóðabút 13.5 með ákveðnu inntaki. Takið einnig eftir að `rafbill` og `Rafbill` er ekki það sama, í Python skiptir máli hvort eru notaðir hástafir eða lágstafir.

Tökum nú nýtt dæmi þar sem við skoðum ímyndað lestarkerfi á Íslandi. Í þessu dæmi höldum við utan um tvennt með klösum, annars vegar lestarstöðvar sem hafa nöfn og eru í ákveðinni fjarlægð frá upphafsstöðinni á leiðinni sinni og hins vegar lestar sem eru á ákveðinni leið og eru staddar á ákveðinni stöð. Í kóðabút 13.7 sjáum við hvernig aðferðir geta skilað einhverju án þess að hafa áhrif á tilviksbreytur og við sjáum einnig að smiðurinn init tekur bara við tveimur breytum frá notanda en skilgreinir þrjár tilviksbreytur, þetta er vegna þess að klasinn býður notandanum ekki að hafa áhrif á þessa breytu við smíð klasans. Notandinn verður því að fá í hendurnar við grunnstillingu lest sem hefur ekki ferðast neitt.

Kóðabútur 13.7: Aðferðir kynntar með lestarkerfi

```
1  class Stod():
2     def __init__(self, nafn, fjarlaegd):
3        self.nafn = nafn
4        self.fjarlaegd = fjarlaegd
5
6  class Lest():
7     def __init__(self, leid, byrjunar_stod):
8        self.leid = leid
9        self.nuverandi_stod = byrjunar_stod
10       self.farnir_km = 0
11
```

```
12     def fara_til_numer(self, numer):
13        return abs(self.leid[numer].fjarlaegd - self.nuverandi_stod.fjarlaegd)
14
15     def fara_til_stod(self, stod):
16        return abs(stod.fjarlaegd - self.nuverandi_stod.fjarlaegd)
17
18     def fara_til_stodvarnafn(self, stodvarnafn):
19        for stod in self.leid:
20           if(stod.nafn == stodvarnafn):
21              return abs(stod.fjarlaegd - self.nuverandi_stod.fjarlaegd)
```

Skoðið hér að þrjár aðferðir eru til þess að segja til um fjarlægð lestar frá stöð og það er til þess fallið að sýna að oft eru margar leiðir til þess að leysa verkefni, sérstaklega þegar þau verða flóknari. Fara til aðferðirnar taka við mismunandi inntaki en þær skila allar sömu niðurstöðu, það er hvað er langt á milli lestar og stöðvar. Síðasta aðferðin er frábrugðin að því leitinu til að hún færir lestina, framkvæmir breytingu á ástandi lestarinnar.

Kóðabútur 13.8: Tilvik af lestum og stöðvum búin til og notuð

```
1  reykjavik = Stod("Reykjavík", 0)
2  borgarnes = Stod("Borgarnes", 76)
3  akureyri = Stod("Akureyri", 388)
4  egilsstadir = Stod('Egilsstaðir', 636)
5
6  leid1 = [reykjavik, borgarnes, akureyri, egilsstadir]
7
8  lest1 = Lest(leid1, reykjavik)
9
10 print(lest1.fara_til_numer(3))
11 print(lest1.fara_til_stod(egilsstadir))
12 print(lest1.fara_til_stodvarnafn('Egilsstaðir'))
13
14 print(lest1)
```

```
1 636
2 636
3 636
4 <__main__.Lest object at 0x7f29f845fb50>
```

Það er gott að sjá að allar aðferðirnar skiluðu sömu niðurstöðinni þegar spurt var hve langt er frá Reykjavík til Egilsstaða en af hverju fengum við svona ljóta útprentun þegar við prentuðum út lestina okkar? Sjáum hvernig við leysum það í næsta kafla.

13.4 Töfra aðferðir

Nú höfum við séð hvernig á að skilgreina okkar eigin aðferðir á klasa. Og við höfum verið að nota eina töfraaðferð til þess að smíða klasana okkar, init. En það er til mýgrútur af töfraaðferðum sem við getum nýtt okkur til þess að gera klasana okkar nothæfari. Í þessum kafla verða nokkrar slíkar teknar fyrir (en alls ekki allar). Við munum að töfraaðferðir eru aðferðir sem eru með tveimur undirstrikum fyrir framan sig og aftan og gegna því hlutverki að útfæra innbyggða virkni.

Helst ber að nefna `__str__` aðferðina, sem nemendur vilja oftast geta beitt strax og skilja ekki hvers vegna print skilar einhverju furðulegu. Hingað til höfum við ekki verið að beita innbyggða fallinu print á klasana okkar í kóðabútum því að hún gerir ekkert skilmerkilegt ennþá (eins og í úttaki kóðabúts 13.8). Til þess að hún geri það þurfum við að útfæra töfraaðferðina `__str__`. Það sem sú aðferð þarf að gera er að skila streng. Nú er það upp á okkur komið hvað okkur finnst vera nógu merkilegar upplýsingar til þess að setja í strenginn sem á að prenta. Hingað til þegar við beitum print fallinu höfum við verið að skoða úttak sem er af einhverri týpu sem við þekkjum,

heiltölur eða strengir til dæmis. En nú þegar við erum með okkar eigin klasa/hluti viljum við kannski fá einvherjar tilteknar upplýsingar í ákveðinni röð.

Skoðum kóðabút 13.9 þar sem við skilgreinum klasa sem heldur utan um tölvuleikjapersónuna okkar aftur, hins vegar ætlum við að sleppa aðferðunum að sinni og bæta við nokkrum klasabreytum. Klasabreytur eru skilgreindar efst í klasa og er nafnavenjan með þá að nota eingöngu hástafi. Það sem klasabreytur gera fyrir okkur er að halda utan um breytur sem við viljum að séu aðgengilegar allsstaðar í klasanum, við viljum ekki endilega að þær séu hluti af inntaki frá notanda við smíð klasans og þeir gera yfirferð og prófun klasans auðveldari. Með auðveldari prófunum er átt við að gildi séu ekki harðkóðuð víðsvegar og erfitt að skipta þeim út (eins og ef nota ætti ákveðna námundun á pí) heldur eru þau skilgreind á einum stað og auðvelt að átta sig á notkun þeirra (ef breytuheitin eru skýr).

Kóðabútur 13.9: Töfraaðferðin __str__

```
class Leikur():
   HAMARKS_LIF  = 100
   LAGMARKS_LIF = 0
   HAMARKS_PENINGUR = 9999
   LAGMARKS_PENINGUR = -9999

   def __init__(self, nafn, peningur, lif):
      self.nafn = nafn
      if(lif > self.HAMARKS_LIF or lif < self.LAGMARKS_LIF):
         self.lif = 100
      else:
         self.lif = lif
      if(peningur > self.HAMARKS_PENINGUR or peningur < self.LAGMARKS_PENINGUR):
         self.peningur = 0
      else:
         self.peningur = peningur

   def __str__(self):
      return "Persónan heitir {} og á {} gullpeninga og hefur {} í líf".format(self.nafn,
    self.peningur, self.lif)

valborg = Leikur('Valborg', 200, 90)
groblav = Leikur('Groblav', 1000000, -44444)
print(valborg)
print(groblav)
```

```
Persónan heitir Valborg og á 200 gullpeninga og hefur 90 í líf
Persónan heitir Groblav og á 0 gullpeninga og hefur 100 í líf
```

Ef þessarar str töfraaðferðar nyti ekki við þá væri úttakið á þessa leið *<__main__.Leikur object at *minnissvæði**. Einnig er nýtt í þessum kóðabút að við vinnum eitthvað með inntakið frá notandanum áður en við stillum tilviksbreyturnar. Þetta er ekki gert á nógu tryggan máta og við munum sjá í kafla 14 hvernig má meðhöndla inntak frá notanda þannig að vafalaust sé um rétt inntak að ræða. En við ætlum enn sem komið er að skoða hlutina á einfaldan og brothættan máta því við erum að kynnast svo mörgu nýju og óþarfi að gera allt kórrétt frá upphafi, mikilvægara er að byggja upp skilning.

Það sem töfraaðferðirnar gera er að gera okkur kleyft að beita innbyggðum föllum eins og print og len á tilvik af klösunum okkar, og að beita hinum ýmsu virkjum (reikni-, samanburðar- og rökvikjum) milli tilvika eða annara gilda.

13.5 Erfðir

Klasarnir okkar hafa hingað til verið skilgreindir með tómum svigum sem segir vélinni að þeir erfi ekki frá neinum klasa nema *object* sem gerði það að verkum að við gátum útfært töfraaðferðir.

Nú ætlum við að skoða í kóðabút 13.10 hvernig á að búa til **yfirklasa** (e. superclass) og **undirklasa** (e. subclass). Við skoðum dæmi þar sem prentari er tekinn fyrir, það sem hann þarf að kunna að gera er að prenta út streng, segja til um blekhlutfallið sitt og minnka blekið um eitt prósentustig. Þetta er alfarið æfing og því ekki endilega mjög raunhæft dæmi, en þar sem við erum að reyna að átta okkur á því hvað erfðir eru þá ætlum við að gera ráð fyrir því að við viljum að allir prentararnir okkar byrji með 100% af bleki og hafi möguleikann á að lækka það. Hins vegar er það ekki útrætt hvernig eigi að fara að því að prenta út og því ætlum við að útfæra sérstaka prentara sem eru eins og grunnprentarinn okkar (með tilliti til bleks) en meðhöndlar prentun á annan máta.

Undirliggjandi ástæður fyrir því að við myndum vilja gera þetta er sú að við viljum að einhver grunn virkni sé til staðar og sé aðgengileg, en það er einhver tiltekin virkni sem við viljum að sé öðruvísi. Tökum dæmi um spilastokk með 52 spilum og við viljum útfæra nýtt spil sem minnir á ólsen ólsen nema hvað við breytum reglunni að áttan breytir ekki hún lit heldur lætur einhvern annan spilara draga tvö spil. Þá í staðinn fyrir að skrifa upp allar reglurnar í ólsen ólsen þá skrifum við bara „alveg eins og ólsen ólsen nema áttan er öðruvísi".

Kóðabútur 13.10: Erfðir kynntar með klasanum Prentari

```
class Prentari():
   BLEK = 100

   def prentun(self, strengur):
      print(strengur)

   def minnka_blek(self):
      self.BLEK -= 1

   def stada_bleks(self):
      print(self.BLEK)

import random
class HandahofsPrentari(Prentari):
   def prentun(self, strengur):
      handahof = random.randint(1,5)
         for i in range(handahof):
            print(strengur)

class InntaksPrentari(Prentari):
   def prentun(self):
      strengur = input('hvað viltu prenta? ')
      fjoldi = int(input('hversu oft viltu prenta það? '))
      for i in range(fjoldi):
         print(strengur)
```

Í kóðabút 13.10 er einungis verið að yfirskrifa aðferðina prentun því að það er aðferðin sem við vildum að væri með einhverjum sértækum hætti. Við vildum ekki bara prenta út einu sinni heldur fá notandann til að segja okkur hversu oft og hvað á að prenta, eða geta gert það handahófskennt oft.

Kóðabútur 13.11: Prentaraklasarnir notaðir

```
p1 = Prentari()
p1.prentun('Fyrsti klasinn')
p1.minnka_blek()
print(p1.BLEK)
p2 = HandahofsPrentari()
```

```
p2.prentun('Handahóf')
p2.minnka_blek()
p2.stada_bleks()
print(p2.BLEK)
p3 = InntaksPrentari()
p3.prentun()
print(p3.BLEK)
```

```
Fyrsti klasinn
99
Handahóf
Handahóf
Handahóf
Handahóf
99
99
hvað viltu prenta? Inntak
hversu oft viltu prenta það? 2
Inntak
Inntak
100
```

Fjölmótun

Tengt erfðum er þess virði að nefna *fjölmótun* (e. polymorphism) í Python. Því það er fráburgðið t.d. C++ og Java. Fjölmótun er sá eiginleiki að tagið sem inntakið okkar bindur okkur ekki lengur, notandinn á að geta sett inn ólíkar týpur án þess að skemma nokkuð.

Fjölmótun í Python virkar þannig að klasar þurfa ekki að erfa frá öðrum klösum til að haga sér eins og þeir. Þetta er vegna þess að þegar vélin athugar hvort að einhver hlutur eigi einhver tiltekin eigindi skoðar hún klasann og þá klasa sem hann erfir frá (í röð) og skilar þeirri útgáfu af eigindinu sem finnst.

Til dæmis HandahofsPrentari og eigindið stada_bleks(), þá er fyrst athugað innan klasans HandahofsPrentari og svo Prentari hvernig eigi að nota stada_bleks. Hins vegar ef við værum að vinna með eitthvað sem við vildum að hegðaði sér eins og prentari án þess að spá í öllu sem prentaraklasinn er hugsaður fyrir gætum við búið til hlut sem útfærir bara aðferðina stada_bleks og erfir ekki frá neinum. Hlutinn myndum við kannski kalla Blekathugun, og það sem aðferðin stada_bleks gerir í þeim klasa er að skrifa stöðu bleksins, á einhverju tæki sem vill notfæra sér þessa aðferð, í tölvupóst.

Ef við tökum praktískara dæmi þá er hægt að sjá fyrir sér klasa sem sér um að vinna með gögn og til þess að geta sent gögnin frá þessum klasa á ákveðinn máta má láta hann fá hlut í hendurnar sem útfærir *write* aðferð. Klasinn sem útfærir write aðferðina þarf ekkert að gera annað en að útfæra þessa einu aðferð á einhvern ákveðinn máta og þá er hægt að fullvissa sig um að gögnin hafi verið skrifuð á þann máta.

Svo ef við viljum eiga nokkra mismunandi klasa sem allir kunna mismunandi write aðferðir þá þurfum við bara að ganga úr skugga um að gangavinnsluklasinn okkar fékk write aðferðin sem við vildum nota úr viðeigandi klasa.

Þetta er kallað **duck typing** og bjóða ekki öll forritunarmál upp á það. Hugtakið kemur úr frasanum „if it looks lika a duck, quacks like a duck and walks like a duck, it's a duck". Hugmyndin er að klasinn sem útfærir einungis aðferðina write fyrir okkur er alveg jafn mikil önd eins og kóðasafnið *os* sem sér um að vinna með skrársafnið og skrifa í skjöl.

Ef við höldum áfram með dæmið um klasana sem útfæra write, þá gæti einn þeirra skrifað í skjal á tölvu úti í þýskalandi, einn sendir skjalið í tölvupósti og einn lætur talgervil lesa það upp í strætó leið 14. Upphaflegi gagnaklasinn veit ekkert um það heldur treystir bara á að fá einhvern hlut í hendurnar sem kann þessa aðferð sama hvernig hún er útfærð.

13.6 Æfingar

Æfing 13.1 Útfærið nýja aðferð í klasann `Lest` úr kóðabút 13.7 sem uppfærir núverandi stöðu lestarinnar. Aðferðin tekur við hlut af taginu `Stod` og notar hann til að uppfæra hvar lestin er staðsett, og hversu langt hún hefur farið.

Afritið kóðann úr kóðabútnum, og bætið þessari nýju aðferð inn í klasann og prófið hana. Til þess að prófa hana þurfið þið nýja stöð, athugið kóðabút 13.8 til að sjá hvernig það var gert. Til dæmis væri hægt að setja Höfn í Hornafirði sem er í 820km fjarlægð frá Reykjavík ef farið er norður. ■

Æfing 13.2 Notið klasann `Leikur` úr kóðabút 13.9 til þess að útfærið töfraaðferðirnar `__eq__` og `__lt__`, sem eru til þess að geta notað samanburðarvirkjana == og <. Búið svo til tvö eintök af klasanum og berið þau saman með þessum samanburðarvirkjum.

Skoðið vel kóðabútinn og sjáið hvernig `__str__` er útfærð, þar er einungis verið að skila streng. Svo skilagildið úr þessum aðferðum sem við erum að vinna í að útfæra hér eru fengið með þeim samanburðarvirkjum sem þau snúast um. ■

14. Villur og villumeðhöndlun

Hingað til þegar við fáum villur í kóðann okkar hefur hann hreinlega hætt keyrslu og við þurft að laga eitthvað. Í kafla 2.4 sáum við upptalningu á þeim helstu villum sem við getum lent í. Það sem við viljum hinsvegar geta gert er að bregðast við villum til þess að forritin okkar haldi áfram keyrslu þrátt fyrir að eitthvað hafi farið úrskeiðis. Við viljum geta sagt vélinni að reyna að gera eitthvað og ef henni tekst það ekki því að það myndi valda villu þá viljum við geta gert eitthvað annað og haldið áfram eða hætt.

14.1 Algengar villur

Byrjum á að rifja upp algengar villur og bætum nokkrum við:

- **Nafnavilla** - *NameError*, nafn á breytu var notað sem er ekki skilgreint
- **Inndráttar villa** - *IndentationError*, röngum inndrætti beitt
- **Málskipanarvilla** - *SyntaxError*, rangt tákn notað eða tákn notað vitlaust
- **Týpuvilla** - *TypeError*, týpan styður ekki aðgerðina sem er verið að framkvæma
- **Vísis villa** - *IndexError*, verið er að nota sætisvísi sem er ekki til í hlutnum
- **Gildisvilla** - *ValueError*, verið er að nota gildi sem er ekki til
- **Eigindavilla** - *AttributeError*, verið er að nota eigindi sem hluturinn á ekki til
- **Lyklavilla** - *KeyError*, verið er að ná í lykil sem er ekki til
- **Endurkvæmnisvilla** - *RecursionError*, þegar búið er að ná hámarks leyfilegri endurkvæmni án niðurstöðu
- **Staðvær-nafnavilla** - *UnboundLocalError*, þegar verið er að vísa í staðvært breytuheiti en það hefur ekki verið skilgreint á þeim stað í gildissviðinu
- **Inntaks/úttaks-villa** - *IOError*, þegar villa kom upp við meðhöndlun inntaks eða úttaks.

Ástæðan fyrir því að nefna nákvæmlega þessar villur en ekki allar sem eru skráðar í skjölun[1] (e. documention) Python forritunarmálsins er vegna þess að þessar villur eru líklegri en aðrar til að koma upp hjá byrjendum og við viljum geta tekið á þeim. Inndráttarvillur og málskipanarvillur er

[1] Opinbera skjölun Python

þó ekki hægt að grípa á keyrslutíma því að þær eru gripnar áður en keyrsla á sér stað og kóðinn hreinlega keyrir ekki neitt. Það er ágætt að hafa í huga að kóðinn okkar þarf að vera réttur og rétt upp settur til þess að geta keyrt yfirhöfuð.

Hinar villurnar viljum við kannski geta gripið og meðhöndlað svo að við getum haldið áfram með það sem við vorum að gera, við viljum ekki að notandinn sé allt í einu læstur úti eða forritið hætti alfarið keyrslu ef eitthvað minniháttar kemur upp eins og að inntakið frá notanda var ekki af réttri týpu eða ekki var hægt að kasta því í rétta týpu.

14.2 Að grípa villur

Til þess að grípa villur og meðhöndla þær þurfum við nokkur ný lykilorð. Þau eru **try**, **except**, **else**, **finally** eða *reyna*, *nema*, *annars*, *að lokum*. Við höfum séð else áður og það virkar nokkuð svipað í þessari stöðu. Það sem try gerir er það sem við viljum reyna á, það sem við höldum að muni valda villu. Við viljum geta reynt að keyra kóðann, til dæmis kalla á einhverja vefþjónustu eða kasta inntaki frá notanda, án þess að forritið hætti. Hins vegar ef að kóðinn sem við reyndum að keyra veldur villu þá getum við gripið hana með except klausu, þannig að við ætlum að reyna að keyra kóða nema ef það virkar ekki þá viljum gera eitthvað annað. Annars (else) ef það virkaði að keyra kóðann þá getum við gert eitthvað vitandi að það mun ekki valda villu. Svo að lokum getum við gert eitthvað burt séð frá því hvort það olli villu eða ekki, finally klausan mun alltaf keyrast.

Flæðiritið fyrir þessa hugmynd er nokkuð svipað skilyrðissetningum með if elif og else. Það kemur ein try setning, á eftir henni koma eins margar except setningar og við viljum (þar sem hver og ein er þá að taka á einhverri tiltekinni villu), þá má koma ein else setning og að lokum má koma ein finally setning. Hún keyrist sama hvað og er notuð til þess að framkvæma þá virkni sem verður að eiga sér stað, eins og til dæmis að loka skjali sem verið er að vinna í.

Ástæða þess að það er gagnlegt að vita hvað villurnar heita er að except klausurnar okkar geta gripið ákveðnar villur og ef sú tiltekna villa kom upp getum við tekið á nákvæmlega því tilfelli. Sjáum í kóðabút 14.1 hvernig á að beita þessum nýju lykilorðum og hvernig uppsetningin á þeim þarf að vera. Gerum ráð fyrir að breytan `tala` sé sett sem eitthvað eins og # í stað skiljanlegrar tölu.

ykku áfram

Prófið ykkur áfram með það.

Kóðabútur 14.1: Hvernig á að beita try - except - else

```
1  tala = input('veldu tölu ')
2  try:
3     tala = int(tala)
4  except:
5     print('þú gafst ekki upp neitt sem mátti túlka sem tölu')
6     tala = 0 # notum þá bara eitthvað annað gildi
7
8  print('talan sem þú ert með er', tala)
9
10 tala = input('veldu tölu: ')
11 try:
12    int(tala)
13 except TypeError:
14    print('ekki gekk að kasta í tölu útaf týpuvillu')
15 except ValueError:
16    print('ekki gekk að kasta í tölu útaf gildisvillu')
17 except AttributeError:
18    print('ekki gekk að kasta í tölu útaf eigindavillu')
19 except:
20    print('ekki gekk að kasta útaf einhverri annarri villu sem ekki er reynt að grípa
       sérstaklega')
21 else:
22    print('það gekk bara víst að kasta í tölu')
```

```
þú gafst ekki upp neitt sem mátti túlka sem tölu
talan sem þú ert með er 0
veldu tölu: #
ekki gekk að kasta í tölu útaf gildisvillu
```

Hér er fyrst einungis gripið ef einhver villa á sér stað en í seinni hlutanum er tekið á þremur mismunandi tilfellum áður en gefist er upp. Þetta gæti verið gott þegar þrjár tilteknar villur eru líklegar og þörf er á að taka á þeim.

Líklega er best að hafa try klausurnar hnitmiðaðar en ekki öll framkvæmdin í forritunu okkar, svona ef ské kynni að eitthvað gæti farið úrskeiðis einhvers staðar.

Kóðabútur 14.2: Hvernig á að beita try - except - else

```
try:
   int('strengur')
except TypeError:
   print('hér er tekið á villu sem á sér ekki stað')
except:
   print('hér er tekið á öllum öðrum villum, ef þessari klausu er sleppt munum við ekki
    grípa neina villu því þetta er vissulega ekki týpuvilla')
else:
   print('það er ljóst að við förum ekki hingað inn því kóðinn veldur villu')
finally:
   print('við förum alltaf hér inn sama hvað, hvort sem try virkaði eða ekki, jú nema við
    gleymdum að grípa villuna og forritið hætti keyrslu')
```

```
hér er tekið á öllum öðrum villum, ef þessari klausu er sleppt munum við ekki grípa neina
    villu því þetta er vissulega ekki týpuvilla
við förum alltaf hér inn sama hvað, hvort sem try virkaði eða ekki, jú nema við gleymdum
    að grípa villuna og forritið hætti keyrslu
```

Þegar við erum að reyna að grípa svona margar villur eins og í kóðabút 14.1 er það vegna þess að við erum ekki viss hvað það er sem mun fara úrskeiðis, try klausan okkar er tiltölulega einföld og því fátt sem kemur til greina til að fara úrskeiðis, en við gætum verið að reyna á margt í einu og því gagnlegt að vita af því að við getum verið að grípa margar villur á einu bretti.

Annar möguleiki sem við viljum geta reynt á er að hreiðra klausurnar okkar þannig að ef við reyndum eitthvað sem gekk ekki viljum við grípa það en reyna eitthvað annað. Þar kemur einnig sterkt inn að vita hvað villurnar okkar heita svo að við getum reynt eitthvað ákveðið byggt á því hvaða villu við fengum. Í kóðabút 14.3 sjáum við hvernig hægt er að halda áfram við að reyna að kasta inntaki þegar það gekk ekki við fyrstu tilraun.

Kóðabútur 14.3: Hvernig á má hreiðra try - except - else

```
try:
   tala = input('skrifaðu tölustaf ')
   tala = int(tala)
except:
   try:
      tala = int(tala[0])
   except:
      print('þú skrifaðir ekki tölu sem hægt var að skilja')
      tala = 0

print('talan var', tala)
```

```
skrifaðu tölustaf fimmtán
þú skrifaðir ekki tölu sem hægt var að skilja
talan var 0
```

Við viljum temja okkur að taka á villum, við viljum heldur sjá snyrtileg villuskilaboð sem væru lýsandi fyrir vandamálið en ekki stafasúpu af torskildum tækniupplýsingum.

Þetta nýtist t.d. ef við erum með forrit sem brothætt og við viljum að notendur skilji hvað fór úrskeiðis eða við erum með vél sem keyrir endalaust (eins og hitamælir á pallinum) sem sendir okkur svo tölvupóst ef hún lendir í villu.

14.3 Æfingar

Æfing 14.1 Búið til fall sem tekur við einu viðfangi og prentar út hvort það sé stærra eða minna en 0 (True eða False dugar, True fyrir stærra og False fyrir minna). Athugið að ekki er hægt að bera saman ólík tög og þið þurfið því að villumeðhöndla inntakið, ef það er ekki sambærilegt við 0 skulið þið í staðinn prenta ękki sambærilegt". Athugið sérstaklega fleytitölur, að ef fleytitölunni 0.1 er kastað í heiltölu verður hún að 0, að ef strengnum "0.1ęr kastað í heiltölu fæst villa, og því þurfið þið að vera á varðbergi fyrir mögulegum fleytitölum. ■

Æfing 14.2 Skrifaðu kóða sem grípur eftirfarandi villur:

a) Nafnavilla.

b) Eigindavilla.

c) Lykilavilla.

d) Gildisvilla.

■

15. Reiknirit

Reiknirit (e. algorithm) er forritsbútur sem sinnir sérhæfðum útreikningi. Dularfyllra er það ekki. Reiknirit sinna því ákveðnum tilgangi og eru þau oft í grunninn stærðfræðlegs eðlis.

Dæmi um reiknirit sem við höfum séð áður í þessari bók væri útfærsla á fjarlægð milli lesta og að setja nýja lestarstöð inn á leið lestar í kóðabút 13.7.

Ástæðan fyrir því að nauðsyn þykir að kynna reiknirit í bók sem þessari er að ef nemendur hafa áhuga á að kynna sér tölvunarfræði í framhaldssnámi er gott að hafa fengið nasasjón af því hvað felst í að beita reikniritum og útfæra þau. Margir nemendur hefja nám í tölvunarfræði með ýmsar forhugmyndir sem eiga sér sumar ekki stoð í raunveruleikanum. Þessi kafli og sá næsti fjalla um þau atriði sem leikmenn átta sig ekki endilega á að séu stór hluti af tölvunarfræði og hugbúnaðarþróun; stærðfræði og samvinna. Þessi kafli er um stærðfræðilegu hliðina og næsti um samvinnuna.

Það sem við ætlum að skoða í þessum kafla eru nokkur grundvallar reiknirit, og hugmyndin um endurkvæmni.

15.1 Röðun

Áður en við getum farið að leita ætlum við að raða. Ef við ímyndum okkur að nokkur spil úr spilastokk séu fyrir framan okkur, þau snúa upp svo við sjáum hvaða spil þetta eru en við sjáum einnig að þau eru ekki í réttri röð. Sú röð gæti verið hver sem er, en gerum ráð fyrir, til einföldunar, að þau séu ekki í vaxandi röð með lægsta spilið vinstra megin og hæsta spilið hægra megin. Hvað er það fyrsta sem við gerum? Það er ekki eitt svar við því. Hugsum okkur hér hvað sé fyrsta skrefið og svo næsta og koll af kolli. Reynum svo að lýsa því, eins og með hnetusmjörssamlokuna í kafla 1.4, þannig að við gætum snúið baki við einhverjum spilum og lýst fyrir einhverri annarri manneskju hvernig hún gæti farið að því að raða spilunum með aðferðinni okkar.

Hér er sterklega mælt með því að þið reynið á þessa æfingu áður en lengra er haldið.

Ein tiltekin leið til að raða spilunum væri að bera saman fremstu spilin og skipta þeim ef það hærra er vinstra megin og gera það fyrir öll spilin út röðina, skoðum töflu 15.1 til að sjá hvernig það

upphafsstaðan	**K**	**10**	4	8
	10	*K*	**4**	8
	4	10	*K*	**8**
lokastaðan	4	8	10	*K*

Tafla 15.1: Hér sést hvernig fjórum spilum er raðað í vaxandi röð. Í hverri stöðu er einungis verið að skoða takmarkaðan fjölda spila til að raða, þau spil eru sýnd með grænum lit í töflunni. Þau spil sem búið er að raða eru sýndi með fjólubláum lit.

Búið að raða þessum hluta		Á eftir að raða	
minna eða jafnt x	stærra en x	x	???

Í næsta ástandi er röðun orðin:

Búið að raða þessum hluta			Á eftir að raða
minna eða jafnt x	x	stærra en x	???

Tafla 15.2: Hér sést hvernig staðan er þegar á að finna stað fyrir eitthvað x, sem gæti verið tala eða spil eða einhver annar hlutur. Athugið að annað hvort eða bæði hólfin gætu verið tóm sem búið er að raða og einnig gæti verið ekkert sem eftir á að raða.

myndi ganga fyrir sig. Þar erum við með einhverja ákveðna upphafsstöðu á spilunum og byrjum á að skoða fremstu tvö spilin, þá „svissum" við þeim til að hærra spilið sé hægra megin. Í næstu atrennu ætlum við þá að skoða spilið sem er næst í röðinni og við „svissum" því þangað til að það er annað hvort úti í enda eða orðið hærra en það sem er vinstra megin. Þá vegna þess að við erum bara með fjögur spil skoðum við síðasta spilið og látum það berast niður spilaröðina þar til það passar.

Reikniritið sem lýsir þessari röð aðgerða heitir *insertion sort* eða innsetningarröðun og í kóðabút 15.1 ætlum við að skoða hvernig skiptingarnar eiga sér stað. Á mynd 15.2 sést hvað er að gerast í einhverju tilteknu skrefi þar sem þarf að finna stað fyrir næsta stak sem á að raða, x.

Skoðum nú kóðann á bakvið þetta reiknirit.

Kóðabútur 15.1: Insertion sort reikniritið

```
1 def insertion_sort(A):
2     #raðar í vaxandi röð
3     i = 1
4     while i < len(A):
5         j = i
```

```
6         while j > 0 and A[j-1] > A[j]:
7            #skiptum stökunum ef þau eru ekki rétt röðuð
8            aj = A[j]
9            ajminus = A[j-1]
10           A[j] = ajminus
11           A[j-1] = aj
12           j = j - 1
13        i = i + 1
```

Nokkrar hugmyndir til að ná upp leikni í notkun á reikniritum er að:

- Reyna að nota reikniritið fyrir einhvern lista af tölum.
- Reyna að finna leið til þess að gera línur 8-11 í einni línu, Python býður upp á það.
- Reyna að láta reikniritið raða í minnkandi röð.

Í kóðabút 15.2 sjáum við útfærslu á bubble sort. Útfærslan felst í tveimur hreiðruðum forlykkjum[1].

Reikniritið virkar í grunninn þannig að það tekur við lista sem á að raða, það rúllar í gegnum listann frá upphafi og út í enda og ýtir stærsta stakinu út í enda. Eins og loftbólur sem þrýstast upp á yfirborðið. Þegar það er búið að rúlla einu sinni í gegnum listann er stærsta stakið komið út í enda og það stak er ekki skoðað aftur heldur álitið á sínum stað. Þá er aftur rúllað í gegnum listann og stakið sem er þá stærst fer út í enda vinstra megin við stakið sem var stærst.

Þannig að ytri for lykkjan keyrir fyrir hvert stak í listanum, eða segir til um hversu oft þurfi að finna stærsta stakið, og innri lykkjan sér um samanburðinn og skiptingarnar. Tökum sérstakleg eftir þar að við getum horft á næsta stak hægra megin þegar við erum að bera saman og það er vegna þess að við hættum fyrir framan aftasta stakið hverju sinni í innri lykkjunni.

Kóðabútur 15.2: Bubble sort reikniritið

```
1     def bubblesort(listi):
2     # n er þá fjöldi staka í listanum
3     n = len(listi)
4
5     # Förum í gegnum öll stökin
6     for i in range(n):
7        # vinstri hliðin er óröðuð
8        # í fyrstu ítrun er i 0 og vinstri_hlid er því jöfn n-1
9        # sem er síðasta stakið í listanum
10       # svo verður vinstri_hlid alltaf minni og minni
11       # Því síðustu i stökin eru komin á sinn stað
12       vinstri_hlid = n-i-1
13
14       for j in range(0, vinstri_hlid):
15          vinstra = listi[j]
16          haegra = listi[j+1]
17          # Hér förum við frá 0 upp í n-i-1
18          # af því að viljum byrja úti í vinstri enda
19          # og við viljum geta skoðað næsta stak fyrir aftan
20          #
21          # Svo skiptum við á stakinu við stakið hægra megin
22          # ef stakið er stærra en það sem er hægra megin
23          if vinstra > haegra:
24          listi[j] = haegra
25          listi[j+1] = vinstra
26
27    return listi
```

[1] sem er yfirleitt ekki góðs viti þegar kemur að tímaflækju, enda er hægt að gera ráð fyrir að tíminn sem það tekur að keyra bubble sort sé n^2, sem segir kannski ekkert fyrir óþjálfað auga en hægt er að treysta því að það er ekki ákjósanlegt.

15.2 Helmingunarleit

Nú þegar við kunnum að raða þá getum við farið að leita. Skoðum nú eitthvert þekktasta reiknirit sem til er. Helmingunarleit að tölu á bili. Hugsum okkur að við séum með raðaðan lista af tölum og við viljum finna eina tiltekna tölu. Ef við ættum að skoða hverja einustu tölu í listanum til að finna hana þá tæki það mjög langan tíma. Eða allavega fyrir okkur sem manneskjur, en allur tímasparnaður er góður. Því að aðgerðin „að skoða spil" kostar einhvern tíma og því færri þannig aðgerðir sem við getum gert því hraðara er reikniritið okkar[2].

Hugmyndin er sú, að verið sé að skoða í fyrstu allan listann sem er raðaður (mjög mikilvægt, annars virkar þetta engan veginn) og miðju gildið er skoðað, ef gildið sem við leitum að er stærra en miðjugildið þá eigum við að vera að leita þeim megin við miðjugildið sem stærri gildi eru. Svo gerum við þetta endurtekið, helmingum alltaf vandamálið þar til við höfum annað hvort fundið gildið sem við leitum að eða bilið sem við erum að leita á inniheldur engin stök.

Kóðabútur 15.3: Helmingunarleit að tölu í röðuðum lista með lykkju

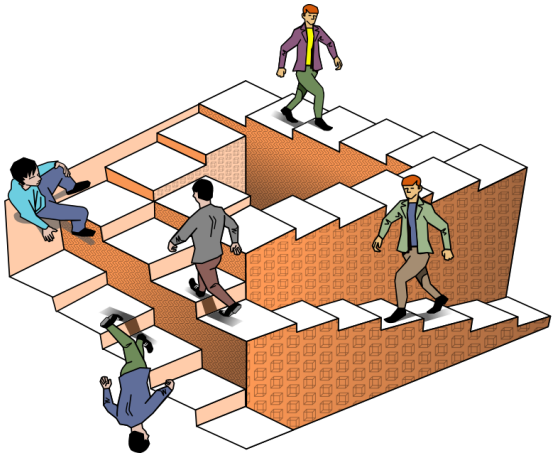
```
def helmingunarleit_med_lykkju(listi, x):
# Þetta fall tekur við lista sem er raðaður í vaxandi röð
# og gildi sem á að leita að í listanum
# Þetta er gert með lykkju og án endurkvæmni

# Fallið skilar sætisvísinum sem gildið er í eða -1 ef gildið er ekki í listanum

# minnsti og hæsti vísirinn í listanum
minnsti = 0
haesti = len(listi)-1

while minnsti <= haesti:
   # Þetta keyrist á meðan minnsti er enn minni eða jafn og haesti, það er við erum
 enn með bil til að leita á

   midjan = int((minnsti + haesti)/2)
   if listi[midjan] == x:
      # við fundum gildið og skilum vísinum sem það er í
      return midjan
   if listi[midjan] > x:
      haesti = midjan - 1
   else:
      minnsti = midjan + 1

# lykkjan hætti keyrslu svo minnsti vísirinn er orðinn hærri en sá hæsti og þá vitum
 við að talan er ekki í listanum
# við skilum því tölunni -1 til að segja að það var enginn vísir sem x var í
return -1
```

15.3 Endurkvæmni

Endurkvæmni (e. recursion) er sú virkni forrits að vísa í sig sjálft. Þið þekkið eflaust listaverk sem virka eins og skynvillur, þar sem manneskja getur labbað í hring upp stiga en endað á sama stað því stiginn fer í raun í hring. Eða þið hafið séð ykkur sjálf í spegli þar sem var annar spegill fyrir aftan og þið sáuð ótal spegilmyndir raðast af ykkur.

Reynum að átta okkur vel á þessu áður en við reynum að beita því. Tökum aðeins óhlutbundnara dæmi. Ímyndum okkur að börn standi í röð og bíði eftir fríum ís, manneskjan sem sér um ísinn

[2]Tölvunarfræðingar eru mjög uppteknir af því hvað aðgerðir og reikningar taka mikinn tíma. Þetta er kallað tímaflækja (e. time complexity) og er tölvunarfræðingum mjög hugleikin. Tímaflækja helmingunarleitar er sérstaklega lág eða $log_2 n$ vegna þess að reikniritið helmingar alltaf vandamálið (e. problem space).



Mynd 15.1: Mynd í anda Escher fengin af openclipart.org

snýr sér að fremasta barninu og segir „hvað viljiði marga íspinna?", barnið vill einn en veit það ekki hvað hin börnin vilja svo það snýr sér að barninu fyrir aftan sig og segir „hvað vilt þú marga?" það barn gerir það sama þangað til þau eru komin út í enda, það barn hefur ekkert annað barn til að spyrja og segir bara „tíu íspinna takk", þá veit næst aftasta barnið að það séu allavega 10 íspinnar og svo 2 fyrir það sem þau vilja, og þannig leggjast tölurnar saman þar til fremsta barnið getur sagt "59 íspinna takk fyrir okkur."

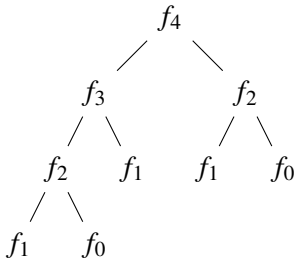
Gott dæmi um hvernig megi beita endurkvæmni til að fá skilmerkilega niðurstöðu er að útfæra fall sem reiknar fyrir okkur einhver gildi í fibonacci röðinni. En áður en við gerum það skulum við skoða enn einfaldara dæmi þar sem við erum með fall sem kallar í sig sjálft og gerir ekkert annað en það. Í kóðabút 15.4 sjáum við einfalda útgáfu af endurkvæmni, þar sem hugmyndin er í raun kynnt án þess að fallið sé neitt gagnlegt. Það eina sem fallið gerir er að athuga hvort að tala sé stærri en núll og ef hún er það þá kallar fallið í sig sjálft með gildi sem er einum lægra, annars ef talan er ekki stærri en núll prentast „við kunnum á endurkvæmni". Hveru oft ætli það prentist ef við setjum inn töluna 5?

Kóðabútur 15.4: Endurkvæmni - einfalt

```
1  def endurkvæmt_fall(tala):
2      if(tala > 0):
3          # á meðan talan er hærri en 0 þá köllum við aftur í fallið
4          # en við köllum í það af gildi einu lægra
5          return endurkvæmt_fall(tala-1)
6      else:
7          # hér erum við komin niður í 0 og prentum eftirfarandi texta
8          print('við kunnum endurkvæmni')
9
10 endurkvæmt_fall(5)
```

```
1  við kunnum endurkvæmni
```

Við sjáum í kóðabút 15.4 að þó að við kölluðum í fallið með tölunni 5 þá fengum við bara einu sinni út strenginn „við kunnum endurkvæmni". Það er vegna þess að við kölluðum í fallið fyrir fimm og það sem fallið gerir fyrir okkur er að klára endurkvæmnina fyrir það kall, það verða ekki til fjögur önnur köll. Heldur verður fimm að fjórum sem skilar okkur svo niðurstöðunni fyrir þrjá og svo koll af kolli þar til við erum komin niður í núll og þá hættir reikniritið keyrslu. Í þessu tilfelli skilar það engu til baka upp fallakallið en klárast engu að síður þarna í línu 8 þegar það kemst þangað.



Mynd 15.2: hér sést hvernig fibonacci fallið kallar alltaf í tvö gildi. Svo eru neðstu gildin, gildin sem vísa ekki í neitt annað, þau sem leggjast saman.

Það sem þetta reiknirit okkar gerir ekki er að skila einhverri niðurstöðu til baka, en nú skulum við skoða endurkvæmt reiknirit sem skilar summu í 15.5. Það reiknar summu af einhverri tölu og öllum jákvæðum tölum lægri en henni, svo talan 5 gæfi útkomuna 5 + 4 + 3 + 2 + 1 sem er 15. Endurkvæmnin er því fólgin í því að upphaflega fallakallið með summa(100) skilar okkur 100 + summa(99), sem skilar okkur 100 + 99 + summa(98) og svo koll af kolli þar til talan 1 fæst, og við fáum útreikninginn 100 + 99 + ... + 1 sem skilar okkur 5050.

Einnig skulum við skoða fall sem reiknar n-tu töluna í fibonacci rununni. Endurkvæmnin í fib(4) er fólgin í því að skoða alltaf tvær summur í einu en þær skila sér í sama fallakallið. Svo fib(4) skila því sem fib(2) + fib(3) skila. Kóðann má sjá í kóðabút 15.6 og á mynd 15.3, nú erum við ekki lengur að skila einu gildi og hala af útreikningi heldur erum við að skila tveimur hölum.

Kóðabútur 15.5: Summa reiknuð endurkvæmt

```
def summa(n):
   if n <= 1:
      return 1
   else:
      return n + summa(n-1)

summa(6)
```

```
21
```

Kóðabútur 15.6: Fibonacci tölur reiknaðar endurkvæmt

```
def fib(n):
   if n <= 1:
      return n
   else:
      return fib(n-2) + fib(n-1)

fib(4)
```

```
3
```

Nú höfum við séð lykkju útfærslu á helmingunarleit sem keyrir á meðan við höfum bil til að leita á en nú skulum við skoða hvernig megi gera þetta endurkvæmt.

Kóðabútur 15.7: Helmingunarleit að tölu í röðuðum lista með endurkvæmni

```
def helmingunarleit_med_endurkvaemni(listi, vinstri, haegri, x):
   # Endurkvæmt fall sem skilar sætisvísinum sem x finnst í í listanum listi
   # Ef x er ekki að finna í listanum listi þá fæst -1
```

```
4
5     # Athugum hvort að vinstri sé enn vinstra megin
6     if haegri >= vinstri:
7
8        # Stillum miðjuna
9        midjan = int((vinstri + haegri)/2)
10       #midjan = math.floor(vinstri + (haegri - vinstri)/2)
11
12       # Athugum hvort gildið sé í miðjustakinu
13       if listi[midjan] == x:
14          return midjan
15
16       # Ef gildið er lægra en miðjan þá þurfum við að leita frá vinstri að miðju
17       elif listi[midjan] > x:
18          return helmingunarleit_med_endurkvaemni(listi, vinstri, midjan-1, x)
19
20       # Annars hlýtur gildið að vera frá miðju að hægra gildi
21       else:
22          return helmingunarleit_med_endurkvaemni(listi, midjan+1, haegri, x)
23
24    else:
25       # Nú er hægra orðið minna en vinstra svo að
26       # við erum búin að leita af okkur allan grun
27       # gildið er ekki í listanum
28       return -1
```

15.4 Æfingar

Æfing 15.1 Búið til fall sem tekur við tveimur hnitum, x og y gildum. Það sem fallið á að gera er að skila tölu byggt á því hversu margar hreyfingar hrókur, taflmaðurinn, þyrfti að fara til að komast úr einu hniti í annað. Í boði eru þrír möguleikar, tvær, ein eða engin hreyfing.

Gerum ráð fyrir að öll möguleg rauntöluhnit séu lögleg fyrir þennan ofurhrók. ■

Æfing 15.2 Við viljum geta sagt til um hvaða leið er styst og hver er lengst, gefið er ákveðin lestarleið í formi lista af tölum. Tölurnar tákna hversu langt stoppin eru frá upphafstöðinni. Þið eigið annars vegar að búa til fall sem tekur við lista og skilar stystu fjarlægð milli tveggja nærliggjandi stöðva og hins vegar eigið þið að búa til fall sem tekur við sambærilegum lista en skilar lengstu fjarlægð milli tveggja nærliggjandi stöðva.

Athugið að fyrir listann `[1,3,7,19]` væri svarið 2 fyrir fyrra fallið en 12 fyrir seinna fallið. Athugið hér sérstaklega bubblesort og hvernig á að skoða tvö stök í einu. ■

Æfing 15.3 Búið til fall sem spyr notandann um fjölda barna sem eru að biðja um íspinna, innan þess falls skulið vera með fall sem endurkvæmt reiknar út summu íspinna sem börnin vilja. Notið random kóðasafnið til að búa til einhvern fjölda íspinna sem hvert barn vill. ■

Eins og fram kom í síðasta kafla þá snýst endirinn á þessari bók um framhaldið fyrir nema sem vilja leggja land undir fót í tölvunarfræðum. Þessi kafli er þó meira í anda hugbúnaðarverkfræði en tölvunarfræði.

Það kemur nefnilega mörgum á óvart hvað góð samskipti eru stór hluti af því að þróa hugbúnað, forhugmyndir um fólk sem situr eitt við tölvuna sína og gerir eitthvað alveg af sjálfdáðum eru sterkar.

Hugmynd þarf einhvern veginn að verða að veruleika og það má vel vera að hugmynd að góðum og nothæfum hugbúnaði hafi sprottið upp hjá einum einstaklingi en flestur hugbúnaður sem við notum í dag, eins og forrit á símum, eru yfirleitt ekki hönnuð, forrituð, prófuð og markaðssett af einni manneskju. Þess vegna er þess virði að taka fyrir stuttan kafla um hvað felst í hugbúnaðarþróun.

Áður en lengra er haldið er þó rétt að taka fram að það er engin ein „rétt" leið til að þróa hugbúnað, fólk verður að prófa sig áfram til þess að finna hvað hentar.

16.1 Útgáfustjórnun

Fyrir það fyrsta er útgáfustjórnun (e. version control) nauðsynleg. Ekki bara mikilvæg, nauðsynleg. Hellingur af lausnum er til, opinn hugbúnaður sem og lokaður, sem sinnir þessu mikilvæga hlutverki.

Útgáfustjórnun kannast kannski sum við sem hafa notfært sér Word úr Office pakkanum, að geta rúllað til baka í einhverja útgáfu af tilteknu skjali þegar það skemmist eða gögn tapast skyndilega. Það er í raun allt og sumt. Að geyma kóðann á bakvið hugbúnaðinn þar sem allir eiga að hafa aðgang hafa viðeigandi aðgang[1].

Burt séð frá aðgangsmálum þá er útgáfustjórnun falin í því að gera litlar breytingar því stór jafnt sem lítil kerfi geta verið brothætt og því mikilvægt að geyma þær breytingar sem við gerum í litlum skrefum svo það sé hægt að snúa við og hætta notkun einhverra breytinga sem komu í ljós að hafa valdið villum. Einnig gerir útgáfustjórnun kóðarýni (e. code review) auðveldari. Kóðarýni er mikivægur hluti af hugbúnaðarþróun, þar sem einhver er fenginn til að fara yfir kóða frá öðrum og

[1] Viðeigandi aðgangur gætu verið t.d. skrifréttindi og lesréttindi, að þau sem þurfa ekki að geta breytt neinu geta bara lesið og þau sem eiga að geta gert breytingar hafa réttindi til að skrifa.

finna hvað mætti betur fara.

Þessi bók var skrifuð með aðstoð git útgáfustjórnunartólsins og gitbub hýsingaraðilans. Fleiri góð tól eru til eins og Bitbucket, SVN, Mercurial og önnur sem eru innbökuð í hugbúnaðarþróunartól (e. IDE eða integrated development environment) eins og VisualStudio. Aðalatriðið í vali á tóli fyrir útgáfustjórnunina er að það henti öllum þeim sem eiga að koma að þróuninni, passi við þau stýrikerfi sem fólk notar, og best er ef fyrri reynsla er góð.

16.2 Stefnur og straumar

Alls konar hugmyndafræði hefur legið til grundvallar við gerð hugbúnaðar og stórra kerfa. Til eru nokkuð stórar stefnur innan hugbúnaðarþróunar og er ein vinsælasta hugmyndafræðin kvik þróun eða agile með sinn eiginn undirkafla.

En það þýðir ekki að það séu ekki til fleiri aðferðafræðir og í öllum þeim er hornsteinninn teymisvinna forritara.

Þegar hugbúnaður var fyrst þróaður á 20. öldinni þá voru verkfræðingar og stærðfræðingar í fararbroddi. Því þarf ekki að koma á óvart að verkfræðileg nálgun varð vinsæl stefna í þróun hugbúnaðar, sú sem mest var beitt heitir fossalíkanið (e. waterfall model). Fossalíkanið byggir á því að komast að því hverjar þarfir og skorður eru á verkefninu, hanna út frá því vöru og prófa hana svo, afurðin er fullbúin vara. Þessi hugmyndafræði virkar fyrir hin ýmsu verkefni þar sem hægt er að vita skorður og þarfir á mjög hnitmiðaðan, skýran og óyggjandi máta. Eftir því sem notendur fengu meira vægi þá þurfti að gera breytingar á því hvernig hugbúnaður var þróaður og fékk önnur hugmyndafræði að ryðja sér rúms, samfelld þróun (e. continuous development) þar sem stöðugt var verið að líta til baka og gera breytingar (þetta mun hljóma afskaplega svipað agile en það eru þó einhverjir megin drættir sem eru ólíkir).

Eftir því sem fleiri fóru að þróa hugbúnað sem leið á öldina því fleiri hugmyndafræðilegir ágreiningar komu í ljós. Önnur hugmyndafræði sem naut mikilla vinsælda var prófunarþróun (e. test driven development eða TDD) sem er ennþá við lýði í dag. Hún snýst um að skorður og þarfir má prófa stöðugt og prófanirnar sýna að hugbúnaðurinn standist þær kröfur sem hann eigi að standast. Gallinn við þá nálgun er að prófanirnar eru mögulega ekki nógu yfirgripsmiklar og eitthvað lendir á milli og gleymist.

16.3 Kvik þróun - agile

Kvik þróun fær sér kafla ekki til að setja þessa hugmyndafræði á einhvern stall heldur því hún er svo ótrúlega vinsæl. Kvik þróun hefur verið notuð bæði sem haldbær þróunaraðferð en einnig sem tískuorð (e. buzz word).

Agile snýst um samfelldan þróunarferil þar sem litlar breytingar eru teknar inn á afmörkuðu tímabili, og þessi tímabil, kölluð sprettir (e. sprints), eru endurtekin þar til hugbúnaðinum er skilað (nema auðvitað honum sé viðhaldið). Afurðin er því enn í vinnslu þó henni sé sleppt í hendur notenda.

Grunnurinn byggir á fjórum hornsteinum:

- Einstaklingar ofar tólum.
- Hugbúnaður sem virkar ofar yfirgripsmikilli skjölun.
- Samskipti ofar samningum.
- Viðbregðni ofar því að fylgja plani.

Ásamt þessum hornsteinum eru tólf grunngildi sem snúa að því hvernig eigi að vinna sem teymi, líta til baka og gera endurbætur.

Þessar grunnhugmyndir eru nokkuð opnar sem hefur leitt til þess að til eru nokkuð mörg afbrigði (e. flavor) af Agile hugmyndafræðinni. Öll afbrigðin eiga það sameiginlegt að taka hornsteinana frekar heilaga en leggja áherslur hver á sín grunngildi eða túlka þau mismunandi.

Helst af afbrigðum ber að nefna SCRUM, XP og Kanban. SCRUM (sem er ekki stytting á neinu, en er þó alltaf skrifað í hástöfum) þar sem mikil áhersla er lögð á hin ýmsu hlutverk þróunarferlisins. XP stendur fyrir „extreme programming" og byggist á því að fólk vinni mjög náið saman, helst tvö eða fleiri á einni tölvu. Kanban er ákveðið vinnuflæðirit sem má nýta með SCRUM og XP en einnig sem bara frjálsleg útfærlsa við Agile hugmyndafræðina.

Ástæða fyrir því að Agile er stundum notað sem tískuorð er vegna þess að þessi hugmyndafræði er það vinsæl að flest fyrirtæki vilja segjast hafa tileinkað sér hana. Þau eru það þó ekki öll, það er engin vottun til fyrir slíkt og eru dæmi um fyrirtæki sem segjast vera kvik en það tekur heilt ár fyrir starfsfólk að skrá sig úr mötuneytisáskrift.

16.4 Að lokum

Nú þar sem við höfum náð góðum tökum á grunninum í forritun í Pyhton og skoðað við hverju má búast ef farið er lengra er hægt að líta til annarra mála eða frekari notkun Python fyrir ákveðin verkefni. Þá er gott að skoða pip, PyPI, anaconda, og virtualenv.

Mikið er um góð námskeið og kennsluefni á netinu, eins og á udemy, khan academy, codewars, codecombat og svo er ýmislegt að finna á youtube.

Ekki hika við að gera mistök, þannig verðið þið betri í því sem þið takið ykkur fyrir hendur.

Kafli 2

Æfing 2.1

Nú þurfum við að athuga þrennt nafn á breytuna, hvernig á að veita henni eitthvað gildi og sjá til þess að það gildi sé heiltala. Til þess að búa til breytu þarf að finna nafn á hana, skrifa það fremst í línuna (ss. vinstra megin við jafnaðarmerkið) og nota svo jafnaðarmerkið til þess að veita breytunni gildi og því næst velja einhverja heila tölu. Það er tölu sem er annað hvort jákvæð eða neikvæð og er ekki með punkt og aukastafi.

```
1 # til dæmis
2 eitthvad_loglegt_breytuheiti_fyrir_heiltolu = 5
3 neikvaed_tala = -1234567890
```

Æfing 2.2

Hér þurfum við að hafa það sama í huga eins og í æfingu 2.1 nema að talan sem við veljum okkur verður að vera með punkti og einum eða fleiri aukastöfum.

```
1 # til dæmis
2 eitthvad_loglegt_breytuheiti_fyrir_fleytitolu = 5.0
3 neikvaed_tala = -12.9999
```

Æfing 2.3

Nú þurfum við að athuga að nota nákvæmlega sömu nöfn og við notuðum í fyrri verkefnum. Vegna þess að hér erum við að vísa í breytur sem búið var að skilgreina, við fáum nafnavillu ef við skrifum nöfn hægra megin við jafnaðarmerkið sem tölvan hefur ekki aðgang að í minni.

```
1 # til dæmis
```

```
thridja_breytan = eitthvad_loglegt_breytuheiti_fyrir_heiltolu +
    eitthvad_loglegt_breytuheiti_fyrir_fleytitolu
print(thridja_breytan)
```

Svo er einnig spurt af hvaða týpu er þessi þriðja breyta? Þar sem önnur breytan er fleytitala verður útkoman fleytitala, það má sjá með því að gera eftirfarandi:

```
type(thridja_breytan)
```

Æfing 2.4

Við beitum deilingu á breytuna allt, og svo gerum við það aftur fyrir nýju breytuna.

```
allt = 1000
helmingur = allt/2
helmingurinn_af_helmingnum = helmingur/2

# eða
allt = 1000
helmingur = allt/2
helmingur_af_helmingnum = allt/2/2  # eða allt/4 því helmginur af helmingnum er fjórðungur
```

Æfing 2.5

Við beitum deilingu á breytuna allt, og svo gerum við það aftur fyrir nýju breytuna.

```
allt = 1000
print(allt)

allt /= 2
print(allt)

allt /= 2
print(allt)
```

Æfing 2.6

Hér þurfum við að átta okkur á því að vélin býst við að nota breyturnar a og b en við eigum eftir að skilgreina þær sem eitthvað. Þekktur rétthyrndur þríhyrningur er til dæmis með hliðarnar 3, 4 og 5.

```
a = 3
b = 4

c_i_odru_veldi = a**2 + b**2
c = c_i_odru_veldi**0.5
print(c)
```

Þá er c 5 í þessu tilfelli en hér má setja hvaða tölur inn sem er til að komast að því hvað c þyrfti að vera löng hlið til þess að þríhyrningurinn sé rétthyrndur.

Æfing 2.7

Þetta eru tillögur, en alls ekki eina leiðin til að fá þessar villur.

```
# nafnavilla
breyta = breyta8 # breyta8 hefur ekki verið skilgreind

# Málskipunarvilla
3breyta = 3 # tölustafur má ekki vera fremst í breytuheiti

# Inndráttarvilla
```

```
8     breyta3 = 3 # of mikið bil fremst í línunni
```

Kafli 3

Æfing 3.1

Hér þurfum við að átta okkur á því hvernig strengir eru skilgreindir, annað hvort með tvöföldum eða einföldum gæsalöppum. Svo og að breytur eru skilgreindar með því að setja nafnið á breytunni vinstra megin við jafnaðarmerki og gildið, í þessu tilfelli streng, hægra megin við jafnaðarmerki.

```
1 strengur = "hér er textinn sem geymist í breytunni strengur" #tvöfaldar gæsalappir báðu
      megin
2 strengur = 'hér er verið að nota einfaldar gæsalappir'
3 strengur = '' # þetta er tómur strengur
```

Æfing 3.2

Fremsti stafurinn er í núllta stæði.

```
1 strengur = "texti byrjar á t"
2 stafur = strengur[0]
```

Æfing 3.3

```
1 strengur = "halló góðan daginn í dag"
2 len(strengur)
```

Æfing 3.4

```
1 strengur = "kex!"
2 print(strengur[3])
```

Æfing 3.5

```
1 len("kex!")
2 # eða
3 print(len("kex!"))
```

Æfing 3.6

```
1 s = "allra handa"
2 print(s[len(s)//2])
```

Æfing 3.7

Athugaðu að þegar er verið að lengja breytuna þá er hún endurskilgreind. Þetta er gert því beðið er um að lengja þriðju breytuna og því gefið í skyn að breytan eigi að fá uppfært gidli. Ef þú hinsvegar lagðir annan skilning í verkefnið og bjóst til fjórðu breytu þá er það líka í lagi að svo stöddu, við erum ekki að besta fyrir minnisnotkun.

```
s1 = "allra handa"
s2 = " bil fremst"
s3 = s1 + s2
s3 = s3*4
```

Æfing 3.8

Hér látum n1 og n2 vera á einhvern hátt stangast á við íslenskar ritunarreglur, svo skeytum við þeim saman eftir að hafa beitt á þá aðferðinni .capitalize() sem lagar þá til eins og fyrirmælin segja til um.

```
n1 = "valBorg"
n2 = "sturluDóttir"
rett_nafn = n1.capitalize() + n2.capitalize()
```

Æfing 3.9

Við höfum ansi frjálsar hendur í þessu verkefni, en athugið að við höfum enn ekki kynnst því hvernig á að gera eitthvað með skilyrðum, lykkjum eða handahófskennt. Svo við verðum að beita einungis þeim aðferðum sem við höfum séð hingað til. Einnig verðum við að passa að ekki sé verið að vísa í sætisnúmer sem eru ekki til staðar í strengjunum. Takið eftir að þessi lausn er einungis hugmynd, í skrefi 1 má skipta um strengi, í skrefi 3 má klippa í sundur á mismunandi hátt, beita fleiri aðferðum og lengja sjaldnar eða oftar eða með öðrum tölum. Aðalatriðið í skrefi þrjú er að sama hvaða strengir eru settir inn í skrefi 1 þá er aldrei vísað út fyrir strenginn og allur lykilstrengurinn er notaður, hvert einasta stæði.

Prófaðu að setja inn mismunandi strengi og prenta út lykilinn í lokin.

Ef þú leystir verkefnið með ákveðnum sætisvísum þar sem þú vissir hversu mörg stafabil voru í strengnum þínum er kóðinn einungis nothæfur á strengi af nákvæmlega sömu lengd. Gott er að skrifa kóða sem leysir verkefni almennt en ekki bara það tiltekna verkefni sem er fyrir framan þig.

```
#skref 1
lykill = "þetta á að verða gott lykilorð"
takn = "0123456789!$%&()acptrewq"

#skref 2
lengd_l = len(lykill)
lengd_t = len(takn)

#skref 3
lykill = lykill[0:lengd_l//4].upper()+takn[0:lengd_t//8]*2 +
    lykill[lengd_l//4:lengd_l//2] + takn[lengd_t//7:lengd_t//4] + lykill[lengd_l//2:-1] +
    takn[0:lengd_t//3] + lykill[-1]*2+takn[-1]*3

#skref 4
lykill = lykill[::-1]
```

Kafli 4

Æfing 4.1

Við búum til lista með hornklofum. Breytan má ekki heita list því það er frátekið lykilorð í Python.

```
listi = []
listi_med_stokum = ["hér settum við eitthvað inn"]
```

Æfing 4.2

Við höfum kynnst núna fjórum týpum, heiltölum (int), fleytitölum (float), strengjum (str) og listum (list). Við lausn þessa verkefnis skiptir ekki öllu máli hvort að stökin voru fyrst sett í breytur og listinn skilgreindur með þeim eða hvort að stökin voru sett beint inn í skilgreininguna. Einnig skiptir ekki öllu máli hvort að stökin hafi verið sett inn um leið og listinn var skilgreindur eða notuð var append() eða instert() aðferðin til að setja í listann.

Aðalatriðið er að listinn varð til og að hann inniheldur þau gögn sem hann átti að innihalda (í hvaða röð sem er).

Til að búa til lista sem inniheldur lista er það gert eins og hvert annað stak, hann er gerður með hornklofum og inniheldur 0 eða fleiri stök.

Svo til að leysa seinna verkefnið þá þarf að vita hvar strengurinn var settur, í þessu tilfelli stæði 2. Svo þarf að endurskilgreina það hvað stæði 2 í listanum inniheldur.

```
listi = [1, 1.2, "strengur", []]

listi[2] = "nýr strengur í stað þess sem var"
```

Æfing 4.3

Vegna þess að við sjáum að heimavist er í staki 2 gætum við þess vegna skrifað nem1[2], en nú skulum við hugsa fram í tímann. Hvað ef listauppbyggingunni yrði breytt og kennitölu yrði bætt við fremst í listann? Þá myndi kóðinn okkar ekki uppfylla skilyrðin „sækja heimavist og netfang" heldur myndi kóðinn okkar sækja síma og heimavist. Með því að reiða okkur eingöngu á það að header listinn sé réttur og að gögnin í nemendalistunum séu í samræmi við hann þá fáum við alltaf rétt gögn sem eru vistuð í dálknum „heimavist" með header.index('heimavist').

Einnig hefðum við getað, í þessu tilfelli bara horft á nem1 og nem2 og skrifað strengina upp. En við viljum horfa fram í tímann þegar við verðum að vinna með lista sem innihalda kannski 50 eða 500 gagnapunkta og að við ætlum ekki að fletta upp heimavistum tveggja nemenda heldur 800. Þá viljum við vera búin að temja okkur að láta tölvuna vinna sem mest fyrir okkur.

```
header = ["nemandi", "sími", "heimavist", "netfang", "lykilorð", "áfangar"]
nem1 = ["Valborg", "9999999", "vestur", "valborg@flensborg.is", "best_practice",
    "FORR2**"]
nem2 = ["Sturludóttir", '00000000', 'austur', "valborg@example.com", "1234", "FORR1**"]

uppl_nem1 = nem1[header.index('heimavist')] + nem1[header.index('netfang')]
uppl_nem2 = nem2[header.index('heimavist')] + nem2[header.index('netfang')]
```

Æfing 4.4

Hér þurfum við að athuga að listinn inniheldur marga lista og það má keðja notkun hornklofa til að sækja gögn.

```
valli = nested_list[3][0][0]
print(type(valli))
zero = nested_list[0][0][0][0]
print(type(zero), zero)
```

Æfing 4.5

Hér er verið að sækja öll þau gögn í listanum „listed" sem ná frá stæði 2 að stæði 2. Það er byrjað er að lesa fyrir framan stæði 2 en einnig er hætt á þeim stað svo stæði 2 er aldrei lesið. Leshausinn hættir á sama stað og hann byrjar, færist ekkert og ekkert tákn er lesið.

Þar sem við erum að vinna með lista skilast gögn af týpunni listi, athugaðu þetta með streng og

þú færð tóman streng.

Æfing 4.6

Hér væri einnig gott að beita print skipunum til að sjá hvað er að gerast í hverju skrefi.

```
tengilidir = ["Halldóra", "Freyja", "Pétur", "Haukur"]
tengilidir.sort()
tengilidir.append("Agnes")
tengilidir.sort()
tengilidir.pop(2))
```

Æfing 4.7

Hér þurfum við að athuga að í þessari lausn er gert ráð fyrir að við vitum hvað Askja endar eftir að hafa verið bætt við og eftir að listanum er raðað. Einnig er hægt að gera þetta með því að nota `index()` til þess að komast að því hvar innri listinn um Öskju er.

```
dyr= [["Bolli", 3, "Hamstur"], ["Snælda", 5, "köttur"]]
dyr.append(["Askja", 13, "hundur"])
dyr[2].append("eltir lauf")
dyr.sort()
dyr[0].pop()
dyr.pop(0)
```

Kafli 6

Æfing 6.1

Þetta er mjög svipað kóðabút 6.5, síðustu lykkjunni.

```
for x in range(100)
   print(x)
```

Æfing 6.2

Athugum hér að til þess að geta haldið utan um summu þurfum við að skilgreina breytu áður en við gerum lykkjuna, sömuleiðis listann af tölunum. Talnalistinn getur verið með hvaða tölum sem er, heiltölum eða fleytitölum. Summuna verðum við að skilgreina og þar sem við höfum ekki séð neina tölu þá skilgreinum við hana sem núll í upphafi. Svo lykkjum við í gegnum listann okkar og notum breytuna sem hleypur í gegnum listann til að bæta við summuna. Þegar lykkjan er búin þá erum við ekki lengur í sama inndrætti og þá ætlum við að prenta út summuna okkar, þá prentast hún bara einu sinni.

```
listi = [1,2,3,4,5,6,7,8,9,10]
summa = 0
for tala in listi:
   summa = summa + tala
print(summa)
```

Æfing 6.3

Svo við gerum það sama og áður, við þurfum summu breytu áður en við förum inn í lykkjuna en við notum `range()` fallið.

```
summa = 0
for tala in range(1000):
```

```
3     summa = summa + tala
4  print(summa)
```

Æfing 6.4

Til þess að geta skoðað þversummu þurfum við að skoða hvern tölustaf fyrir sig og þá þurfum við að kasta í streng og skoða hvert stak í honum. Þetta veldur því að við þurfum lykkju innan í lykkju. Einnig þurfum við að kasta á milli taga og nota samanburð. Hér er vísbending.

```
1  for tala in range(100):
2     talna_strengur = ?
3     ?
4     for stak in talna_strengur:
5        ? += ?
6     if(? > 6):
7        ?
```

Áður en þið skoðið lausnina skulið þið gera heiðarlega tilraun til að fylla inn fyrir spurningarmerkin. Hér kemur langur texti um hvað þarf að gerast svo að þið sjáið ekki lausnina alveg strax.

Það sem þarf að setja inn er að gera `tala` að streng svo hægt sé að rúlla í gegnum hana í innri lykkjunni.

Áður en innri lykkjan er keyrð þarf að skilgreina breytu sem á að halda utan um summuna, ástæðan fyrir því að það gerist í ytri lykkjunni en ekki utan hennar eins og áður er vegna þess að við viljum fá nýja summu sem er skilgreind sem 0 fyrir hverja einustu tölu í ytri lykkjunni.

Þá erum við komin með breytu sem heldur utan um summu og getum ferðast í gegnum vísa talna strengs, það sem við þurfum þá að gera er kasta hverju staki fyrir sig í tölu svo að við getum notað það í útreikningi. Þá viljum við leggja það saman við summuna og uppfæra summuna með þessari samlagningu.

Þegar við erum búin með innri lykkjuna erum við komin með þversummu fyrir einhverja eina tölu úr talnalistanum sem ytri lykkjan er að skoða. Þá viljum við spyrja er þessi þversumma stærri en 6?

Ef svo er þá viljum við prenta út töluna sem hafði þessa þversummu.

```
1  for tala in range(100):
2     talna_strengur = str(tala)
3     tversumma = 0
4     for stak in talna_strengur:
5        tversumma += int(stak)
6     if(tversumma > 6):
7        print(tala)
```

Æfing 6.5

Athugið hér að nafnalistinn er ólíkur ykkar, þessi listi var gerður til að sýna að ANNA og Albert innihalda ekki táknið sem beðið var um og komast því ekki í gegn. Ef óskað væri eftir að hafa það tákn líka væri hægt að rifja upp `.lower()` úr kafla 3 eða hvernig eigi að nota rökvirkja í segð[4].

```
1  listi = ["Halldóra", "ANNA", "Sigurður", "Albert", "Jóna", "Valborg", "Unnur", "Pétur",
       "Unnar"]
2  for stak in listi:
3     if "a" in stak:
4        print(stak)
```

[4] þá í stað if "a"in stak væri if "a"in stak.lower(), eða með rökvirkja if "a"in stak or "A"in stak

Æfing 6.6

Hér er beðið um að láta skilyrði lykkjunnar vera `True` og nota lykilorðið `break`

```
while(True):
   break
```

Æfing 6.7

Hér þarf að athuga inndráttinn á öllum aðgerðunum og að upphaflega lykkjuskilyrðið sé rétt skilgreint.

```
tala = 3
while(tala < 20):
   print(tala)
   tala += 1
```

Æfing 6.8

Hér eru báðar útgáfur sýndar af þeim tillögum sem voru nefndar, en eins og með svo margt annað í forritun þá tókst ykkur mögulega að gera þetta öðruvísi. Aðalmarkmiðið var að geta notað lykilorðið `in` í skilyrði lykkjunnar.

```
listi = ["nammi", "popp", "ávextir", "popp", "grænmeti", "popp", "hunang", "popp",
    "brauð"]
while("popp" in listi):
   visir = listi.index("popp")
   listi.pop(visir)
   print(listi)

#eða
while("popp" in listi):
   listi.remove("popp")
   print(listi)
```

Kafli 7

Æfing 7.1

Hér vitum við að það eru þrjú stök í ndinni, svo að aftasta stakið hefur sætisnúmerið 2.

```
nd = (1,2,3)
tala = nd[2]
```

Æfing 7.2

Rifjum upp að til að ítra í gegnum hlut af gefinni stærð er best að nota for lykkju.

```
talna_nd = (1,34,432,324,999,1,2,3,1,3,55,664,10000)
for tala in talna_nd:
   if tala > 100:
      print(tala)
```

Æfing 7.3

Við erum beðin um að búa til nd eð tveimur stökum, svo erum við beðin um að geyma útkomu sem þýðir að við þurfum breytu. Sú breyta á að innihalda svarið við hvort að fyrri talan sé stærri en sú

seinni svo það er sanngildi. Að því loknu erum við beðin um að nota samskeytingu en útkoman er ekki nd svo við þurfum að setja hana í nd til að geta beytt samskeytingu.

```
nd = (1,2)
tala1, tala2 = nd
svar = tala1 > tala2
ny_nd = nd + (svar,)
```

Æfing 7.4

Rifjum upp `input()` fallið úr kafla 5, einnig `range()` fallið úr kafla 6.

```
nd = ()
for i in range(4):
    svar = input("hver er uppáhalds liturinn þinn?")
    nd = nd + (svar,)
print(nd)
```

Kafli 8

Æfing 8.1

Hér er aðalatriðið að geta bætt við eftir skilgreiningu en ekki bara að geta búið til orðabók sem er skilgreind frá upphafi með einhverju lykla og gildis pari.

```
bok = {}
bok['nýr lykill'] = 'Halló Heimur!'
```

Æfing 8.2

Hér þarf að muna eftir range fallinu til að auðvelda okkur vinnuna annars er þetta sama verkefni og æfing 8.1.

```
bok = {}
for i in range(5):
    bok[i] = i**2)
```

Æfing 8.3

Það eru til allavega tvær leiðir til að fjarlægja stak úr orðabók og fyrst við erum beðin um að fjarlægja tvö skulum við nota báðar aðferðirnar.

```
bok = {1: "þetta skal tekið", 2: "þetta skal vera", 3: "hiklaust fjarlægt"}
bok.pop(1)
bok.popitem()
```

Æfing 8.4

Hér þurfum við að athuga að áður en við förum að skoða hvað sé minnsta gildið þá þurfum að skilgreina breytu sem við viljum vera að bera saman við, og þarf sem við ætlum að prenta út 0 ef við finnum enga tölu lægri en það þá skilgreinum við breytuna sem við notum til samanburðar sem 0.

```
bok = {'lykill1': [3,4,2,4,2], 2: [-90, 2,3,1], "þrír": [-3,1000]}
minnsta_gildi = 0
for gildi in bok.values():
    if min(gildi) < minnsta_gildi:
```

```
5       minnsta_gildi = min(gildi)
6 print(minnsta_gildi)
```

Æfing 8.5

Það sem við þurfum að gera hér er að athuga sérstaklega hvort að við séum í raun að vinna með streng, við munum að lykilorðið fyrir streng er **str** og við athugum hvort að týpan af því sem við erum með í höndunum (hvert stak fyrir sig) sé strengur því að við megum ekki bæta strengnum x aftan við hvað sem er. Einnig þurfum við að setja það inn aftur í staðinn, þessi leið er ekki fullkomin til þess en skoðið þennan kóða og áttið ykkur á hvað er um að vera áður en þið skoðið hina útfærsluna.

```
1 bok = {0: ["hér", "eru nokkrar týpur", 1, 2, 3, True], 1: ["líka hér", "nokkur tög", {1:
      ['Þetta telst ekki með', 'því þetta er innan orðabókar']}]}
2 for lykill, gildi in bok.items():
3    for stak in gildi:
4       if type(stak) is str:
5          stadur = gildi.index(stak)
6          gildi[stadur] = stak + 'x'
```

Athugið að eitt auka hérx, ef því er bætt við fyrir ofan sést hve illa kóðinn grípur sum tifelli. Hér er ekki verið að nota sætisnúmer með uppflettingu á stakinu heldur með númeri keyrslunnar.

```
1 bok = {0: ["hér", "hérx" ,"eru nokkrar týpur", 1, 2, 3, True], 1: ["líka hér", "nokkur
      tög", {1: ['Þetta telst ekki með', 'því þetta er innan orðabókar']}]}
2 for lykill, gildi in bok.items():
3    for i in range(len(gildi)):
4       if type(gildi[i]) == str:
5          gildi[i] = gildi[i] + 'x'
```

Æfing 8.6

Þar sem þessi æfing er krefjandi eru hér eingöngu vísbendingar til að leysa hana.

Athugum hér að við þurfum orðabók þar sem lyklar eru td. ęr sólin bláǫg gildi lykilsins væri "rangt", þegar hún er tilbúin þá getum við rúllað í gegnum hana og prentað út lyklana. Á eftir því að hafa prentað út lykil viljum við setja input skipun fyrir notandann, svarið ætlum við ekki að geyma neitt frekar en bara til að athuga hvort að það sé það sama og svarið (gildið á lyklinum sem við vorum að skoða). Þá förum við í reiknikúnstirnar að hækka ef rétt.

Þegar allar spurningarnar eru þá búnar er loka einkunn komin, við getum borið einkunnina saman við fjölda lykla eða lengdina á bókinni ef það er það sama þá prentum við út auka textann um að þetta fór á besta veg með hæstu einkunn.

Æfing 8.7

Þar sem þessi æfing er krefjandi eru hér eingöngu vísbendingar til að leysa hana.

Fyrir það fyrsta er að orðabókin með sveitarfélögunum sé stöðluð, það er rvk fyrir Reykjavík og þá hfj fyrir Hafnarfjörð en ekki á víxl. Því næst er að athuga að spyrja notandann með input() fallinu og vegna þess að beðið er um mismunandi sveitarfélög þá verður bæði að athuga hvort að sveitarfélagið sem notandinn gaf upp sé til og það þarf einnig að athuga hvort að það sé það sama og viðkomandi gaf upp síðast. Til þess þarf þá að nota while lykkju. Þegar tvö mismunandi sveitarfélög eru komin er þá hægðarleikur að leggja saman gildin og prenta út nöfnin á þeim ásamt samanlögðum fjölda íbúa.

Kafli 9

Æfing 9.1

Þetta eru klækir, þar sem það veldur tvíræðni að nota tóma slaufusviga til að skilgreina bæði tóma orðabók og tómt mengi var ákveðið að tómir slaufusvigar þýða alltaf orðabók. En tómamengið er gífurlega mikilvægt og því nauðsynlegt að geta búið það til og átt það svo það er að sjálfsögðu hægt. Með þessu móti er hægt að búa til tómt mengi sem hægt er svo að vinna með.

```
mengi = set()

```

Æfing 9.2

```
listi = ["hér er strengur1", "ólíkur öllum öðrum", "eins", "eins", "Eins", "ekki eins"]
mengi = set(listi)
```

Æfing 9.3

Hér erum við að rifja upp ýmisslegt gagnlegt sem er ekki gott að gleyma. Það fyrsta er hvernig á að ítra með for og hitt er að nota range. Það sem er nýtt hinsvegar er að við notum range til að rúlla upp í 4 því að við viljum að 3 sé með, en ekki fjórir því að hæsti vísirinn í 5 staka lista er 4.

```
talnalisti = [1,2,3,4,5]
vinnslulisti = []
for i in range(4):
tala = talnalisti[i] + talnalisti[i+1]
vinnslulisti.append(tala)
talnamengi = set(talnalisti)
talnamengi.update(vinnslulisti)
print(len(talnamengi) == 10)
```

Æfing 9.4

```
mengi1 = {1,2,3,4,5,6,7,7,8}
mengi2 = {2,4,7,9,7,6,4,3,4,44,4,44,4,4}
mengi3 = mengi1 & mengi2
```

Kafli 10

Æfing 10.1

Við þurfum ekkert að hugsa um return á þessu stigi málsins bara að muna eftir hvernig á að setja upp fall og nota það.

```
def fall1(a, b):
   print(b, a)
fall1(1,"halló")
fall1("prufa", "hvað?")
fall1(2, 4)
```

Æfing 10.2

Hér erum við komin í smá vandræði, það er að við viljum að notandinn setji inn strengi í fallið okkar en við gerum ekkert til þess að fullvissa okkur um að svo sé (sjáum það í kafla 14) svo við

ætlum að sjá hvað gerist þegar við notum fallið eins og okkur sýnist þó svo að við vitum að fallið taki bara við strengjum.

Það sem þarf að gerast er einföld skilyrðissetning og útprentun. Einföld á þann hátt að hún er ekki hreiðruð og að við þurfum bara fáa samanburðarvirkja. En að vísu má sleppa samanburðarvirkjunum með því að nota innbyggt fall sem heitir *max()*. Prófið ykkur áfram með að breyta þessari lausn og setja inn þrjá eins strengi.

Lausnin með samanburðinum er ófullkomin því að aldrei er leyfilegt að tveir strengir sem eru aftar í stafrófinu séu jafnir (hvað þá allir þrír).

Reynið að leysa þetta án þess að nota max() í else.

```
def fall2(a = "Anna", b = "Bára", c = "Carl"):
   if(a > b and a > c):
      print(a)
   elif(b > a and b > c):
      print(b)
   elif(c > a and c > b):
      print(c)
   else:
      print(max(a,b,c))
```

Æfing 10.3

Nú getum við aftur beitt max fallinu en þá til þess að henda þeirri tölu út svo að við getum beitt max aftur. Athugum að muna eftir því að skila.

```
def fall3(listi):
   gildi = max(listi)
   listi.remove(gildi)
   haesta = max(listi)
   return haesta

fall3([1,2,3,4,55,66])
```

Æfing 10.4

Nú þurfum við lykkju fyrir listann, samanburð við gildið og breytu sem heldur utan um talninguna.

```
def fall4(gildi, listi):
   talning = 0
   for stak in listi:
      if stak == gildi:
         talning += 1
   return talning

fall4(4, [1,2,3,4,5,4,2,3,4,5,4444])
fall4("halló", ["halló", 123, {}, {"halló": "halló"}, ["telst ekki með hér", "halló"]])
```

Æfing 10.5

Hér erum við í fyrsta sinn að vinna með skilagildi af týpunni Boolean, spennandi.

```
def fall5(str1, str2):
   if(str1[0].lower() == str2[0].lower()):
      return True
   elif(str1[0].lower() in "aáeéiíoóuúæyýö" and str2[0].lower() in "aáeéiíoóuúæyýö" ):
      return True
   return False

fall5("breki", "Bolli")
```

```
fall5("ester", "Ösp")
fall5("kanína", "Portúgal")
```

Hér sjáum við að return False kemur fyrir í sama inndrætti og skilyrðissetningin, má það? Það er vegna þess að ef við fórum aldrei inn í skilyrðissetninguna, þá gátum við aldrei skilað True og því hljótum við að eiga skila False. Sama niðurstaða fengist með því að bæta við `else return False`. Takið svo eftir .lower() til að staðla inntakið svo að við fáum örugglega True fyrir Bolli og breki, og ester og Ösp.

Æfing 10.6

```
def fall6(tala1, tala2):
   if tala1%2 == 0 and tala2%2 == 0:
      return min(tala1, tala2)
   return max(tala1, tala2)

fall6(2, 4)
fall6(1, 3)
fall6(8, 1)
fall6(9, 3)
```

Æfing 10.7

```
def fall6(listi):
   talning = 0
   for stak in listi:
      if stak == 0 and talning == 0:
         talning = 1
      elif stak == 0 and talning == 1:
         talning = 2
      elif stak == 7 and talning == 2:
         return True
   return False
```

Hér sjáum við að return False kemur fyrir í sama inndrætti for, það er vegna þess að þegar for lykkjan er búin þá höfum við ekki náð að finna 7 á eftir tveimur núllum og því skilum við false. Sú setning má ekki vera inni í for lykkjunni því að þá myndum við ekki klára að skoða öll stökin áður en við myndum úrskurða um hvort mynstrið væri til staðar eða ekki.

Æfing 10.8

Rifjum upp svarið við 7.4

```
nd = ()
for i in range(4):
   svar = input("hver er uppáhalds liturinn þinn?")
   nd = nd + (svar,)
print(nd)
```

Við vildum nota þetta innan falls og við vildum ekki bara prenta útkomuna heldur skila henni, en við vildum jafnframt vera aðeins að athuga inntakið með skilyrðissetningu.

Prófið fallið með bæði tölu og streng eða öðru tagi.

```
def fall8(x):
   nd = ()
   if type(x) != int:
```

```
        return nd

    for i in range(x):
        svar = input("hver er uppáhalds liturinn þinn?")
        nd = nd + (svar,)
    return nd
```

Kafli 11

Æfing 11.1

input skipunin er í rauninni bara þarna í staðinn fyrir time.sleep(), núna er það notandinn sem ákveður hvenær næsta lína af kóða er keyrð. Það sem við þurfum að fatta er að reikna rétt út og finna tímann.

```
adan = time.time()
input("ýttu eftir 10 sek")
nuna = time.time()
print(nuna-adan)
```

Æfing 11.2

Hér þurfum við aðallega að átta okkur á því hvaða föll við viljum nota úr random pakkanum, úr listanum viljum við nota choice en til að finna eitthvað tákn viljum nota randint. Athugum hér að þegar við notum randint þá gefum við upp bil af tölum eins og 1,10 þá eru bæði 1 og 10 með, svo þegar við notum það til að búa til sætisvísi af handahófi verðum við að passa að fara ekki út fyrir löglega sætisvísa.

```
rugl_ord = random.choice(listi)
rugl_takn = rugl_ord[random.randint(0,len(rugl_ord)-1)]
print(rugl_ord, rugl_takn)
```

Kafli 12

Æfing 12.1

```
%%writefile doc1.txt
Þessi byrjar ekki á sérhljóða
En þessi gerir það
```

```
with open('doc1.txt', mode = 'r') as doc1:
linur = doc1.readlines()
for lina in linur:
if(lina[0].lower() in 'aáeéiíoóuúyýæö'):
print(lina)
```

Æfing 12.2

```
%%writefile doc2.txt
Krummi svaf í klettagjá,
```

```
kaldri vetrarnóttu á,
::verður margt að meini::
Fyrr en dagur fagur rann,
freðið nefið dregur hann
::undan stórum steini.::

Allt er frosið úti gor,
ekkert fæst við ströndu mor
::svengd er metti mína.::
Ef að húsum heim ég fer
heimafrakkur bannar mér
::seppi' úr sorp að tína.::

Öll er þakin ísi jörð,
ekki séð á holtabörð
::fleygir fuglar geta.::
En þó leiti út um mó,
auða hvergi lítur tó;
::hvað á hrafn að éta.::

Á sér krummi ýfði stél,
einnig brýndi gogginn vel,
::flaug úr fjallagjótum::
Lítur yfir byggð og bú
á bænum fyrr en vakna hjú,
::veifar vængjum skjótum.::

Sálaður á síðu lá
sauður feitur garði hjá,
::fyrrum frár á velli.::
Krunk, krunk, nafnar, komið hér,
krunk, krunk, því oss búin er
::krás á köldu svelli.::
```

```
import random
with open('doc2.txt', mode = 'r') as doc2:
   linur = doc2.readlines()
   print(random.choice(linur))
```

Kafli 13

Æfing 13.1

Það sem þarf að athuga hér er að `Lest` á tilviksbreytuna `nuverandi_stod` sem við viljum uppfæra en það má ekki gerast fyrr en við erum búin að reikna hversu langt er þangað. Til þess að finna hvað er langt á milli getum við notað einhverja af þeim þremur aðferðum sem búið var að útfæra. Einnig má ekki gleymast að uppfæra kílómetrastöðuna. EFtirfarandi er kóðinn fyrir útfærsluna á aðferðinni en prófanir eru eftirlátnar lesanda.

```
def ny_nuverandi_stod(self, stod):
   km = self.fara_til_stod(stod)
   self.farnir_km += km
   self.nuverandi_stod = stod

   return self.farnir_km
```

Æfing 13.2

Hér eru margar leiðir til þess að leysa verkefnið, hvað er það sem gerir tvo leiki jafna? Hér er það útfært þannig að tveir leikir eru jafnir ef nöfnin eru þau sömu, sömuleiðis er útfærslan á < er þannig að einn leikur er strangt minni en annar ef þar er minni peningur.

Þetta gæti líka verið gert með rökvirkjum, til þess að tveir leikir séu jafnir þurfa allar tilviksbreytur að vera eins, eða einhvern veginn allt öðruvísi.

Prófanir eru eftirlátnar lesanda.

```
def __eq__(self, other):
   return self.nafn == other.nafn

def __lt__(self,other):
   return self.peningur < other.peningur
```

Kafli 14

Æfing 14.1

Nú þurfum við að vera vakandi fyrir því hvað það er sem notandinn getur sett inn sem tölu, hvað við viljum prenta út hverju sinni og hvernig við prófum að við höfum náð öllum tilvikum.

```
def samanburdur_vid_null(vidfang):
   try:
      int(vidfang)
   except:
      try:
         float(vidfang)
      except:
         print("ekki sambærilegt")
      else:
         print(0 < float(vidfang))
   else:
      if(type(vidfang) == float):
         print(0 < vidfang)
      else:
         print(0 < int(vidfang))

samanburdur_vid_null("0.1")
samanburdur_vid_null(0.1)
samanburdur_vid_null("3")
samanburdur_vid_null([1])
samanburdur_vid_null("fimm")
samanburdur_vid_null({"a":1})
samanburdur_vid_null(-23)

```

Æfing 14.2

Hér eru kóðastubbar sem valda eftirfarandi villum, fleiri leiðir eru til að fá villurnar. Svarið í a. lið er að því gefnu að breytan `bolti` hafi aldrei verið skilgreind áður. Hér er það í höndum lesenda að skrifa svo kóðann sem grípur villurnar.

a) `bolti`

b) `"halló".sort()`

c) `["a"]`

d) `int("vidfang")`

Kafli 15

Æfing 15.1

```
def hnit(hni11, hnit2):
   # gert er ráð fyrir að hnit sé annað hvort nd eða listi
   if(hnit1 == hnit2):
      return 0
   if(hnit1[0] == hnit2[0] or hnit1[1] == hnit2[1]):
      return 1
   else:
      return 2
```

Æfing 15.3

Hér er mikið atriði að prófa allskonar mismunandi lista til að vera viss um að þið séuð með kórréttan kóða.

```
def minnsta_fjarl(listi):
   styst = abs(listi[1]-listi[0])
   for i in range(len(listi)-1):
      if(styst > abs(listi[i+1]-listi[i])):
         styst = abs(listi[i+1]-listi[i])
   return styst

def mest_fjarl(listi):
   mest = abs(listi[1]-listi[0])
   for i in range(len(listi)-1):
      if(mest < abs(listi[i+1]-listi[i])):
         mest = abs(listi[i+1]-listi[i])
   return mest
```

Æfing 15.3

Hér gætum við að notað input fallið og randint

```
import random as rnd
def ispinnar():
   def summa(fjoldi):
      if fjoldi <= 1:
         return 10
      else:
         return rnd.randint(0,10) + summa(fjoldi - 1)

   fjoldi = input('Hvað eru mörg börn í röðinni? ')
   try:
      int(fjoldi)
      return summa(int(fjoldi))
   except:
      return 10

```