# Inngangur að forritun í Python

Forritun fyrir byrjendur

Valborg Sturludóttir

*First printing, Sept 2020*

# Efnisyfirlit

# Fyrsti hluti

laga gæsalappir
vspace staðla

# 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 framhaldssnámi 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 íterefni 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 að 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 spurningin „hvað kanntu mörg forritunarmál?" út í hött. 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 málýskur innan tungumála. Nær væri að spyrja hvort viðkomandi hafi meiri áhuga á framenda eða bakenda forritun, hvað 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 eiga klósett. Innan tölvunar 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 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).

Uppbyggining er þannig að fyrri hlutinn snýr að því að kynna lesandann fyrir grunn virkni 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 eru kynntir til sögunnar klasar og aðferðir sem lesandinn útfærir upp á eigin spýtur. 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 tölvunarfræða 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. Í gegnum bókin fylgjum við svo þremur verkefnum sem verða þyngri og flóknari eftir því sem fleiri hugtök eru kynnt til sögunnar.

## 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 er 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 mikið notað, algengt mál svo það er praktískt að hafa undirstöðu skilning á því.
6. Nefnt í höfuð á Monty Pyhton grínhópsins.

## 1.3 Uppsetning

.

Víðsvegar um bókina, aðallega í upphafi , má finna númeraða kóðabúta sem eru ekki teknir úr vinnubók og eru því ekki eins litakóðaðir og þeir sem eru teknir úr vinnubókum. 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.

## 1.4 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) til að 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.

Þessari bók fylgja einnig nokkrar vinnubækur úr Jupyter sem lesandinn getur nýtt sér. Hér á mynd sést hvernig tóm Jupyter vinnubók lítur út. Virkninni er skipt upp í sellur og keyrslurröð sellanna skiptir máli, við sjáum seinna mikilvægi þess að geta skipt upp kóða svona og hvers vegna

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

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.

Hér væri réttast að skoða Vinnubók 1 sem fylgdi þessari bók.

hér væri gott að
fyrir dæmi úr vi
sem á að fylgja

En við keyrslu á kóða þarf einnig að hafa í huga að tölvan gerir nákvæmlega það sem við segjum henni að gera og ekker 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 þér. Þær gera nákvæmlega það sem þú baðst um. Nákvæmlega eins og þú baðst um það.

segja miklu mei
um hvernig á að
vinnubækur yfirh

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 nokkur og ef véliin kann ekki nú þegar skil á:

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

Svo þó svo að þér hafi þótt þessi útskýring á samlokugerð 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 þú hefur vanist dagsdaglega. Þar ertu ekki að gefa tölvunni þínar eigin skipanir heldur ertu að beita skipunum sem aðrir forritara hafa samið og sett upp í hugbúnaðinn sem þú ert að nota.

## 1.5 Málskipan

Málskipan (e. syntax) er hugtak sem þýðir hvernig á að skrifa kóða svo að hann þýðist í vélamál sem tölvan skilur. Málskipan eru 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 málskipunarvilla (e. syntax error). Python er frábrugðið öðrum forritunarmálum á þann hátt að málskipanin krefst þess að kóðinn sé settur upp á ákveðinn hátt. Líkja því má við að þurfa ekki að hafa greinamerki í huga þegar við ljúkum setningum heldur setjum við setningarnar okkar á réttan stað í samræðunum.

### 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 1.1: Réttur Python kóði

```
# Réttur Python kóði sem keyrist
4 + 8
5 + 6
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 1.2: Rangur Python kóði

```
# Illa skrifaður Python kóði sem keyrist ekki
4 + 8
   5 + 6
breyta = 9 * 2
```

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

```
// Java
int i = 7;
i + 5;

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

Kóðabútur 1.4: 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 frábrugnu 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 greinamerkin 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, orðabókum (einnig kallaðar hakkatöflur) og boolean gildum. Í seinni hlutanum bæstast svo við sett .

sett?

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

```
1 # Heiltölur, enginn aukastafur
2 42
3 100000
4 -139
5
6 # Fleytitölur, aukastafur/ir fyrir aftan punkt
7 4.0
8 3.1415926
9 -100.98
```

## 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 %. 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.

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*, athugasemdir 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 heild. Einnig eru þarna bil á milli talna fremst í línu og tákna, það er líka til að gera kóðan læsilegri, bilin mega bara ekki vera fremst í línunni enn sem komið er. Athugasemdir í kóða eru mjög mikilvægur hluti af skjölun kóða og ættu öll sem vilja tileinka sér forritun að venja sig á að skrifa athugasemdir. Í fyrstu erum við ekki að skrifa flókinn kóða svo athugasemdirnar segja okkur ekki mikið, en þegar kóðinn er ekki augljós eða lausn á verkefni ekki augljós er gott að skrifa athugasemdir. Flest allir kóðabútar eru skjalaðir með athugasemdum til að gera þá læsilegri því allur kóði í bókinni er skrifaður fyrir fólk til að skilja. Kóði sem þið komið til með að skrifa seinna meir 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  # Þegar eftirfarandi kóði er keyrður ætti útkonan að vera 10
3  6 + 4
4
5  # Frádráttur framkvæmdur með -
6  # Þegar eftirfarandi kóði er keyrður ætti útkonan að vera 10
7  14 - 4
8
9  # Margföldun framkvæmd með *
10 # Þegar eftirfarandi kóði er keyrður ætti útkonan að vera 10
11 10 * 2
12
13 # Deiling framkvæmd með /
14 # Athugið að þetta er fleytitöludeiling sem skilar nákvæmu svari
15 # Þegar eftirfarandi kóði er keyrður ætti útkonan að vera 10.0
16 60 / 6
17
18 # Heiltöludeiling framkvæmd með //
19 # Athugið að þessi deiling er frábrugðin þeirri sem þið kannist við
20 # Hér viljum við vita hversu oft, heil tala, ein tala gengur upp í aðra og okkur er sama
       um afganginn
21 # Þegar eftirfarandi kóði er keyrður ætti svarið að vera 10
22 177 // 17
```

```
23
24 # Veldishafning framkvæmd með **
25 # Hér er mikilvægt, eins og með deilinguna, að hafa í huga hvor talan kemur á undan.
26 # Fyrst kemur talan sem hefja á í veldi og svo kemur talan sem er veldisvísirinn
27 # Þegar eftirfarandi kóði er keyrður ætti svarið að vera 9
28 3 ** 2
29
30 # Leifareikningur framkvæmdur með % (e. modulus)
31 # Þetta er eitthvað alveg nýtt og framandi, en þó ekki óskiljanlegt
32 # Það sem þetta reiknar er hversu mikil leif eða afgangur er eftir þegar heiltöludeilingu
       er beitt.
33 # Þegar eftirfarandi kóði er keyrður ætti svarið að vera 7
34 177 % 17
```

Í öllum þessum dæmum var verið að vinna með heiltölur, þó var útkoman úr deilingunni (stundum kölluð fullkomin deiling) fleytitala. Hvað gerist ef þessir sömu útreikningar eru gerðir með fleytitölum? 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. En hvað gerist ef við breytum aðeins fyrri tölunni en ekki seinni tölunni? Þá ertu að nota ólíkar týpur og slíkt er vandmeðfarið, en í þessu tilviki er það í lagi þar sem Python gerir þá ráð fyrir að það sé í lagi að reikna allt með fleytitölum og framkvæmir reikninginn eins og þú hafir verið að beita fleytitölum í hvívetna og niðurstaðan verður þá að sjálfsögðu fleytitala.

## 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. Á hefbundinni reiknivél sem notuð er í stærðfræðitíma í framhaldsskóla er takki sem á stendur ANS. Það stendur fyrir answer og ef ýtt er á hann getur vélin geymt síðasta gildið sem hún gaf sem svar og unnið svo með það til að gefa næsta svar. Flottari vélar geta svo geymt nokkuð mörg svör en það er útfyrir gagnsemi þessarar samlíkingu. Þegar ýtt er á þennan takka er minnisvæði í reiknivélinni tekið frá og skrifað er í það gildi, sem er svo sótt þegar ANS er notað í útreikningi. Að sama skapi má láta Python úthluta minnissvæði í tölvunni fyrir þær breytur sem þið viljið geyma. Munurinn er sá að þið nefnið sjálf hvað minnisvæðið er merkt sem, eruð ekki bundin við að nota ANS og að þið eruð svo gott sem með óteljandi minnissvæði.

Að gefa minnissvæði merkingu og gildi er gert með *gildisveitingu*. Gildisveiting þýðir að nú er einhver ákveðinn merkimiði kominn með eitthvað til að geyma. Sjáum einfalt dæmi um þetta.

Kóðabútur 2.3: Breytur kynntar

```
1  # Hér er ég að fara að búa til breytu sem heitir val
2  val = 5
3
4  # Þegar ég keyri línuna fyrir ofan segi ég vélinni að hafa aðgengilegt minnisvæði sem ég
       get notað með því að skrifa orðið val, og settu í það svæði gildið 5.
5
6  # Svo ég er að veita breytunni val gildið 5, þess vegar er það kallað gildisveiting.
7
8  # Svo get ég notað breytuna mína
9  # þegar þetta er keyrt fæst svarið 10
10 val + 5
```

Ef þú prófar þig áfram við að búa til breytur gætir þú rekist á svolítið sem hefur ekki gerst áður í vinnubók, 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 þú sagðir

vélinni að geyma þetta, þú sagðir 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 hver 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.

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. Áherslulitunin í númeruðu kóðabútunum í þessari bók er marklaus því hún er mjög frumstæð. Dæmi um það sem fær áherslulitun eru frátekin lykilorð og tölustafir.
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 gera 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`.

Kóðabútur 2.4: Dæmi um gildisvetingar réttar og rangar

```
# Hér er ég að fara að búa til breytu sem heitir val
val = 5

# Hér er ég ekki að búa til breytu sem heitir val heldur er ég að segja að talan fimm er ekki lengur til sem heiltala heldur gæti hún vísað í hvað sem er sem er geymt í minnisvæði merktu val, ólöglegt.
5 = val

# Hér bý ég til breytu sem heitir heiltala sem fær gildið 0
heiltala = 0

# Hér yfirskrifa ég lykilorðið fyrir týpuna heiltala og læt það innihalda 0
int = 0
# þetta er harðbannað og ef þetta gerist er ekki nóg að þurrka þetta út og keyra aftur, nú þarf að endurræsa kjarna vinnubókarinnar.

Gott nafn = 1.0
# Þetta er ekki bara bannað vegna bilsins á milli orðanna ,,Gott'' og ,,nafn'' heldur er þ að líka ljótt því að það byrjar á stórum staf

3_litlar_mys = 3
# má ekki byrja á tölustaf eða tákni

```

```
utreiknud_laun_eftir_skatt = 0.65 * laun
# frábært, lýsandi og gott breytuheiti (hér er þó gert ráð fyrir að vélin þekki breytuna
    laun)
```

Nú þegar við höfum séð hvernig má skilgreina breytu 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 segi ég vélinni hvað það er sem ANS vísar á, svo segi ég vélinni að mig langar til þess að búa til nýja breytu sem á að byggja á því sem ANS inniheldur. Í þessum kóðabút er svo haldið áfram með þessa afleiddu breytu 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 breytan okkar. Ef þessi kóðabútur er keyrður þá kemur á *staðalúttak* [1] það gildi sem breytan **x** 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

```
# Hér framkvæmi ég einhvern útreikning sem ég geymi í breytunni ANS
ANS = 5**2 + (4+8.9)**2

# Segjum að þetta hafi verið endapunkturinn í löngu algebrudæmi og nú veit ég hvað y er,
    og get þá nýtt það til að finna x eins og verða vill svo oft í stærðfræði að x sé
    týnt. Gefum okkur að x = 3 * y og því fæst
x = 3 * ANS

# Nú ef við viljum reikna eitthvað út með x eigum við það til í minnissvæði merktu x með
    réttu gildi. Til dæmis með því að búa til breytu fyrir hálft x.
halft_x = x/2


# Nú langar okkur til að vera viss um að við séum við vitrænt svar svo við biðjum tölvuna
    um að segja okkur hvað er geymt í breytunni x.
print(x)
```

Við megum beita print skipuninni óspart og hvetur höfundur til þess að lesandi venji sig á að skoða úttakið sitt í hverju þrepi áður en leitað er hjálpar til annarra. Print er *fall*, við skoðum föll nánar í kafla 10 en þangað til munum við kynnast nokkrum innbyggðum föllum eins og print().

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* þar sem við uppfærum gildi í breytu með því að nota reiknivirkja eins og + eða - með gildisveitingu =. Þetta sést betur í kóðabút

Kóðabútur 2.6: "Reiknivirkjagildisveiting"

```
# ég ætla að telja nemendur inn í stofuna
# ég byrja með 0 nemendur
nem = 0

# svo sé ég fyrsta nemandann minn og endurskilgreini þar með breytuna nem
```

[1] 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?

```
nem = nem + 1

# þá uppfæri ég gildið sem nem breytan inniheldur og er hún núna það sem hún var (0) + 1, ef ég keyrði þessa línu aftur væri það orðið að (1) + 1 og svo koll af kolli eftir því sem ég keyri þessa línu oftar

# önnur leið til að skrifa þetta er með reiknivirkjagildisveitingu
nem += 1

# þá uppfæri ég gildið nem um það sem hún var + 1

#segjum að nemendur koma inn í stofuna í pörum þá væri formúlan svona:
nem += 2

# þá uppfærist gildið í nem um +2 í hvert sinn sem þessi kóðalína er keyrð

# Ef ég væri að reikna stofnstærð á bakteríum sem tvöfaldast á hverjum klukkutíma gæti ég gert það svona:
stofn_staerd = 30
stofn_staerd *= 2 # og keyrt svo þessa línu fyrir hvern klukkutíma

# Tökum annað dæmi, byrjum á að skoða hvernig megi geyma útreikninga í mismunandi breytum
thusund = 1000
fimm_hundrud = thusund/2
tvo_hundrud_og_fimmtiu = fimm_hundrud/2

# ef ef mér er sama um breytuna thusund og vil þess í stað bara halda utan um alla upphæðina mína í einni breytu og helminga hana tvisvar þá get ég gert þetta
allt = 1000
allt /= 2
allt /= 2

#nú er allt orðið að 250

# ég má líka nota aðrar breytur í uppfærslunni minni
# hér ætla ég að reikna út hver hækkun launa yrði milli ára ef ég fengi alltaf 2% launahækkun

laun = 100
verdbolga = 0.02
laun *= 1 + verdbolga # þetta má svo keyra endurtekið til að skoða fram í tímann
```

Hér 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. Í kóðabútnum að ofan sést að athugasemdir eru skrifaðar inni í línum líka, það er stundum gagnlegt að skrifa stuttar athugasemdir inni í kóða með þessum hætti en betra er þó að skjala hann skilmerkilega efst við viðeigandi kóðabút. Allar athugasemdir eru hunsaðar af vélinni og því hefur allt sem er fyrir aftan # merkið, hver sem það er, engin áhrif á útkomuna.

# 3. Strengir

Til þess að geta sýnt og notað texta þarf gagnatýpu til að halda utan um hann. Í flestum 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, munum við þurfa sér tákn til þess að segja vélinni gögn af hvaða týpu hún er að vinna með.

Táknin sem skilgreina strengi eru gæsalappir. Einfaldar eða tvöfaldar.

Kóðabútur 3.1: Strengir skilgreindir

```
1  # Fyrsti strengurinn okkar
2  "halló"
3
4  #strengur geymdur í breytu
5  textinn_minn = "halló ég má skrifa mörg orð inn í þessar gæsalappir"
6
7  einfaldar_gaesalappir = 'ég má líka skrifa innan einfaldra gæsalappa'
8
9  thetta_virkar_ekki = 'gæsalappirnar þurfa að passa saman"
10
11 # og ef ég vil skrifa mjög langan texta nota ég þrjár gæsalappir
12 langi_textinn_minn = ''' ég má skrifa eins langa setningu hér og ég vil því að þetta
       verður alltaf álitið sem ein lína, hins vegar ef ég nota öðruvísi gæsalappir og
       langar að gera kóðan læsilegan er hægt að brjóta hann upp án þess að nota þessar þ
       reföldugæsalappir, við sjáum það eftir smá'''
```

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

innubók inni í kóðabút

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.

Kóðabútur 3.2: Strengir og reikniaðgerðir

```
1
2  # Reikniaðgerðirnar sem við þekkjum eru +, -, *, /, //, **, og %
3  # Lesandinn er hvattur til þess að gera prófanir á þessu í vinnubók upp á eigin spýtur
4  # Með því að skilgreina streng og reyna að nota reikniaðgerð á hann með tölum eða öðrum
       strengjum
5
6
7  # Gerum ráð fyrir að þessar prófanir hafi átt sér stað og niðurstaðan sé sú að þær
       aðgerðir sem hægt er að framkvæma eru + og *
8  # En hvað gerist þegar við notum þær?
```

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 "hallóblessęn "bless"+ "halló"verður að "blesshalló".

Kóðabútur 3.3: Samskeyting strengja

```
1
2  strengur_a = "þetta er a strengurinn minn"
3  strengur_b = " og þetta er b strengurinn minn"
4
5  # Nú get ég sameinað þessa strengi með því að setja annan þeirra fyrir aftan hinn
6  sameinadir_a_og_b = a + b
7
8  # ef við prentum út strenginn fáum við
9  "þetta er a strengurinn minn og þetta er b strengurinn minn"
10 #takið eftir að það er bil á milli strengjanna, það er eingungis vegna þess að b
       strengurinn er skilgreindur þannig að fyrst kemur bil fremst í strengnum
11
12 # röðin skiptir máli þegar strengir eru sameinaðir svona
13 sameinadir_b_og_a = b + a
14
15 # þetta útprentað skilar okkur:
16 " og þetta er b strengurinn minnþetta er a strengurinn minn"
17
18 # Takið eftir að þarna er ekkert bil á milli strengjanna.
19
20 # Á þessu er hægt að svindla:
21 fyrsta_nafn = "Valborg"
22 seinna_nafn = "Sturludóttir"
23 fullt_nafn = fyrsta_nafn + " " + seinna_nafn
24
25 # Þarna sameinaði ég þrjá strengi þar sem ég vissi að hvorugur strengjanna minna
       innihéldi bil ákvað ég að setja það á milli með auka samskeytingu.
```

Þegar við notum * til þess að margfalda streng erum við að *lengja* (e. multiply) hann. Strengjalenging virkar þannig að þú tilgreinir hversu oft, í heilum tölum, þú vilt að strengurinn sé endurtekinn.

Kóðabútur 3.4: Strengjalenging

```
1  eitt_ord = "kex"
2  eitt_ord*3
3
```

```
#skilar okkur
"kexkexkex"
```

## 3.3 Vísar í streng

Strengir eru af einhverri lengd, 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() (fyrir length).

Kóðabútur 3.5: Stafabilafjöldi

```
strengur1 = "kex"
# þessi strengur er augljóslega þrjú stafabil
len(strengur1)
# skilar okkur 3

strengur2 = "kex með smjöri, osti og sultu"
len(strengur2)
# skilar okkur 29, takið eftir að tóm bil og greinamerki teljast með
```

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.

Kóðabútur 3.6: Vísir 1

```
strengur = "kex"
# þessi strengur er augljóslega þrjú stafabil

print(strengur[1])

# þetta skilar okkur stafnum 'e'
```

Eins og sést í kóðabút **??** 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 3.5) 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.

### 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 því eins og þeir eru eða endurskilgreina þá alveg.

### 3.3.2 Neikvæðir vísar

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 að þá myndi verða til tvíræðni (e. ambiguity). 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. Við notum vísana svona: [vísir sem á að byrja fyrir framan : vísir sem á að hætta fyrir framan: stærð á skrefi sem á að taka við lesturinn]. Þarna eru semsagt tveir tvípunktar sem er verið að nota og tölurnar sem koma á milli þeirra er afmörkunin á því hvað við viljum lesa upp úr tilteknum streng. Nú er vert að nefna að þegar við notum þessa málskipan eru ákveðin gildi sjálfgefin, það er við þurfum ekki að taka þau fram.

Sjálfgefin gildi við að ná í hlutstreng:

1. vísirinn sem við byrjum fyrir framan er fremsti stafurinn, eða fremst.
2. vísirinn sem við hættum fyrir framan er fyrir aftan aftasta stafinn, svo síðasta táknið er lesið.
3. skrefastærðin er sjálfgefin 1, það er að við skoðum hvert einasta tákn og hoppum ekki yfir neitt stak.

Kóðabútur 3.7: Hlutstrengir

```
1  strengur = "kex með smjöri, osti og sultu"
2
3  # mig langar að sækja allan strenginn
4  sami_strengur = strengur[::] # hér nota ég þann kost í Python að setja inn fyrir mig
       sjálfgefin gildi
5
6  #mig langar að sækja það sem kemur fyrir aftan táknið x í kex
7  aftan_x = strengur[3::]
8
9  #mig langar ekki í síðasta stafinn
10 nema_sidasti = strengur[:-1:]
11
12 #mig langar í allan strenginn nema annan hvern staf
13 annar_hver = strengur[::2]
14
15 # mig langar að sækja orðið kex og mig langar einnig að sækja orðið sultu
16 kex = strengur[0:3]
17 sultu = strengur[-5:]
18
19 #einnig er hægt að lesa afturábak með því að taka neikvæða skrefastærð
20 ofugur = strengur[::-1]
```

## 3.4 Aðferðir á strengi

Áður en aðferðir á strengi eru kynntar þarf að útskýra stuttlega hvað aðferðir eru. Við höfum séð print() fallið notað t.d. í kóðabút 2.5 og í kóðabút 3.5, það er innbyggt fall í Python sem prentar það sem beðið er um á staðalúttak. Það fall virkar eitt og sér og þarf bara að skrifa nafnið á því, gera sviga og setja inn í svigana það sem við viljum láta fallið fá . 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.

er ég búin að úts staðalúttak?

vísa í falla kaflar

vísa í eitthvað se útskýrir viðföng

Kóðabútur 3.8: Aðferðir kynntar

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

fylla inn í þenna bút

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".

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

Takið eftir því að þarna er orðið lykilorðið „skilar"það er 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.

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 gagnast okkur að skoða. Höfundur hvetur þó til að lesandi geri ítarlegar tilraunir og prófanir.

# 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, þau gögn sem við viljum geyma sem 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 strengir. 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 # Fyrsti listinn okkar er tómur
2 listinn_minn = []
3
4 # þegar við skilgreinum lista aðgreinum við stökin með kommum
5 nyr_listi = ["núllta stakið", 1, 2, 3.0, "fjórða stakið", [5]]
```

Í kóðabút 4.1 sjáum við að við erum með 6 stök í listanum nyr_listi sem er skilgreindur í línu 6 . 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.

passa að breyta kóðabút til að þe vísun haldist

Ef við hugsum okkur töflureikni eins og Calc eða Excel þá getum við ímyndað okkur að ein lína sé eins og einn listi, hver dálkur er stak í listanum og ein röð er listinn sem heldur utan um þau. 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

Kóðabútur 4.2: Listar af listum

```
1 # Ef við ættum skjal í töflureikni sem héldi utan um allt starfsfólk í fyrirtæki gæti
      hausinn á því litið svona út:
2 # Nafn Tölvupóstur Deild Símanúmer
3
```

```
4  # Svo er hver röð fyrir neðan það útfyllt með upplýsingum um einhvað tiltekið starfsman,
       t.d.:
5  # Jóna Jónsdóttir jona@fyrirtaeki.is Póstur 4445555
6
7  # Ef þetta væri útfært í Python með listum væri það gert svona:
8
9  starfsfolk = [["Jóna Jónsdóttir", "jona@fyrirtaeki.is", "Póstur", "4445555"],
10           ["Kristján Kristjánsson", "kristjan@fyrirtaki.is","Laun","4445589"],
11           ["Halldóra Halldórudóttir", "halldora@fyrirtaeki", "Skrifstofa", "4445500"]]
```

Við tökum eftir því að listinn starfsfolk í kóðabút 4.2 í línu 9 inniheldur þrjá aðra lista, og þeir eru aðgreindir með kommum alveg eins og stökin inni í hverjum innri lista fyrir sig eru einnig 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 er því fyrir okkur til að geta lesið kóðann auðveldlegar. Þetta er ekki eins og inndrátturinn sem við munum sjá og beita í næsta kafla, Segðir, skilyrðissetningar og sanngildi.

## 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 kleyft að nálgast gögn eftir sætisvísum. 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 dæmi.

Kóðabútur 4.3: Listar af listum

```
1  # listi skilgreindur
2  [1,2,3]
3
4  # listi skilgreindur og geymdur í breytu
5  listinn = [1,2,3, "langur strengur sem hefur einnig sætisnúmer"]
6
7  # gögn sótt upp úr listanum
8  listinn[1] # skilar okkur tölunni 2
9
10 # Við getum líka sótt gögn upp úr þeim gögnum sem leyfa það
11 listinn[3][0] # skilar okkur stafnum l
```

Við sjáum í kóðabút 4.3 í línu 11 að þar erum við að keðja (e. to chain) hornklofana okkar. Þetta megum við því að listinn[3] skilar okkur til baka strengnum "langur strengur sem hefur einnig sætisnúmerǫg við megum sækja úr honum stak númer 0 sem er aðgengilegt með því að gera [0] fyrir aftan listinn[3]. Þetta gagnast okkur sérstaklega þegar við lítum aftur á starfsmanahaldið okkar hér á undan og viljum geta sótt gögn upp úr innri listum. Sjáum hvernig við getum fengið upplýsingar sem eru skráðar um tiltekið starfsman úr listanum sem við geymdum í kóðabút 4.2.

Kóðabútur 4.4: Unnið með gögn úr lista

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

```
# þetta skilar okkur
["Jóna Jónsdóttir", "jona@fyrirtaeki.is", "Póstur", "4445555"]

print(starfsfolk[0][0])
# þetta skilar
"Jóna Jónsdóttir"
# þar sem nafnið hennar er 0 stakið í innri listanum
```

### 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.

Kóðabútur 4.5: Unnið með gögn úr lista

```
print(starfsfolk[0][0])

# við munum að þetta skilar
"Jóna Jónsdóttir"
# en við munum að nafnið var óvart vitlaust skráð svo við breytum því

starfsfolk[0][0] = "Jóna Alfreðsdóttir"
# þetta skilar okkur engu því að við vorum hér að skilgreina eitthvað, segja tölvunni að
    geyma eitthvað
# en við erum búin að endurskilgreina starfsfolk listann og ef við köllum í hann núna
    sjáum við að hann er breyttur

print(starfsfolk)
# þetta skilar
[["Jóna Alfreðsdóttir", "jona@fyrirtaeki.is", "Póstur", "4445555"], ["Kristján
    Kristjánsson", "kristjan@fyrirtaki.is","Laun","4445589"], ["Halldóra Halldórudóttir",
    "halldora@fyrirtaeki", "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

- 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 4.6.
- 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] (ef við gerum ráð fyrir að hafa ekki keyrt neinar aðrar aðferðir á hann).
  - 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 4.7
- 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] (gerum ráð fyrir að við höfum ekki keyrt neinar aðrar aðferðir á hann).
  - aðferðin skilar engu svo það er ekkert vit í því að gera x = listinn_minn.sort()
  - sjá notkun í kóðabút 4.8.

Kóðabútur 4.6: .pop() aðferðin

```
test = [1,2,3]

x = test.pop()
# x inniheldur núna töluna 3 og test inniheldur listann [1,2]

y = test.pop(0)
# y inniheldur núna töluna 1 og test inniheldur listann [2]
# ekki er hægt að setja inn vísi sem er ekki til staðar í listanum

z = test.pop(1) # þetta veldur villu því að listinn inniheldur einungis stakið 2 sem er í sæti 0
```

Kóðabútur 4.7: .append() aðferðin

```
test = []

# test er breyta sem inniheldur tóman lista, sem okkur langar að bæta í eftir að hafa búið hann til

test.append(1)
# nú inniheldur test stakið 1 í vísi 0, [1]

test.append("nú bætum við streng aftast í listann")
# nú inniheldur test listann [1, "nú bætum við streng aftast í listann"]

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"])
# athugum hér að hornklofar skilgreina lista og innan í svigum aðferðarinnar er bara eitt hornklofapar svo við erum bara að bæta við einum lista aftast í test listann okkar
print(test) s
# skilar okkur þessu:
# [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"]]

# bætum nú aftast í innri listann með .append() aðferðinni
test[2].append("hér var bætt aftast í innri listann, ekki er komið nýtt stak í test")
```

```
19
20 print(test)
21 # skilar okkur þessu:
22 # [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"]]
23 # lengdin á listanum er enn bara 3 og vísarnir eru 0,1 og 2.
```

Athugum hér að í kóðabút 4.7 í línu ?? þá er stak í vísi 0 1, stak í vísi 1 er "nú bætum við streng aftast í listannǫg í vísi 2 er stakið ["hér er heill listi", "með nokkrum stökum", ęn hann er samt einn stakur listi", ǫg telst því sem að bæta við einu staki"]. Þar sem stakið í vísi 2 er einnig listi þá má nota aðferðina .append() á hann og það sem gerist í línu ?? er að gögn eru sótt upp úr listanum með hornklofanotkun, eins og í kóðabút 4.4. Þannig keðjum við og getum bætt aftast í þennan innri lista með .append().

Kóðabútur 4.8: .sort() aðferðin

```
1  # test = [1,6,3,1]
2
3  test.sort()
4  # nú er listinn breyttur
5
6  print(test)
7  # skilar : [1,1,3,6]
8
9  test = ["b", "a", "m", "z"]
10 test.sort()
11 print(test)
12 # skilar : ["a", "b", "m", "z"]
13
14 test = [1, "6"]
15 test.sort()
16 # hér verður villa því að tölur og strengir eru ekki samanburðarhæfir
17
18 starfsfolk = [["Jóna Jónsdóttir", "jona@fyrirtaeki.is", "Póstur", "4445555"],
19            ["Kristján Kristjánsson", "kristjan@fyrirtaki.is","Laun","4445589"],
20            ["Halldóra Halldórudóttir", "halldora@fyrirtaeki", "Skrifstofa", "4445500"]]
21 starfsfolk.sort()
22 print(starfsfolk)
23 # skilar
24 # [["Halldóra Halldórudóttir", "halldora@fyrirtaeki", "Skrifstofa", "4445500"],
25   ["Jóna Jónsdóttir", "jona@fyrirtaeki.is", "Póstur", "4445555"],
26   ["Kristján Kristjánsson", "kristjan@fyrirtaki.is","Laun","4445589"],]
27
28 # Bætum við auðkennisnúmeri fremst í listann
29 starfsfolk = [[1927, "Jóna Jónsdóttir", "jona@fyrirtaeki.is", "Póstur", "4445555"],
30  [782, "Kristján Kristjánsson", "kristjan@fyrirtaki.is","Laun","4445589"],
31  [981,"Halldóra Halldórudóttir", "halldora@fyrirtaeki", "Skrifstofa", "4445500"]]
32
33 starfsfolk.sort()
34 print(starfsfolk)
35 # skilar
36 #  [[782, "Kristján Kristjánsson", "kristjan@fyrirtaki.is","Laun","4445589"],
37    [981,"Halldóra Halldórudóttir", "halldora@fyrirtaeki", "Skrifstofa", "4445500"],
38    [1927, "Jóna Jónsdóttir", "jona@fyrirtaeki.is", "Póstur", "4445555"]]
39
40 # En ef það gleymist að setja inn auðkennisnúmer þegar nýjum starfskrafti er bætt við í
       listann:
41 starfsfolk.append(["Þórður Hjaltalín", "thordur@fyrirtaeki.is", "Markaðs", "4445571"]
42 print(starfsfolk)
```

```
# skilar
#  [[782, "Kristján Kristjánsson", "kristjan@fyrirtaki.is","Laun","4445589"],
   [981,"Halldóra Halldórudóttir", "halldora@fyrirtaeki", "Skrifstofa", "4445500"],
   [1927, "Jóna Jónsdóttir", "jona@fyrirtaeki.is", "Póstur", "4445555"],
   ["Þórður Hjaltalín", "thordur@fyrirtaeki.is", "Markaðs", "4445571"]]

starfsfolk.sort()
# þetta veldur núna villu þar sem raðað er eftir fremsta staki hvers lista og fremstu
    stökin eru ekki af týpum sem hægt er að bera saman
```

# 5. Segðir, skilyrðissetningar og sanngildi

Kóða má skipta í segðir (e. expressions) og yrðingar (e. statements). Segðir eru línur þar sem eitthvað er metið sem gildi, ef við líkjum því við tungumál væru það setningar þar sem einhver niðurstaða fæst eins og „er rigning”? Yrðingar eru línur þar sem eitthvað er sett fram sem þarf ekki að meta, í tungumáli væru það setningar á borð við „mér er kalt”

Þessi skipting er ekkert sérlega merkileg að svö stöddu en 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 búlsk gildi eða sanngildi. 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 og eru upprunin úr búlískri algebru [1] (e. Boolean algebra). Nú er það flestum kunnug staðreynd að tölvur vinna með 0 1 í grunninn, en hvernig það er notað í æðri forritunarmálum (e. high level programming languages) er ekki eins augljóst.

Í þessum kafla verður farið yfir búlísk gildi, samanburð (e. comparison) og samanburðarvirkja (e. comparison operators), rökvirkja (e. logical operators) og svo skilyrðissetningar (e. conditional statements).

## 5.1 Sanngildi

Eins og kom fram í inngangi kaflans eru búlsk gildi 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, með því að skoða mismunandi aðstæður, fengið rökrétt svar við t.d. spurningunni „er rigning og er mér kalt?” sem við getum skrifað sem s1 og svo annarri spurningu sem er „er rigning eða er mér kalt?” sem við getum kallað s2.

[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 og því hvetur höfundur til að lesandi fletti allavega upp wikipedia greininni.

Tafla 5.1: Sanntafla

| p | q | s1 | s2 |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 |

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. 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  test = True
2  # þetta geymir gögn af týpunni Bool
3
4  test = true
5  # þetta veldur villu þar sem nú er verið að biðja um að test innihaldi það sama og
       breytan true vísar á
6  # munurinn er í stóru og litlu t.
7
8  test = "True"
9  # þetta er ekki af týpunni Bool heldur er þetta strengur
10
11 test = False
12 # þetta geymir gögn af týpunni Bool
```

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, það er 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?

Nú þurfum við nýtt hugtak, við erum búin að kynnast reiknivirkjum (e. arthimic operators) eins og + og - í kafla 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

```
# nú viljum bera saman einhver gögn, búum okkur til breytur til að bera saman
strengur1 = "abc"
strengur2 = "bcd"
strengur3 = "3"
tala1 = 3
tala2 = 3.0
tala3 = 4

# nú erum við komin með nokkrar breytur til að gera prófanir á:

# byrjum á að skoða hvort að 3 sé jafngilt 4 eða tveir jafnlangir strengir séu jafngildir
tala1 == tala3
# þetta skilar False

strengur1 == strengur2
# þetta skilar False

# en hvað með þetta?
tala1 == tala2
# þetta skilar True þar sem til að geta borið talnatýpur saman er þeim kastað í
    sambærileg gögn (skoðum kast í seinna í kaflanum)

strengur3 == tala1
# þetta skilar False þar sem ekki er verið að vinna með eingöngu gögn af talnatýpum

# Allt það sem skilar okkur True með == skilar okkur False með !=
# og öfugt, það sem skilar False með == skilar okkur True með !=

# skoðum þá minna en og stærra en

strengur1 < strengur2
# þetta skilar okkur True þar sem strengur1 er framar í stafrófinu, ekki er verið að bera
    saman lengdina á strengjunum

tala1 < tala2
# þetta skilar okkur False þar sem tölurnar eru jafngildar, sáum það að hér að ofan, og ö
    nnur getur ekki verið bæði minni og jöfn á sama tíma

tala1 <= tala2
```

```
37 # þetta skilar okkur True þar sem spurt er hvort að tala1 sé annað hvort minni en eða
       jöfn hinni breytunni
38
39 # Það skiptir máli hvernig goggarnir snúa, a > b hér er spurt hvort a sé stærra en b, b <
       a, hér er spurt hvort b sé minna en a (sem er sama spurningin).
```

### 5.2.2 Rökvirkjar

Rökvirkjar (e. logical operators) í Python eru þrír, þeir eru **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 segja okkur um ástandið á þeim einhvern veginn saman.

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.

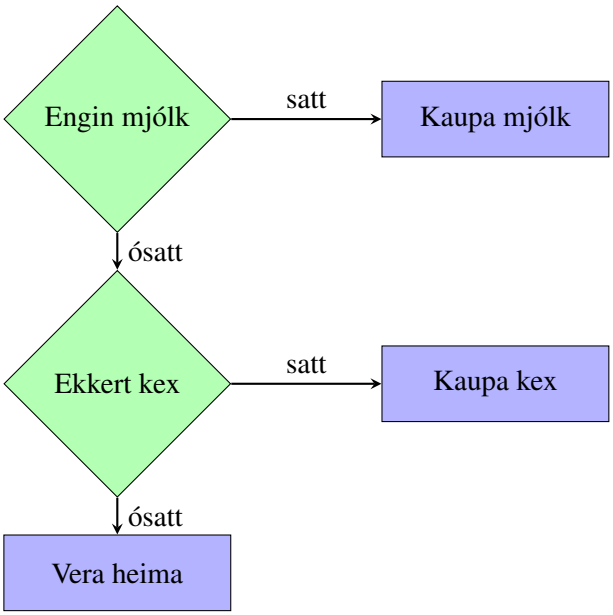
- **and** til þess að segð með þessum rökvirkja sé sönn þurfa báðar hliðar að vera sannar, annars er hún ósönn
  - Það má líta á _og rökvirkjann eins og margföldun, hann hefur forgang umfram _eða.
  - Þar sem satt er 1 og ósatt 0 þá ef við margöldum með 0 fáum við alltaf 0 út.
  - „það er heitt úti” 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, annar er hún ósönn.
  - Það má líta á _eða rökvirkjan 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”.

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 [2] 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.

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

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”. 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.

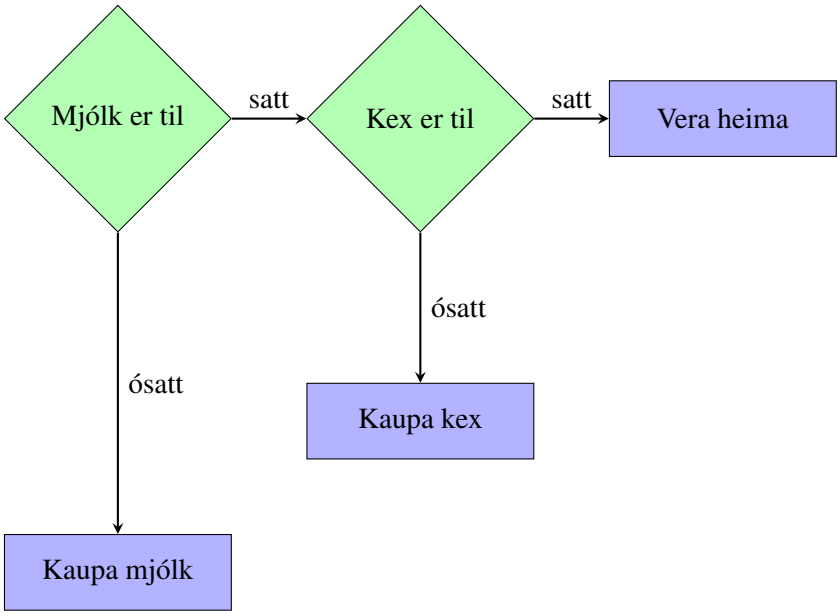
Skoðum því nú hvað það þýðir að nota skilyrðissetningar (e. conditional statements) í Python með lykilorðunum **if - elif - else**, sem verða þó kynnt annarri röð.

### 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 . 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.

orð

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.




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

Skoðum nú kóðabút 5.3 og hvað er átt við með réttum inndrætti, hér er einungis sýnt if setning ein og stök. 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 m = True # m er yrðingin hvort að til sé mjólk eða ekki, við gefum okkur í upphafi að til
      sé mjólk því er m upphafsstillt sem True
2
3 if(not m):
4    # fara í búð
5    print('við fórum í búðina og keyptum mjólk')
6
7 # Við áttum mjólk svo þetta prentast ekki, við förum framhjá aðgerðunum sem eru undir þ
      essari if skipun
```

### 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, og það eru einu skilyrðin. 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.

Kóðabútur 5.4: else notað

```
m = True # m er yrðingin hvort að til sé mjólk eða ekki, við gefum okkur í upphafi að til sé mjólk því er m upphafsstillt sem True

if(not m):
   # fara í búð
   print('við fórum í búðina og keyptum mjólk')
else:
   # takið eftir að hér var ekkert skilyrði athugað
   # við förum hingað inn ef skilyrðið fyrir ofan var ekki keyrt
   print('vera heima')

# Við áttum mjólk svo við fáum út 'vera heima'

if(3 < 4):
   print("þrír er minna en fjórir")
else:
   print('ég fer ekki hingað inn, því 3 er vissulega minna en fjórir, en það er gott að vera við öllu búin')
```

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". 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.

Kóðabútur 5.5: elif notað

```
m = True # m er yrðingin hvort að til sé mjólk eða ekki, við gefum okkur í upphafi að til sé mjólk því er m upphafsstillt sem True
k = False # k er yrðingin hvort að til sé kex eða ekki, við gefum okkar að í upphafi sé ekki til neitt kex og k er því False


if(not m):
   # fara í búð
   print('við fórum í búðina og keyptum mjólk')
elif(not k):
   # keyrist ef not m var metið sem False OG not k metið sem True
   print('við fórum í búðina og keyptum kex')
else:
   # takið eftir að hér var ekkert skilyrði athugað
   # við förum hingað inn ef skilyrðið fyrir ofan var ekki keyrt
   # tökum eftir að það skilyrði er nú elif setningin okkar
   print('vera heima')

# Við áttum mjólk en við áttum ekki kex, svo not k var metið sem True og við fáum svo við
```

```
    fáum 'við fórum í búðina og keyptum kex'

# Við förum ekki niður í else nema að allt fyrir ofan sé ósatt.

if(5 < 4):
   print("fimm er minna en fjórir")
elif(4 < 4):
   print("fjórir er minna en fjórir!")
elif(3<4):
   print("þrír er minna en fjórir")
elif(2<4):
   print("þetta prentast ekki því að ég get bara farið inn í eina klausu í hverri
    skilyrðissetningu")
   print("og það gerðist hérna fyrir ofan, svo þetta skilyrði verður ekki keyrt")
else:
   print('ég fer ekki hingað inn, því 3 er vissulega minna en fjórir, en það er gott að
    vera við öllu búin')
```

### 5.3.4 Hreiðrun

Hreiðrun (e. nesting) þýðir að setja 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

```
m = True # m er yrðingin hvort að til sé mjólk eða ekki, við gefum okkur í upphafi að til
    sé mjólk því er m upphafsstillt sem True
k = False # k er yrðingin hvort að til sé kex eða ekki, við gefum okkar að í upphafi sé
    ekki til neitt kex og k er því False
p = 100 # p eru hversu mikinn pening við erum við á okkur.

if(not m):
   # fara í búð, ef við komumst hingað inn þá var m False og not m var þá True
   # nú sjáum við hvað þarf að gerast til þess að við getum keypt mjólkina ef við komumst
    hingað inn
   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 k):
   # keyrist ef not m var metið sem False OG not k metið sem True
   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:
   # takið eftir að hér var ekkert skilyrði athugað
   # við förum hingað inn ef skilyrðið fyrir ofan var ekki keyrt
   # tökum eftir að það skilyrði er nú elif setningin okkar
   print('vera heima og geyma allan peninginn')

# Við áttum mjólk sem var heppilegt, því að við hefðum ekki getað keypt hana
# Við áttum ekki kex, og svo heppilega vildi til að við vorum með nóg fyrir kexpakka og þ
    ví fáum við niðurstöðuna:
# '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 viljum fá inntak (e. input) frá notandanum.

Þá lærum við um nýtt innbyggt fall í Python sem heitir input(). Það sem input() gerir er að það tekur við streng sem það birtir á úttaki (e. output) þar sem notandinn sér það og getur sett eitthvað inn. 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. Skoðum kóðadæmi í kóðabút 5.7, þar sem við geymum svarið frá notanda í breytu og viðfangið sem við setjum inn í fallið er strengur sem inniheldur spurninguna sem notandinn sér. Vert er að athuga að allt sem er sett inn sem svar í þennan glugga er kastað í streng, það er breytan sem við notum til að taka á móti því sem input() fallið skilar er af týpunni strengur. 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

Kóðabútur 5.7: input() fallið notað

```
svar = input('skrifaðu nafnið þitt')

# hér birtist gluggi fyrir okkur til að skrifa eitthvað inn í sem er merktur sem
    'skrifaðu nafnið þitt'
# ég skrifa nafnið mitt og ýti á vendibil (e. enter) takkann og þá er svarið mitt komið í
    breytuna svar

print('halló', svar, 'mikið heitir þú fallegu nafni')

# Fyrir inntakið Valborg myndi nú prentast:
# 'halló Valborg mikið heitir þú fallegu nafni'
```

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

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

```
# Búðu til breytu sem inniheldur nafnið þitt:
nafn = 'Valborg'
# prentaðu hana út ásamt "halló:"
print('Halló: Valborg') # hér er ekkert verið að nota breytuna nafn heldur horfir
    forritarinn á kóðann og sér að tölvan eigi að prenta út 'Valborg' og setur það inn
    handvirkt.

# Annað dæmi væri að finna tákn sem er í miðjum streng
strengur = "þessi strengur hefur 29 tákn!"
# miðjan í þeim streng er því í tákni 14 sem tölvan getur náð í með len(strengur)//2
# og getur því fundið miðjutáknið með
strengur[len(strengur)//2] # skilar okkur tákninu í vísi 14

```

```
# en hér er einnig hægt að taka nokkur skref sem er algengt að byrjendur geri
print(len(strengur)) # þetta skilar okkur 29
print(29/2) # þetta skilar okkur 14.5 eða print(29//2) sem skilar 14
print(strengur[14]) # þetta skilar okkur tákninu í miðjunni
```

Eins og sést í kóðabút 5.8 að til að komast að því að setja vísi 14 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

```
# Þessi strengur inniheldur bara tölur
talnastrengur = "123"

# og því má kasta honum í heiltölu eða fleytitölu
int(talnastrengur)
# þetta skilar okkur tölunni 123

float(talnastrengur)
# þetta skilar okkur tölunni 123.0

fleytitolustrengur = "123.0"

float(fleytitolustrengur)
# þetta skilar okkur tölunni 123.0

int(fleytitolustrengur)
# þetta skilar okkur villu því að heiltölur eru ekki með punkti þó að þetta sé í grunninn
    sama tala og 123

# Nánast öllu má svo kasta í streng, þá er hverju tákni úthlutað sætisvísi og geymt eins
    og það er skrifað.
str(123)
# þetta skilar okkur "123"

str([1,2,3])
# þetta skilar okkur "[1,2,3]"

# Það sem gerist þegar við köstum í lista er að hverju ítranlegum hluta af gögnum er
```

```
    varpað í stak í listanum
# Skoðum betur í kafla um lykkjur hvað ítranlegt þýðir.

list("123.0")
# þetta skilar okkur ["1", "2", "3", ".", "0"]

list(123.0)
# Þetta skilar okkur villu því að tölur eru ekki ítranlegar, þær hafa ekki sætisnúmer
    (ekki eins og tölvan skilgreinir þau).
```

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.

afla um try

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.

# 6. Lykkjur

Í Python eru til tvær tegundir af lykkjum, þær heita **for** og **while**. 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.

Nöfnin á lykkjunum verða ekki þýdd sérstaklega í þessari bók, öðruvísi en að nota hugmyndina á bakvið nöfnin til að segja að við gerum eitthvað á meðan eða fyrir hvert stak þó lykkjurnar séu enn kallaðar for og while lykkjur.

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 uppþvottahanska, 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 hlutirnir hægra megin eru búnir. Þetta á ágætlega við um virkni for lykkja.

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 á ágætlega við um virkni while lykkja.

## 6.1 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**, **break** og **in**.

Það sem þessi lykilorð gera er má sjá í kóðabút 6.4, sjá einnig kóðabúta 6.2 og 6.5:

- 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 málskipunar 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.

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.

Skoðum nú lykilorðið in áður en lengra er haldið til þess að öðlast dýpri skilning á því hvernig for lykkjan virkar.

Það sem þetta orð gerir er að spyrja hvort að eitthvað sé „í" einhverju öðru eða taka fyrir stak í hlut sem hefur vísa eða stök. Þannig að þetta býr til segð sem skilar sanngildi eða einu tilteknu tákni eða staki úr hlut.

Kóðabútur 6.1: Lykilorðið in

```
# við viljum vita hvort að eitthvað tiltekið tákn sé í einhverjum ákveðnum streng:
"a" in "Valborg"
# þetta er eins og að spyrja með samanburðarvirkja hvort eitthvað sé satt eða ekki
# þetta skilar okkur True þar sem táknið a er til staðar í strengnum Valborg

"x" in "Valborg"
# þetta skilar okkur False þar sem táknið x er ekki til staðar í strengum Valborg
```

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.

## 6.2 Lykkjur

Til þess að keyra kóða endurtekið án þess að afrita og líma eða handvirkt keyra hann oft, þá notum við lykkjur. Lykkjur eru kóðabútur sem keyrist endurtekið eftir ákveðnum reglum, þær lykkjur sem eru til í Python eru for lykkjur og while lykkur. 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").

### 6.2.1 For

For lykkkjur nota lykilorðið **for** ásamt lykilorðinu textbfin, við höfum séð hvernig á að nota lykilorðið in, og fengið samlíkingu úr daglegu lífi til að átta okkur á því hvað for lykkjan gerir. Nú skulum við líta á kóðabút 6.2 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 6.2: For lykkjur

```
# við byrjum á að skilgreina streng
strengur = "Valborg"

# athugum hvernig á að setja upp okkar fyrstu lykku
for stafur in strengur:
   print(stafur)

# Hérna búum við til breytuna stafur sem fær gildi hvers staks fyrir sig uppúr breytunni
    strengur
# úttakið verður
# V
# a
# l
# b
# o
# r
# g

# sjáum hvernig við getum sett skilyrði inn í lykkjuna

for stafur in strengur:
   if(stafur == 'a'):
      print(stafur)

# úttakið okkar verður
# a

# Skoðum nú hvernig við ítrum í gegnum lista:
listinn_minn = [0, "0", [0]]

for x in listinn_minn:
   # nú hleypur x í gegnum stökin í breytunni listinn_minn
   print(x)

# þetta skilar okkur úttakinu
# 0
# 0
# [0]
```

Tökum eftir í kóðabút 6.2 að breytan sem notuð er til að ítra í gegn er ekki skilgreind áður en við komum að lykkjunni, við þurfum ekki að gera x = eða strengur = áður en við komum að lykkjunni. 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.

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.

For lykkjur eru því helst gagnlegar þegar við vitum hversu oft við viljum að lykkjan keyri, 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 þegar við sóttum nokkur tákn upp úr streng eða lista, nema þau eru viðföng í fall svo þau eru aðgreind með kommum en ekki tvípunktum. Viðföngin eru heilartölur og þeim er raðað svona:

1. talan sem á að byrja að nota (hér má sleppa því að setja þetta inn því að sjálfgefið gildi er 0)
2. talan sem á að hætta fyrir framan (þetta verður að setja inn, því að þetta er aðalatriðið)
3. tala sem segir til um skrefastærðina (sjálfgefið gildi er 1 og þessu má sleppa)

Kóðabútur 6.3: range() fallið

```
# við viljum prenta út sléttar tölur á bilinu 0 og upp í 10 þar sem 10 er með
for tala in range(11):
   # við tókum fram að við hættum fyrir framan 11 og næsta heila tala þar fyrir neðan er 10 svo hún er með
   if(tala%2 == 0):
      # ef afgangurinn úr deilingu með tveimur er jafngildur 0, þá hlýtur tala að vera deilanleg með tveimur og er því slétt
      print(tala)

# þetta skilar okkur úttakinu:
# 0
# 2
# 4
# 6
# 8
# 10

# við viljum núna skoða hvaða tölur eru deilanlegar með 3 á bilinu 10 upp í 20, við viljum ekki telja 20 með.
for tala in range(10, 20):
   if(tala%3 == 0):
      print(tala)

# þetta skilar okkur úttakinu
# 12
# 15
# 18

# við viljum núna prenta aðrahvora tölu á bilinu 100 til 106, báðar meðtaldar
for tala in range(100, 107, 2):
   print(tala)

# þetta skilar okkur úttakinu:
# 100
# 102
# 104
# 106

# við viljum að lykkjan okkar prenti út einhver skilaboð ákveðið oft, segjum 3 sinnum.
for x in range(3):
   # fyrst er x 0, svo 1, svo 2, sem þýðir að lykkjan keyrir þrisvar sinnum
   print('bílalúgudýraspítali')

# þetta skilar okkur úttakinu
# bílalúgudýraspítali
# bílalúgudýraspítali
# bílalúgudýraspítali
```

Nú vitum við að ef okkur vantar eitthvað ákveðið oft, eða við vitum nákvæmlega hversu oft við viljum að lykkjan keyri þá getum við beitt range() fallinu. Nú skulum við skoða hvernig lykkju lykilorðin sem við skoðuðum í undirkaflanum Gagnleg lykilorð (6.1) nýtast með for lykkjum í kóðabút 6.4.

Kóðabútur 6.4: Lykilorðin pass continue og break notuð með for lykkju

```
# við viljum búa til lykkju sem á að rúlla í gegnum 1000 tölur en við erum ekki viss hvað á að gera innan í lykkjunni
for x in range(1000):
   pass
# hér fáum við ekkert úttak en jafnframt enga villu.

# við viljum búa til streng úr stórum tölum í einhverjum lista, tölum sem eru hærri en 50
storar_tolur = ""
for x in [1, 2, 52, 9, 53, 2]:
   if (x < 50):
      continue
      print(x)
   storar_tolur += str(x)

print(storar_tolur)
# þetta skilar okkur úttakinu
# 5253

# við viljum búa til lykkju sem ítrar í gegnum lista af tölum en ef hún sér töluna 13 þá hættir hún keyrslu
listi_af_tolum = [1,5,7,9,13,15,17,18]
for tala in listi_af_tolum:
   if(tala == 13):
      break
      print(tala)
   else:
      print(tala)

# þetta skilar okkur úttakinu
# 1
# 5
# 7
# 9
```

Tökum eftir að í kóðabút 6.4 þá er tilfellið sem notað er til að sýna continue kannski ekki mjög nytsamlegt þar sem auðvelt væri að gera bara if (x > 50) : storar_tolur += str(x) en þar sem við erum að reyna að kynnast notkun lykilorðsins þá er þetta ágætis dæmi. Það sem gerist er að ef x < 50 er satt þá förum við inn í skilyrðissetninguna og þar sem við förum inn í hana þá rekumst við á orðið continue og um leið og það er lesið þá förum við efst í lykkjuna og úthlutum x nýju staki, print(x) gerist ekki og því er eina úttakið okkar það sem við prentum eftir að keyrslu lykkjunnar lýkur. Við sjáum að lykkjunni er lokið því að print(storar_tolur) kemur fram í sama inndrætti og for lykilorðið, sem þýðir að kallið print(storar_tolur) tilheyrir ekki lykkjunni, er ekki undir henni.

Þegar við notum break gerist það sama, um leið og break er notað þá hættir keyrsla lykkjunnar, breytan tala fær ekki nýtt gildi og hún endar með sem talan 13. Að sama skapi þá prentum við ekki út 13 því að break gerist fyrir ofan print() kallið. Þetta lykilorð getur reynst ómetanlegt þegar við skoðum while lykkjur.

### 6.2.2 While

While lykkjur nota lykilorðið **while** og keyra „á meðan" eitthvað skilyrði er enn satt. Þær eru helst ganglegar 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 keyrslunnar 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 6.5 til þess að sjá hvernig má auðveldlega lenda í vandræðum við gerð slíkra lykkja og hvernig uppsetning þeirra lítur út.

Kóðabútur 6.5: while lykkjur

```
# Grunnuppbyggingin kynnt:
while(True):
   # inndreginn kóði sem tilheyrir lykkjunni - stef lykkjunnar
   pass

# þessi lykkja keyrir að eilífu vegna þess að skilyrðið fyrir henni er True og ekkert breytir því í stefi hennar

while(True):
   # nú ætlum við að reyna að komast út
   break

# við komumst út úr lykkjunni, hún keyrði einu sinni og hætti strax keyrslu, ekki mjög gagnleg lykkja en hún keyrði allavega ekki að eilífu

while(False):
   print('þetta mun aldrei prentast því að stef lykkjunnar mun aldrei keyrast')

# prófum núna að gera segð sem skilyrði

x = 5
while(x > 1):
   # Við munum keyra stef lykkjunnar að minnsta kosti einu sinni þar sem segðin er sönn í fyrstu keyrslu, 5 er vissulega stærri en 1
   # en nú verðum við að gera eitthvað sem hefur áhrif á x þannig að lykkjan okkar eigi möguleika á að hætta, það er við verðum að gera það að verkum að x > 1 verði einhvern tímann ósatt.
   if(x > 1):
      print("talan er", x, "og hún er enn stærri en 1")
      x -= 1

# úttakið verður:
# talan er 5 og hún er enn stærri en 1
# talan er 4 og hún er enn stærri en 1
# talan er 3 og hún er enn stærri en 1
# talan er 2 og hún er enn stærri en 1

# Tökum aðeins sértækara dæmi, sem byggir á sauðakóða en ekki Python kóða sem hægt er að keyra.
# Til þess að sjá fyrir okkur betur hvað er að gerast.

fjoldi_i_stofu = 5
while(True):
   if(fjoldi_i_stofu == 0):
      slökkva ljós
      break

   # við komumst bara hingað ef það er einhver í stofunni svo við ætlum að telja fjölda í stofu aftur
   fjoldi_i_stofu = talning

# Stef þessarar lykkju byggir á því að tölvan viti hvað slökkva ljós og talning þýðir, en
```

```
    þar sem ætlunin var ekki að gera kóða sem virkar heldur sýna lesanda virknina fær
    lykkjan að vera í friði svona.
```

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 6.6: Ap nota else með while

```
# Við viljum áfram skoða það hvenær á að slökkva ljósin og við getum gert það svona:
while(fjoldi_i_stofu > 0):
   fjoldi_i_stofu = talning
else:
   slökkva ljósin

# nú slökkvast ljósin þegar fjoldi_i_stofu er orðinn 0.

# skoðum dæmi sem hægt er að keyra

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)

# þetta skilar okkur úttakinu
# 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
```

# 7. N-dir

Nú ætlum við að kynnast nýrri týpu, hún heitir **n-d** (lesist ennd) (e. tuple). 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 nd. Líklega eina orðið í íslensku sem inniheldur ekki sérhljóða.

Hún líkist listum að því leitinu til að margar af sömu aðgerðum sem má gera á lista má gera á n-dir. Hún líkist strengjum því að hún er óbreytanleg.

Ástæðan til að nota ndir í stað lista er sú að það getur verið hagkvæmara, ndir nota ekki eins mikið minni, og við sjáum 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 er 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), við getum náð í hluta úr ndinni (með hornklofum eins og hlutstrengi eða hluta úr lista)

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

Kóðabútur 7.1: Ndir skilgreindar

```
1 # byrjum á því að sýna hvernig svigar skilgreina ekki endilega nd
2 a = (3+4)*2
3 # hér fær breytan a gildið 14 því að sviginn er notaður fyrir röð aðgerða.
4
5 b = (1)
```

```
# hér gerir sviginn ekkert og b inniheldur töluna 1, það er sviginn heldur utan um röð aðgerða en þær eru engar

# Nú skilgreinum við ndir
a = () # þetta verður tóm nd
b = (1,) # þetta verður nd sem inniheldur eitt stak, athugið kommunotkunina
c = (1, 1, 2, 2, 5) # þetta verður nd sem inniheldur 5 stök

# Nú skulum við skoða hvaða aðferðir eru til á þessa nýju týpu og notum til þess breyturnar b og c
b.index(1) # skilar okkur úttakinu 0 þar sem talan 1 er í 0ta vísi í ndinni b.
c.count(1) # skilar okkur úttakinu 2 þar sem talan 1 kemur tvisvar sinnum fyrir í ndinni c
c + b # skilar okkur nýrri nd sem inniheldur (1, 1, 2, 2, 5, 1) þar sem b hefur verið skeytt aftan við 1

for tala in c:
   print(tala)

#skilar okkur úttakinu
# 1
# 1
# 2
# 2
# 5

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

c[1:3] # skilar okkur úttakinu (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 notað þær til að spara minni þegar við þurfum litla lista sem þarf bara að nota tímabundið. Einnig geta þær nýst til að halda utan um breytur sem á svo að nota hverja í sínu lagi seinna. Við sjáum þetta í kóðabút 7.2 og tökum eftir að vissulega megi útfæra fyrri þrjú, af þessum fjórum atriðum nefndum, með listum þá er það ekki endilega það besta í stöðunni, ef við ætlum að hugsa um gagnaheilindi og minnisnotkun.

Kóðabútur 7.2: Ndir notaðar

```
# byrjum á því að skilgreina eina nd til að nota
notanda_upplysingar = ("valborg", "rosalega gott lykilorð", "netfang@internet.is")

# við fengjum villu við að gera notanda_upplysingar.sort(), notanda_upplysingar.append(x) eða notanda_upplysingar.remove(x) því að þessar aðferðir eru ekki til á hlut af taginu nd
# þær eru til þess fallnar að breyta listum og við megum ekki breyta ndum

# en það sem við getum gert er að sækja þessi gögn og vista í breytum
# vissulega er hægt að gera þetta:
notandanafn = notanda_upplysingar[0]
lykilord = notenda_upplysingar[1]
netfang = notenda_upplysingar[2]

# en það sem er öflugra og snjallara að gera er að láta Python "aftroða" (e. unpack) á eftirfarandi máta:
```

```
14 notandanafn, lykilord, netfang = notenda_upplysingar
15
16 # nú parast hvert stak í ndinni á þessar breytur í þeirri röð sem þær eru skilgreindar.
17 # núllti vísir á fremstu breytuna og svo koll af kolli
18
19 # Hér er mikilvægt að til þess að aftroða ndinni með þessum hætti þurfa að vera
       jafnmargar breytur vinstra megin við jafnaðarmerkið eins og eru stök í ndinni, annars
       fáum við villu.
```

Að ná í nokkrar breytur í einu getur verið ákjósanlegt þegar við erum við fáar breytur, eins og við sjáum í kóðabút 7.2 þá er lína 14 ágætlega þægileg (auðvelt að skilja hana og lesa) á meðan línur 8-10 taka óþarflega mikið pláss og eru ekkert endilega læsilegri fyrir vikið. 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.

# 8. Orðabækur

Ný týpa sem vil ætlum að nú að fást við heitir **orðabækur** (e. dictionaries) og lykilorðið þeirra er **dict**. Orðabækur 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, það er þær geyma fyrir okkur gögn af öðrum týpum. Orðabækur eru þó frábrugnar listum að því leitinu til að þær eru *óraðaðar*, sem þýðir að þær hafa enga sætisvísa. Við getum því ekki sótt gögn í orðabækur með því að vita *hvar* þau eru við þurfum að vita *hver* þau eru. Orðabækur eru mjög öflugt fyrirbæri, og því þess virði að kynna sér vel hvernig þessi týpa virkar.

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, ef við grípum einhvers staðar í lyklakippuhringinn og horfum á þann lykil sem er hægra megin við fingurgóma okkar sem fremsta lykilinn og númerum lyklana eftir því þá er ekki víst að næst þegar við tökum kippuhringinn upp að við grípum á sama stað niður og að sami lykill verði fremstur. En 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. Við skoðum betur hvernig við getum gengið úr skugga um aðgreinanleika og hvað það þýðir.

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 Lyklar og gildi

Eins og kom fram í inngangi er gögnum í orðabókum skipt niður á lyklana sem ganga að þeim. Lyklarnir þurfa að vera aðgreinanlegir, hvað þýðir það? Skoðum innbyggða fallið hash() til þess að átta okkur á því hvað má nota sem lykil, sjá kóðabút 8.1. Það sem hash() fallið gerir er að skila okkur einu tilteknu heiltölu gildi, tveir hlutir sem eru álitnir jafngildir fá sömu heiltöluna úr hash()

fallinu. Ekki er hægt að kalla í hash() af öllum týpum, því sumar týpur eru óhakkanlegar.

Í kóðabút 8.2 sjáum við hvernig á að skilgreina orðabók, lykla og gildispör og hvernig á að aðgreina stök. Stak í orðabók er eitt lykla og gildispar. Lyklar þurfa að vera hakkanlegir og því geta listar og orðabækur ekki verið lyklar en hvað sem er má vera gildi, eins og sést í kóðabút 8.1. Lyklar og gildi geta verið breytur, en þá eins og alltaf þegar við notum breytur þurfum við að vera búin að skilgreina breytuna áður en við notum hana.

Kóðabútur 8.1: Skoðum hash() fallið til að skilja aðgreinanleika gagna

```
a = [1,2,3] # a er listi
hash(a) # skilar villu

b = 12
hash(b) # skilar 12

c = 12.2
hash(c) # skilar stórri heilli tölu.

d = 12.0
hash(d) # skilar 12

# svo hash skilar okkur tölu eftir því hvernig má túlka gögn sem eina heila tölu ef það
    er mögulegt, svo b og d eru óaðgreinanleg og því má ekki nota bæði sem lykla í sömu
    orðabók.
```

Ástæðan fyrir því að við þurfum að skilja þetta er vegna þess að við þurfum að átta okkur á því hvað má setja sem lykil og hvers vegna við getum kannski ekki notað einhvern tiltekinn lykil.

Kóðabútur 8.2: Orðabækur kynntar

```
# Skilgreinum orðabækur:
ordabok1 = {} # inniheldur ekkert
ordabok2 = {'lykill': 'gildi'} # inniheldur strenginn lykill sem lykil og svo er
    tvípunktur sem aðgreinir lykilinn frá gildinu sem er strengurinn 'gildi'

ordabok3 = {1: 'gildi á lykli 1 sem er heiltala, 2: 'gildi sem er á lykli 2', 3: 'takið
    eftir að pörin eru aðgreind með kommu'}

# Hvernig á að sækja gögn ef þau eru ekki með sætisnúmeri:
ordabok2['lykill'] # þetta skilar okkar 'gildi'
ordabok3[3] # þetta skilar okkur 'takið eftir að pörin eru aðgreind með kommu'

# Hvernig á að setja inn gögn eða endurskilgreina lykil
ordabok2['lykill'] = 'nýtt gildi' # nú er búið að endurskilgreina gildið á þessum lykli
ordabok2['nýr lykill'] = 'nýtt gildi' # nú er búið að búa til nýjan lykil sem fékk
    eitthvað gildi
```

## 8.2 Ítrað í gegnum orðabækur

Nú höfum við séð for lykkjur í kafla 6 og hvernig mátti lykkja í gegnum lista í kóðabút 6.2. 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 er útfærsluatriðið sem ákveðið var að yrði gert þannig að þegar óskað er eftir að ítra í gegnum orðabók eru lyklarnir hennar eingöngu teknir fyrir. Hins vegar er hægt að gera bæði lykla og gildi eða einungis gildin aðgengileg með því að kalla í aðferðirnar .items() og .values(), að rúlla í gegnum orðabók án þess að taka fram einhverja aðferð er eins og að hafa kallað í aðferðina .keys(). Nú eru nöfnin á þessum aðferðum ágætlega lýsandi:

- ordabok.keys() við fáum í hendurnar ítranlegan hlut sem inniheldur alla lykla í lista úr breytunni ordabok (e. view)
- ordabok.values() við fáum í hendurnar ítranlegan hlut sem inniheldur öll gildi í lista úr breytunni ordabok (e. view)
- ordabok.items() við fáum í hendurnar ítranlegan hlut sem inniheldur lista af tvenndum (nd með tveimur stökum) úr breytunni ordabok (e. view).

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 8.3.

Kóðabútur 8.3: Skoðum hvernig megi ítra í gegnum orðabækur

```
# Skilgreinum orðabók sem heldur utan um bókasafnið okkar, þar sem lykill er höfundur og gildi er listi af bókum sem við eigum eftir þann höfund:
bokasafn = {'Beazley': ['Python Essential Reference'],'Halldór Laxness': ['Íslandsklulkka', 'Salka Valka], 'Auður Haralds': ['Hlustið þér á Mozart', 'Læknamafían', 'Hvunndagshetjan'] }

# til þess að vinna með alla þá höfunda sem við eigum til getum við gert eftirfarandi:

for hofundur in bokasafn:

   # við vitum að bokasafn[hofundur] gefur okkur gildi þess höfundar, sem er í okkar tilfelli alltaf listi

   # athugum nú hvað við eigum margar bækur eftir höfundana með því að skoða lengd listans
   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)


# við getum einnig gert þetta:

for hofundur in bokasafn.keys():
   # nú er hofundur breytan sem hleypur í gegnum ítranlega hlutinn sem .keys() skilar alveg jafngild breytunni hofundur í lykkjunni fyrir ofan
   # svo vid getum enn gert bokasafn[hofundur]

   if hofundur < "Miðgildi":
      print('höfundurinn", hofundur, "er framarlega í stafrófinu")
   else:
      print("höfundurinn", hofundur, "er aftarlega í stafrófinu")

# en ef við viljum einungis skoða gildin, það er listana sem innihalda bækurnar sjálfar og okkur er sama um höfundana

for bokalisti in bokasafn.values():
   # bokalisti er breyta sem inniheldur lista
   # svo við getum ítrað í gegnum hann
   for bok in bokalisti:
```

```
40        # ef hún er með langan titil viljum við prenta hana út:
41        if(len(bok) > 20):
42           print(bok)
43
44  # Og ef við viljum skoða bæði í einu án þess að þurfa að sækja gildið á lykilin sjálf með
        orðabók[lykill]:
45
46  for hofundur, bokalisti in bokasafn.items():
47     # ef bókalistinn er ákveðið langur þá langar okkur að prenta út nafnið á höfundinum
48     if(len(bokalisti) > 5):
49        print(hofundur, "er mjög vinsæll höfundur")
50     elif(len(bokalisti) > 2):
51        print(hofundur, "er frekar vinsæll höfundur")
52     elif(len(bokalisti) > 1):
53        print(hofundur, "gæti verið vinsælli")
54     elif(len(bokalisti) == 1):
55        print(hofundur, "er vissulega til staðar")
56     else:
57        print(hofundur, "á ekki tiltall til einnar bókar í þessu bókasafni")
```

Nú gera lykkjurnar í línu 6 (engin aðferð, bara lykkjað í gegnum bókasafnið eins og Python gerir á sjálgefinn máta) og þessi í línu 46 (.items() aðferðin sem fær par sem látið er í tvær breytur hofundur og bokalisti) nokkurn veginn það sama en við tökum eftir að sú seinni er aðeins læsilegri því að breytan bokalisti er nokkuð lýsandi fyrir það hvað hún inniheldur á meðan bokasafn[hofundur] gæti verið strengur eða nd eða eitthvað allt annað (eins og enn önnur orðabók).

# 9. Mengi

Mengi (eða sett) 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 vélinn 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 = {}
```

## 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’, og ’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 í línu 10 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.

Kóðabútur 9.2: Mengi skilgreind

```
1  # Skilgreinum mengi með endurtekningum
2  mengid_mitt = {1,2,3,4, 1, 2, 3, 4}
3  print(mengid_mitt)
4  # úttakið verður
5  # {1, 2, 3, 4}
6
7  # notum lykilorðið til að búa til mengi úr streng
8  print(set("Valborg Sturludóttir vinsamlegast"))
9  # úttakið verður
10 # {’b’, ’i’, ’ó’, ’n’, ’l’, ’a’, ’g’, ’d’, ’o’, ’V’, ’ ’, ’e’, ’S’, ’r’, ’s’, ’u’, ’t’,
       ’m’, ’v’}
```

## 9.2 Aðgerðir

Aðgerðir sem hægt er að gera á set er að bæta við staki, **add**(), fjarlægja stak, **remove**() og uppfæra mengið með mörgum stökum, **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.

Stærðfræðilegar aðgerðir sem hægt er að gera með mengi er að finna sniðmengi eða sammengi tveggja mengja, eða mengi sem inniheldur einungis stök sem eru ekki í báðum mengjunum sem verið er að sameina. Allt þrennt af þessu er hægt að gera með innbyggðum föllum sem taka við tveimur mengjum og skila einu mengi til baka eða aðferðum á mengi sem breyta þá menginu sem aðferðin var kölluð á með tilliti til mengisins í viðfanginu.

Þetta reynist 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.

# 10. Föll

Föll (e. functions), eins og lykkjur, eru kóðabútur 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 skrifa þau.

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 bók 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: kynna min og m einhvern tímann

- 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")
- min("breyta sem ég vil fá minnsta gildið úr")
- min(nokkrar, tölur, og, ég, vil, vita, hver, er minnst)
- range(0,50)
- type("breyta sem viðfang sem ég vil vita týpuna á")

Þetta eru innbyggð föll, 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 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 sem við skilgreindum þau í
- 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 Pyhton 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 einum inndrætti innar en def lykilorðið tilheyrir fallinu okkar.

Kóðabútur 10.1: Föll skilgreind

```
1  # Búum til okkar fyrsta fall, það sem það á að gera er að prenta "Halló Heimur!" því að þ
       að er tölvunarfræðileg klisja sem má ekki sleppa.
2  def prentunarfall():
3     # fallið heitir prentunarfall og það er notað með því að kalla í prentunarfall()
4     print("Halló Heimur!")
5
6  # athugið að hér notuðum við lykilorðið def og svo skrifuðum við nafnið á fallinu, það
       sem fallið gerir er að keyra eina print skipun.
7  # en hér er ekkert úttak, þar sem við kölluðum ekki í fallið heldur skilgreindum það
       eingöngu
8
9  # til að kalla í fallið og nota það þarf að keyra eftirfarandi línu, að sjálfsögðu eftir
       að hafa keyrt skilgreininguna að ofan
10 prentunarfall()
11 # úttakið verður
12 # Halló Heimur!
```

## 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.

Til dæmis ef við værum að raða í íslenska stafrófsröð væri lítið vit í því að bjóða notandanum að setja inn sem viðfang eitthvert annað stafróf. Eða ef við værum að lóðsa bílstjóra með GPS myndum við ekki vilja leyfa bílstjóranum að setja inn nýjar götur.

Annað sem mikilvægt er að átta sig á er að viðföng fá breytuheiti sem eru aðeins aðgengileg innan fallsins en ekki utan þess. Því ætlum við að skoða nýtt hugtak áður en við skoðum stöðubundin og sjálfgefin viðföng.

### 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. Hins vegar 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.

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. Vilja meina 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. Dæmi væri:
def prentunarfall(strengur):
print(strengur)
Þegar við köllum í fallið með prentunarfall() þá fáum við villu því að það vantar að setja inn hvað strengur á að vera, strengur er nafnið á viðfanginu en það skiptir ekki öllu máli á þessu stigi málsins, öðru en að villumeldingin sem við fáum upp segir vantar eitt stöðubundið viðfang sem er strengur. Við þurfum þá að kalla í fallið og setja eitthvað inn í svigann þegar við köllum í það, eins og prentunarfall('Halló Heimur!') sem myndi þá prenta út "Halló Heimur!".

Kóðabútur 10.2: Stöðubundin viðföng kynnt

```
1 def fall(a,b,c):
```

```
2     a**b/c
3
4  # Þarna eru viðföngin a b og c sett inn í fallið, sem þýðir að þegar fallið er notað þarf
       að setja gögn, sem á að gefa fallinu, í þessari röð, því það skiptir máli hvernig á
       að vinna með viðföngin inni í fallinu.
5  # a þarf að koma fyrst því að það er það sem er hafið í veldið af b sem er svo deilt með c
6  # þar sem við þekkjum röð aðgerða og þessa tilteknu reiknivirka vitum við að a = 2, b = 3
       og c = 5 gefur okkur ekki sömu niðurstöðu og a = 5, b = 2 og c = 3
```

Athugum að í kóðabút 10.2 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.

Annað sem vert er að athuga er að þó að við hefðum kallað í fallið þá hefði ekkert skéð, þar sem fallið gerir ekkert við útreikninginn. Við sjáum í kafla 10.4 hvað hægt er að gera annað en að prenta bara út. En þetta fall hvorki prentar út né skilar og það er eingöngu hugsað til útskýringar á fyrirbærinu stöðubundin viðföng.

Lítum á kóðabút 10.3 til að sjá hvernig betur mætti fara með skilgreininguna úr kóðabút 10.2.

Kóðabútur 10.3: Stöðubundin viðföng kynnt

```
1  def hefja_i_veldi_og_deila(grunntala,veldisvisir,deiling):
2     grunntala**veldisvisir/deiling
3
4  # Þarna er öllum sem hafa aðgang að þessum kóða gerð skýr skil á því hvað þetta fall
       gerir og greinilegt sé að inntakið eigi að vera tölur
5  # Fallið gerir þó enn ekkert annað en að vera uppfræðandi því að ekkert kemur út þegar
       við notum fallið
```

Ef við rifjum upp hugtakið gildissvið úr fyrri undirkafla og skoðum hvernig það á við um kóðabút 10.3 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) 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 tiltekin radíus. Fallið gætið litið svona út:
def hringur(radius, midja):
    # það sem fallið gerir.
Nú ef mig langar til að fá einhvern hring í hendurnar get ég ekki sagt við vélina láttu með hafa hring með radíus radius. 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.

er búið að tala u except sem mun

### 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. Dæmi væri:

```
def prentunarfall(strengur = "Halló Heimur!"):
    print(strengur)
```

Ef við köllum í fallið með prentunarfall() og gefum því ekkert viðfang þá notar tölvan skilgreininguna sem er innan svigans þar sem viðföngin eru tekin fram. Sjáum þetta í kóðabút 10.4, 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 10.4: Sjálfgefin viðföng kynnt

```
1  def hefja_i_veldi_og_deila(grunntala = 1,veldisvisir = 1,deiling = 1):
2     print(grunntala**veldisvisir/deiling)
3
4  hefja_i_veldi_og_deila()
5  # hér köllum við í fallið en gefum því ekkert, við fáum þó enga villu, því sjálfgefnu
       skilgreiningarnar eru notaðar
6  # úttakið er
7  # 1
8
9  hefja_i_veldi_og_deila(2, 2, 2)
10 # hér gefum við fallinu í té tölur í röð eins og þetta séu stöðubundin viðföng
11 # ef við gefum upp fleiri viðföng fáum við villu, ef við gefum upp færri þá parast þau í þ
       eirri röð sem þau berast
12 # úttakið verður
13 # 2
14
15 hefja_i_veldi_og_deila(deiling = 2)
16 # hér notum við sjálfgefin gildi fyrir allt nema deiling, við tökum sérstaklega fram hvað
       það á að vera
17 # úttakið verður
18 # 0.5
19
20 hefja_i_veldi_og_deila(deiling = 2, veldisvisir = 1, grunntala = 4)
21 # hér notum við sjálfgefin gildi, í annarri röð en þau voru skilgreind, það veldur ekki
       villu því að tölvan parar viðföngin eftir nafni
22 # úttakið verður
23 # 2
24
25 # skoðum núna blöndu, endurskilgreinum sama fallið með nýjum viðföngum:
26
27 def hefja_i_veldi_og_deila(grunntala,veldisvisir,deiling = 1):
28    print(grunntala**veldisvisir/deiling)
29
30 # Athugum hér að stöðubundnu viðföngin verða að koma á undan.
```

```
# köllum nú í fallið með mismunandi aðferðum:

hefja_i_veldi_og_deila() # hér fáum við villu því búist er við að notandi gefi grunntala og veldisvisir í té.

hefja_i_veldi_og_deila(1,2) # hér fáum við á úttakið 2

hefja_i_veldi_og_deila(1, 2, 2) # hér fáum á úttakið 0.5

hefja_i_veldi_og_deila(1, 2, deiling = 4) # hér fáum við á úttakið 0.25

# notum breytur til að setja inn í fallið:
tala = 2
veldi = 2
deila = 2
hefja_i_veldi_og_deila(tala, veldi, deiling = deila) # hér fáum við á úttakið 2
```

## 10.4 Skilagildi

Nú höfum við séð hvernig á að búa til föll, við höfum séð hvernig á að láta föllin vinna með sjálfgefið inntak og það næsta sem við viljum skoða er hvernig á að láta föllin okkar skila gildum þannig að hægt sé að fá útkomu úr þeim 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 3.8), til þess að geta notað úttkomuna þá getum við búið til breytu sem grípur það sem aðferðin *skilar*. Í kóðabútum 10.3 og 10.4 þá er fallið hefja_i_veldi skilgreint, en aldrei er hægt að vinna eitthvað með úttkomuna úr útreikningnum, það sem er gert er annað hvort ekkert eða úttkoman er 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 (eins og flest lykilorðin sem við höfum séð hingað til, að mati höfundar). 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 10.5

Kóðabútur 10.5: Hvernig á að láta fall skila gildi með return skipuninni

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

# nú höfum við séð mýgrút leiða til að kalla í fallið svo við ætlum að einbeita okkur að því hvernig megi nota það sem fallið skilar í stað þess hvernig eigi að kalla í það með mismunandi viðföngum.
# Athugum að  hefja_i_veldi_og_deila(2, 2, 2) skilar okkur 2

tala = fall(2,2,2)
print(tala)
# hér verður úttakið 2

utreikningur = fall(2,2,2) * 4
print(utreikning)
# hér verður úttakið 8

breyta = utreikningur * tala
print(breyta)
```

```
17 # hér verður úttakið 16
```

Nú höfum við skilað einu gildi sem er heiltala eða fleytitala og unnið með hana með þeim aðgerðum og reiknivirkjum 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.

Sjáum í kóðabút 10.6 hvernig hægt er að skila mörgum gildum og setja þau í breytu eða breytur.

Kóðabútur 10.6: Hvernig á að skila mörgum gildum

```
1  def skilum_morgum_gildum(strengur, tala, listi):
2     # Þetta fall tekur við streng, tölu og lista
3     # Fallið skilar tveimur tölum og listanum aftur óbreyttum
4     # Fyrri talan er hversu oft strengurinn í viðfanginu kom fyrir í listanum
5     # Seinni talan er hversu margar tölur í listanum eru stærri en talan í viðfanginu
6
7     skilatala = 0
8     strengja_talning = 0
9     for x in listi:
10       # x stendur það stak sem verið er skoða úr listanum listi
11       if x == strengur:
12          strengja_talning +=1
13       if(type(x) == int or type(x) == float ):
14          if (tala < x):
15             skilatala += 1
16    return strengja_talning, skilatala,  listi
17
18 # hér gerist ekkert nema að við köllum í fallið
19 skilum_morgum_gildum("halló", 2, ["halló", "bless", 11, 6])
20 # Þetta skilar okkur úttakinu
21 (1, 2, ["halló", "bless", 11, 6])
22
23 # En ef okkur langar ekki að fá út n-d þá getum við gert dálitla töfra
24 talning_strengs, staerri_tolur, listinn = skilum_morgum_gildum("halló", 2, ["halló",
       "bless", 11, 6])
25
26 # Nú fáum við ekkert úttak en breyturnar innihalda 1, 2 og ["halló", "bless", 11, 6] í þ
       essari röð
27
28 # Við getum ekki notað fleiri en eina breytu nema að fjöldi þeirra passi við fjölda
       skilagilda:
29 a, b = skilum_morgum_gildum("halló", 2, ["halló", "bless", 11, 6])
30
31 # þetta veldur villu því annað hvort er í boði að hafa eina breytu sem inniheldur n-d eða
       þrjár breytur sem taka hver við sínu skilagildi.
```

## 10.5 Lokun

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 10.7, 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 10.7: Innri föll kynnt

```
1  def breyta_strengjum(strengur):
```

```
   # þetta fall tekur við streng og skilar honum eftir að hafa breytt honum einhvern veginn

   def fremsti_stafur_er_nuna_aftastur(strengur):
      # þetta fall tekur við streng og lætur fremsta stafinn í honum aftast og aftasta stafinn fremst og skilar breytingunni
      # ef strengurinn er eitt eða færra stafabil gerir fallið ekkert við strenginn og skilar honum óbreyttum
      if(len(strengur > 2)):
         fremst = strengur[0]
         aftast = strengur[-1]
         strengur = aftast + strengur[1:-1] + fremst
         return strengur
      else:
         return strengur

   # nú er skilgreiningin á innra fallinu búin og við erum stödd í ytra fallinu
   # nú getum við kallað í innra fallið

   skilastrengur = fremsti_stafur_er_nuna_aftastur(strengur)
   return skilastrengur

# Nú getum við kallað í ytra fallið
strengur = breyta_strengjum("halló")
# strengur inniheldur núna
# "óllah"

# En ytra fallið gerði samt eiginlega ekki neitt, hvers vegna erum við að nota það en ekki innra fallið?
strengur =  fremsti_stafur_er_nuna_aftastur("halló")

# hér fáum við villu því að það er ekkert fall til, í því gildissviði sem við erum stödd, sem heitir þessu nafni sem við höfum aðgang að, þess vegna verðum við að nota ytra fallið.

```

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 10.8 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 þá ætti frekar að beita henni heldur en klösum.

Kóðabútur 10.8: Lokun kynnt

```
def prentunarfall(strengur):
   # prentunarfall tekur við einu viðfangi
   # prentunarfall skilar falli sem prentar þann streng ásamt viðbót
   def prentum():
      # prentum tekur ekki við neinu en notar viðfangið úr ytra fallinu
      # prentum skilar strengnum úr viðfanginu ásamt viðskeytingunni 'hér er viðbót af akademískri ástæðu'
      return str(strengur) + " hér er viðbót af akademískri ástæðu"
   return prentum

a = prentunarfall('halló heimur')
print(a) # skilar okkur að a sé fall sem sé geymt í einhverju minnishólfi
print(a()) # nú köllum við í fallið a sem skilar okkur úttakinu 'halló heimur hér er viðbót af akademískri ástæðu'
```

```
13
14 # Við megum svo nota prentunarfallið okkar aftur með einhverju öðru viðfangi
15 b = prentunarfall('nýr strengur')
16 print(b()) # skilar okkur úttakinu 'nýr strengur hér er viðbót af akademískri ástæðu'
17
18 # Sjáum ögn haldbærari notkun
19
20 def niðurtalning(n):
21    # niðurtalning tekur við tölu
22    # niðurtalning skilar falli sem telur niður úr þeirri tölu
23
24    def teljari():
25       # nú lendum við í vandræðum með n, þar sem við höfum ekki aðgang að því hérna
26       # og við lögum það með því að nota lykilorðið nonlocal (eða óstaðvært) til að segja
     vélinni að leita annarsstaðar að gildi fyrir n
27       nonlocal n
28       while(n>-1):
29          print(n)
30          n -= 1
31
32    return teljari
33
34 # nú getum við skilgreint fall sem telur niður frá 10 og notað það þegar okkur hentar
35 teljum_fra_tiu = niðurtalning(5)
36
37 # nú hentar okkur að nota það:
38 teljum_fra_fimm()
39 # það skilar okkur úttakinu
40 # 5
41 # 4
42 # 3
43 # 2
44 # 1
45 # 0
```

Tökum eftir að í skilgreiningum á a og b í kóðabút 10.8 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. Við notuðum breytuna teljum_fra_fimm til þess að geyma fyrir okkur að kalla í fallið niðurtalning() með viðfanginu 5, svo þegar okkur hentar þá getum við beitt fallinu sem telur niður fyrir okkur. 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 Exercises

This is an example of an exercise.

**Exercise 10.1** This is a good place to ask a question to test learning progress or further cement ideas into students' minds. ■

## 10.7 Problems

**Problem 10.1** What is the average airspeed velocity of an unladen swallow?

# Part Two

# Atriðisorðaskrá