Gathering on Raspberry Pi

—Minecraft Pi with Python—

2014 年 7 月 5 日

1 Python について

Python（パイソン）は、広く使用されている汎用のスクリプト言語である。コードのリーダビリティが高くなるように言語が設計されていると言われ、その構文のおかげで、C などの言語に比べて、より少ないコード行数でプログラムを表現することができると言われている。小規模なプログラムから大規模なプログラムまで、さまざまなプログラムをクリアに書けるように、多くのコードが提供されている。

1.1 特徴

Python はインタプリタ上で実行されることを前提に設計されており、以下のような特徴をもっている:

- 動的な型付け
- ガベージコレクション
- マルチパラダイム
- モジュール、クラス、オブジェクト等の言語の要素が内部からアクセス可能であり、リフレクションを利用した記述が可能。

1.2 名前の由来

Python は、オランダ人のグイド・ヴァンロッサムによって開発された。名前の由来は、イギリスのテレビ局 BBC が製作したコメディ番組『空飛ぶモンティ・パイソン』である。Python という英単語は爬虫類のニシキヘビを意味し、Python 言語のマスコットやアイコンとして使われることがある。

1.2.1 グイド・ヴァンロッサム

グイド・ヴァンロッサム（Guido van Rossum、1960 年 1 月 31 日 - ）は、オランダ人プログラマであり、プログラミング言語 Python の作者として知られている。Python コミュニティでは “Benevolent Dictator for Life”（BDFL、慈悲深き終身独裁者）として知られ、Python 開発を常に監督し決定を下す（本人のブログで自身を “Python’s BDFL” と紹介している）。2012 年末まで Google に勤務し、勤務時間の半分は Python 開発に費やしていた。2013 年 1 月以降は Dropbox に勤務している。

1.3 起源

Python の起源について、ヴァンロッサム自身は 1996 年に次のように書いている。

> Over six years ago, in December 1989, I was looking for a “hobby” programming project that would keep me occupied during the week around Christmas. My office … would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python’s Flying Circus).
>
> （日本語訳）「6 年以上前の 1989 年 12 月、私はクリスマス前後の週の暇つぶしのため「趣味」のプログラミングプロジェクトを探していた。オフィスは閉まっているが、自宅にはホームコンピュータがあるし、他にすることがなかった。私は最近考えていた新しいスクリプト言語のインタプリタを書くことにした。それは、ABC からの派生であり、Unix/C ハッカーの注意をひきつけるかもしれないと考えた。ちょっとしたいたずら心から（『空飛ぶモンティ・パイソン』の熱烈なファンだったというのも理由の 1 つ）、プロジェクトの仮称を Python にした」

2000 年にはさらに次のように書いている。

> Python’s predecessor, ABC, was inspired by SETL - Lambert Meertens spent a year with the SETL group at NYU before coming up with the final ABC design!
>
> （日本語訳）「Python の先祖である ABC は、SETL の影響を受けている。Lambert Meertens は ABC の設計に到達するまでニューヨーク大学の SETL グループで約 1 年を過ごしていた」

1.4 バージョンについて

Python は現在バージョン 2.x から 3.x への移行期にある。

2014 年 7 月時点での 2.x の最新バージョンは 2.7、3.x の最新バージョンは 3.4 となっている。3.x は 2008 年に 3.0 が登場して以来もう 6 年経つのだが、2.x との後方互換性がない（2.x のコードが 3.x で動くとは限らない）ため、好んで 2.x を使う人も多いようだ（ネット上では 2.x と 3.x のコードの情報が混在しているので注意が必要）。

本講義でも、本来は最新バージョンである 3.4 を用いるべきであるが、Minecraft Pi との連携は 2.7 が推奨されているため、2.7 を中心に解説を行う。

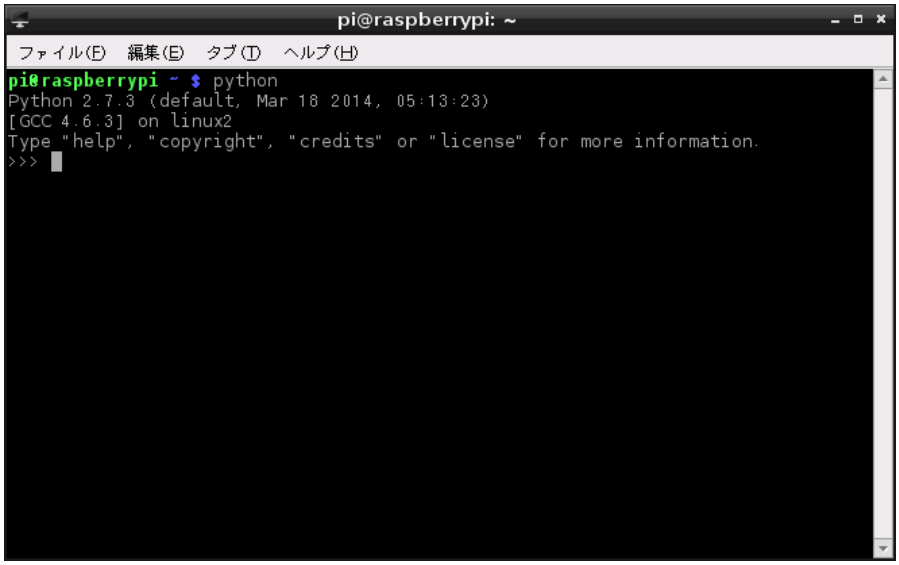
2　気軽な入門編

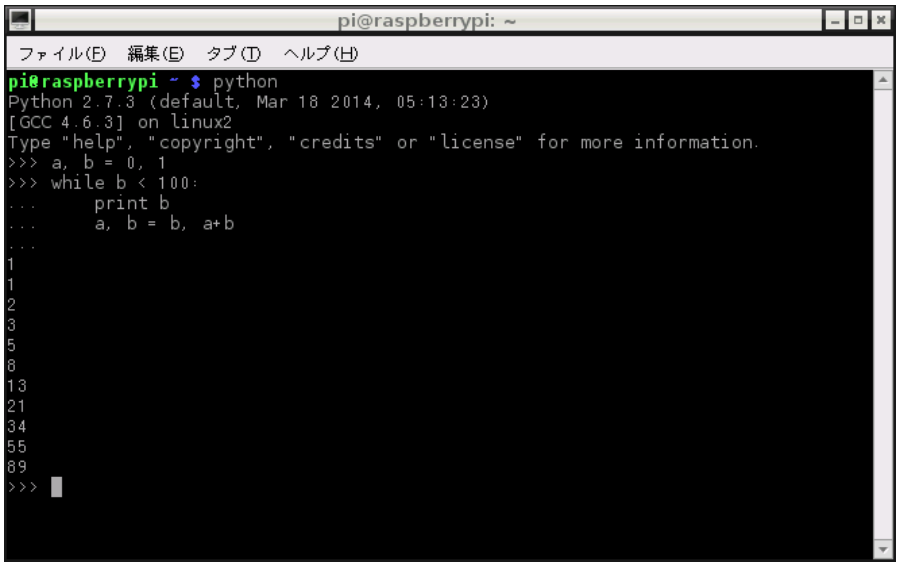
Python プログラムをまずは気軽な気持ちで初めてみよう。

Raspberry Pi のデスクトップにある LXTerminal を起動、そこで “Python” と入力しエンターキーを押すと、Python インタープリタの「対話モード」が起動する。

2.1　対話モード

Python インタープリタの「対話モード」が起動されると左に大なり記号３つ（>>>）が表示される。このコマンド入力を促すプロンプトを「プライマリプロンプト」という。継続行には「セカンダリプロンプト」が出てくるが、これはデフォルトではドット３つである（...）。インタープリタは、バージョンを示すウェルカムメッセージを表示してからプロンプトを出す。複数行で論理構成する場合には、継続行が必要である。





セカンダリプロンプトを終了するには空白のままエンターキーを押し、プライマリプロンプトを終了するには、Ctrl+D（Ctrl キーと D キー）を押す。

2.2 Python を電卓として使う

簡単なコマンドから少し使ってみよう。まずは対話モードに入るべく、インタープリタを起動してプライマリプロンプト（>>>）が出るのを待つ（すぐ出なければ何かがおかしい）。

2.2.1 数値

インタープリタは簡単な電卓になる。式を入れれば答えを表示するからだ。式の文法は直感的で、演算子としての +，-，*，/ （加法、減法、乗法、除法）は他の言語と同じように使える。丸括弧で式のグルーピングもできる。例：

```
>>> 2+2
4
>>> # これはコメント
... 2+2
4
>>> 2+2 # コードの行にもコメントできる
4
>>> (50-5*6)/4
5
>>> 8/5
1
```

最後の計算で、あれ？と思った人は正しい。

実は、Python 2.x は「整数同士の計算結果は整数で出す」という決まりがある（3.x ではそのルールが撤廃され、整数同士の計算でも結果がちゃんと小数になる）。もし、8/5 の結果をきちんと表示したいならば：

```
>>> 8/5.0
1.6
>>> 8.0/5
1.6
>>> 8.0/5.0
1.6
```

のように、計算する数のどれかを小数にする必要がある。ちなみに、整数同士の除法は、切り下げされる：

```
>>> 7/3
2
>>> 7/-3
-3
```

整数と小数を相互に変換する関数（float() と int()）がある。int() では小数点以下が切り捨てられる：

```
>>> float(1)
1.0
>>> float(7/3)
2.0
>>> float(7/3.0)
2.3333333333333335
>>> int(1.5)
1
>>> int(-1.5)
-1
>>> pi=3.141592653589793
>>> int(pi)
3
```

等号（=）は変数に値を代入するのに使う。このとき「結果（答え）」は表示されない：

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

代入は複数の変数に同時に行える：

```
>>> x = y = z = 0 # x,y,z をゼロとする
>>> x
0
>>> y
0
>>> z
0
```

変数は使用前に「定義」（値を代入）しなければならない。やらなければエラーになる：

```
>>> # 未定義の変数にアクセスを試みる
... n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

複素数もサポートされている。虚数は接尾辞「j」または「J」を付加して示す（i ではないので注意）。実部がゼロでない複素数は「(**実部** + **虚部** j)」と書く。また、「complex(**実部**, **虚部**)」のように、関数で生成で

きる：

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0, 1)
(-1+0j)
>>> 3+1j*3
3+3j
>>> (3+1j)*3
9+3j
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

最後の計算から分かるとおり、実部と虚部は小数として扱われる。

z.real, z.imag はそれぞれ複素数型変数 z の実部と 虚部を表す。このように、複素数型変数名のあとに、「.real」や 「.imag」をつけると、その変数の実部や虚部になる：

```
>>> (1+1j) / 2
(0.5+0.5j)
>>> z = -1+2j
>>> z.real
-1.0
>>> z.imag
2.0
```

整数と小数を相互に変換する関数群（float() と int()）は、複素数には作用しない —— 複素数を実数に変換する正しい方法なんて存在しないからだ。その大きさを求めるには、関数 abs() を使う：

```
>>> float(1+1j)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't convert complex to float
>>> abs(1+1j)
1.4142135623730951
```

対話モードでは、最後に表示した式を変数「_」（アンダースコア）に代入してある。つまり、Python を電卓として使うと計算を続けていくのが楽である：

```
>>> tax = 8.0 / 100
>>> price = 100
>>> price * tax
8.0
```

```
>>> price + _
108.0
>>> int(_)
108
```

この変数（_）はユーザーからはリードオンリーとして扱うべきものだ。値を明示的に代入してはならない —— 特異な振る舞いをもつこのビルトイン変数を隠蔽する、同じ名前だが無関係のローカル変数を生成することになるからだ。

2.2.2 文字列

Python では数値に加え、文字列を扱うこともできる。これはさまざまな方法で表現できる。ちなみに、\ はバックスラッシュといい、¥キーで入力できる。引用符にはシングルクォートもダブルクォートも使える：

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

インタープリタは、文字列演算（操作）の結果を、あたかもそれを入力にタイプするかのように表示する。つまり引用符（クォート）で囲み、バックスラッシュによってその引用符やその他のおかしなキャラクタをエスケープするのだ。これは結果値を正確に示すためである。表示の引用符がダブルクォートとなるのは、文字列自体がシングルクォートを含んでダブルクォートを含まない場合のみであり、それ以外はシングルクォートだ。

複数行にわたる文字列リテラルを書く方法はいろいろある。まず、行の最後にバックスラッシュ（\）を置き、次の行に論理が継続することを示すと、続け書きができる：

```
>>> hello = "これは比較的長い文字列で、C言語と同じやり方で\n\
... 複数行のテキストを含んでいる。\n\
...         行頭に入れたスペースが意味を持つので\
... 注意されたし。"
>>> print hello
これは比較的長い文字列で、C言語と同じやり方で
複数行のテキストを含んでいる。
        行頭に入れたスペースが意味を持つので注意されたし。
```

また、トリプルクォート（""" または '''）の対で文字列を囲む、という手がある。この場合、行頭をエスケープしてやる必要はない。文字列にそのまま納まってくれる：

```
>>> hello = """これは比較的長い文字列で、C言語と同じやり方で
... 複数行のテキストを含んでいる。
...         行頭に入れたスペースが意味を持つので
... 注意されたし。"""
>>> print hello
これは比較的長い文字列で、C言語と同じやり方で
複数行のテキストを含んでいる。
        行頭に入れたスペースが意味を持つので
注意されたし。
```

「raw（生の）」文字列で文字列リテラルを作ると、\n は改行に変換されない。また行末のバックスラッシュや、その後ろの改行文字は、データとして文字列に含まれるようになる。だから次のようになる：

```
>>> hello = r"これは比較的長い文字列で、C言語と同じやり方で\n\
... 複数行のテキストを含んでいる。"
>>> print hello
これは比較的長い文字列で、C言語と同じやり方で\n\
複数行のテキストを含んでいる。
```

文字列は + 演算子で連結できるし、* 演算子で繰り返すこともできる：

```
>>> word = 'Help' + 'Me'
>>> word
'HelpMe'
>>> '<' + word*5 + '>'
'<HelpMeHelpMeHelpMeHelpMeHelpMe>'
```

隣接する２つの文字列リテラルは自動的に連結される。つまり上の例の１行目は「word = 'Help' 'Me'」とも書けるわけだ。ただしこれはリテラル同士の場合にのみ有効で、文字列演算には作用しない：

```
>>> 'str' 'ing'             # ←これは OK
'string'
>>> 'str'.strip() + 'ing'   # ←これは OK
'string'
>>> 'str'.strip() 'ing'     # ←これは無効
  File "<stdin>", line 1
    'str'.strip() 'ing'
                  ^
SyntaxError: invalid syntax
```

文字列には添字付け（インデックス付け）をすることができる。この添字（インデックス）は文字列の１文字目が 0 である。１文字のキャラクタは長さが 1 の文字列である。スライス表記で文字列の一部（部分文字列）を指定することができる。これは２つのインデックスをコロンで区切る表記だ：

```
>>> word[4]
'M'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
```

スライスは便利なデフォルト値をもつ。インデックスの第１文字の省略時デフォルトが 0、第２文字の省略時デフォルトが文字列のサイズとなっているのだ。

```
>>> word[:2]   # 最初の２文字
'He'
>>> word[2:]   # 最初の２文字を取った残り
'lpMe'
```

Python の文字列は変更不能である。文字列中のインデックスで指定した位置に代入を行うとエラーが出る：

```
>>> word[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[:1] = 'Splat'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

とはいえこれは、いろいろ組み合わせて新しい文字列を生成すればよい：

```
>>> 'x' + word[1:]
'xelpMe'
>>> 'Splat' + word[4:]
'SplatMe'
```

スライスでは便利な恒等式が成立する。常に「`s[:i] + s[i:] == s`」なのだ：

```
>>> word[:2] + word[2:]
'HelpMe'
>>> word[:4] + word[4:]
'HelpMe'
```

スライスにおかしなインデックスを入れても適切に扱われる。大きすぎる値は文字列の長さに置き換えられるし、上側境界が下側境界より小さいと空の文字列が返されるのだ：

```
>>> word[1:100]
'elpMe'
>>> word[10:]
''
>>> word[2:1]
''
```

インデックスには負の数も使える。これは右から数えることを示す。ただし「-0」は「0」とまったく同じであり、右から数えないことに注意：

```
>>> word[-1]    # 最後の文字
'e'
>>> word[-2]    # 最後から２番目の文字
'M'
>>> word[-2:]   # 最後の２文字
'Me'
>>> word[:-2]   # 最後の２文字を除いて全部
'Help'
>>> word[-0]    # （-0 と 0 は等しいので）
'H'
```

スライスの動作を覚えるには、インデックスとは文字と文字の「間」を指す数字であり、最初の文字の左端が 0 になっている、と考えればよい。つまり n 文字からなる文字列の最後の文字の右がインデックス n となる：

```
 +---+---+---+---+---+---+
 | H | e | l | p | M | e |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

数字の並びは上が文字列中のインデックス 0〜6 の位置を示す。下は対応する負のインデックスだ。インデックス i から j までのスライス（[i:j]）は、境界 i と境界 j に挟まれた文字すべてから成る。

非負のインデックスにおいては、スライスの長さは２つのインデックスの差だ（インデックスの値が文字列長の範囲に収まっている限り）。たとえば word(1:3) の長さは 2 である。

ビルトイン関数 len() は文字列の長さを返す：

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

2.2.3 リスト

Python には複合したデータのための型がいくつかあって、他の種類の値をまとめるのに使える。もっとも万能なのがリスト（list）で、これは角括弧の中にカンマ区切りの値（アイテム）を入れていくことで書ける。リストのアイテムがすべて同じ型である必要はない：

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

リストインデックスも文字列インデックス同様に、0 から始まり、スライス、結合、その他が可能である：

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo!']
```

文字列の場合は要素を指定して個々の文字を変更することはできないが、リストの場合は個々の要素は変更できる：

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

スライスへの代入も可能であり、これによってリストの長さを変えることも、リストの内容をすべてクリアすることもできる：

```
>>> a[0:2] = [1, 12]    # アイテムをいくつか置換
>>> a
[1, 12, 123, 1234]
>>> a[0:2] = []         # いくつか削除
>>> a
[123, 1234]
```

```
>>> a[1:1] = ['bletch', 'xyzzy']  # 挿入
>>> a
[123, 'bletch', 'xyxxy', 1234]
>>> a[:0] = a          # 頭に自分自身（のコピー）を挿入
>>> a
[123, 'bletch', 'xyxxy', 1234, 123, 'bletch', 'xyxxy', 1234]
>>> a[:] = []          # リストをクリア：全アイテムを空リストで置換
>>> a
[]
```

ビルトイン関数 len() はリストにも使える：

```
>>> a = ['a', 'b', 'c', 'd']
>>> len(a)
4
```

リストは入れ子にできる（リストをもつリストが生成できる）：

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> p
[1, [2, 3], 4]
>>> len(p)
3
>>> p[1]
[2,3]
>>> p[1][0]
2
```

この入れ子のリストの末尾に追加することもできる：

```
>>> p[1].append('xtra')
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

最後の例で p[1] と q がまったく同じオブジェクトを参照していることに注目！
つまり、 p[1] と q をまったく同じもの（変数）として扱うことができることが分かる。

2.3 プログラミング、はじめの一歩

Python は 2+2 より複雑なことにももちろん使える。たとえばフィボナッチ数列のはじめの方は、以下のように書くことができる：

```
>>> # フィボナッチ級数：
... # ２項の和により次項が定まっている
... a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
1
1
2
3
5
8
```

この例には、新しい機能をいくつか入れてある。

- 最初の行では「多重代入」を行った。これにより変数 a と b は、新しい値 0 と 1 をそれぞれ得ている。最後の行でも多重代入を使った。こちらでは、代入が行われる前にまず右辺側にある式がすべて評価されることを示した。
- while ループは条件（ここでは b < 10）が真（true）である限り実行を繰り返す。Python では、0 でない整数が真、0 が偽である。この例で使ったテストは単純比較である。標準的な比較演算子の書き方は <（小なり）、>（大なり）、==（等しい・イコール）、<=（小なりイコール）、>=（大なりイコール）、!=（等しくない・ノットイコール）となっている。
  ※プログラミング言語では通常 = は代入を表し、== は等しいことを表している。
- ループのボディ部分にはインデントがかかっている（行頭が字下げされている）。Python では文のグルーピングにインデントを使う。Python は（まだ）インテリジェントな入力行編集機構を提供していないので、インデントはタブやスペースにより、自前で入力しなければならない。実地ではテキストエディタを使い、もっと複雑な入力を行うことになるはずだ。ほとんどのテキストエディタには自動インデント機能がある。対話環境で複合文を入力するときは、最後に空白行を加えて文の完了を知らせてやらねばならない（パーサからは、あなたが入力したのが最後の行がどうか判別がつかないからだ）。ひとつの基本ブロックの中の行同士はインデントを揃える必要があることに注意。

2.4 if 文

Python では他の言語にあるような普通の制御構文が使える。この種の構文でもっともお馴染みなのは多分 if 文だろう：

```
>>> x = int(input("整数を入れて下さい："))
整数を入れて下さい：42
>>> if x < 0:
...     x = 0
...     print '負数はゼロとする')
... elif x == 0:
...     print 'ゼロ'
... elif x == 1:
...     print 'ひとつ'
... else:
...     print 'もっと'
...
もっと
```

elif 部分はもっと続けてもよいし、なくてもよい。また else 部分はオプション（なくてもよい）である。キーワード elif は「else if」の短縮で、インデントだらけになるのを防ぐ。

2.5 その他の制御構文

Python には他のプログラミング言語でもよく使われる for 文（ただし、Python の使い方は特殊）やその他制御構文が多数存在するが、今回は省略する。

また、Python はオブジェクト指向プログラミングが可能であるが、この説明も今回は省略する。オブジェクト指向プログラミングについては、後期の研究ゼミ一回目で Java を用いて解説する予定である。

3 Minecraft との連携

Python は Minecraft Pi edition と連携させることができる。

3.1 Minecraft の操作

Minecraft Pi edition では マウス と キーボード を両方使う必要がある。

3.1.1 マウス操作

上下左右移動	見る方向 (進行方向) の指定
左ボタン	ブロックを除く
右ボタン	ブロックをブロックを置く、ブロックを剣でたたく
ホイール	アイテム一覧から使うものを選択

3.1.2 キーボード操作

キー	動作
W	前進
A	左移動
S	後退
D	右移動
スペース	ジャンプ、ダブルタップで飛行の開始/終了、押し続けるとより高く飛ぶ
シフト	下降、押し続けると低く移動
E	一覧を開く
1-8	アイテム一覧から使うものを選択
ESC	メニューの表示/非表示
TAB	マウスを解放
ENTER	メニューの選択

3.2 Python からの操作

Minecraft Pi が起動しているのを確認したら、LXTerminal を起動して mcpi/api/python までカレントディレクトリを移動して対話的に Python を操作する：

```
pi@raspberrypi ~ $ cd mcpi/api/python/
pi@raspberrypi ~/mcpi/api/python $ python
Python 2.7.3 (default, Jan 13 2013, 11:20:46)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import mcpi.minecraft as mcpi
>>> mc = mcpi.Minecraft.create()
```

```
>>> mc.camera.setFollow()
>>> mc.camera.setNormal()
>>> mc.getPlayerEntityIds()
[1]
>>> mc.player.getPos()
Vec3(26.3,18.0,85.7152)
>>> mc.player.getTilePos()
Vec3(26,18,85)
>>> mc.player.setTilePos(26,38,85)
>>> mc.player.getTilePos()
Vec3(26,38,85)
>>> mc.player.getPos()
Vec3(26.5,38.0,85.5)
>>> mc.player.setTilePos(0,0,0)
>>> mc.getHeight(0, 0)
2
>>> mc.player.setTilePos(0,2,0)
>>> mc.setBlocks(-3, 2, -3, 3, 22, 3, 45)
```

Minecraft の画面が Python のメソッドを実行するたびに変化する。これだけでは分かりづらいので、操作した内容にコメントを追加しておく：

ディレクトリを移動して、python **を起動。**

```
pi@raspberrypi ~ $ cd mcpi/api/python/
pi@raspberrypi ~/mcpi/api/python $ python
Python 2.7.3 (default, Jan 13 2013, 11:20:46)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
```

mcpi **ディレクトリの** minecraft **モジュールをインポートして** mcpi **として参照。**

```
>>> import mcpi.minecraft as mcpi
```

Minecraft **クラスのオブジェクトを作成して変数** mc **に代入。これで** Minecraft **に接続。**

```
>>> mc = mcpi.Minecraft.create()
```

カメラを自キャラの上空から見下ろす視点に設定。

```
>>> mc.camera.setFollow()
```

カメラを自キャラの前方を見る通常モードに戻す。

```
>>> mc.camera.setNormal()
```

自キャラの ID を取得。

```
>>> mc.getPlayerEntityIds()
[1]
```

自キャラの位置を取得。

```
>>> mc.player.getPos()
Vec3(26.3,18.0,85.7152)
```

自キャラの位置を整数値で取得。

```
>>> mc.player.getTilePos()
Vec3(26,18,85)
```

自キャラを指定位置まで移動。

```
>>> mc.player.setTilePos(26,38,85)
```

自キャラの位置を整数値で取得。

```
>>> mc.player.getTilePos()
Vec3(26,38,85)
```

自キャラの位置を取得。

```
>>> mc.player.getPos()
Vec3(26.5,38.0,85.5)
```

自キャラの位置を原点まで移動。

```
>>> mc.player.setTilePos(0,0,0)
```

原点の（地表の）高さを取得。

```
>>> mc.getHeight(0, 0)
2
```

地表の高さに移動。

```
>>> mc.player.setTilePos(0,2,0)
```

原点付近に赤いレンガの塔を建てる。

```
>>> mc.setBlocks(-3, 2, -3, 3, 22, 3, 45)
```

その他、詳しい内容は下記のサイトを参照のこと。
http://www26.atwiki.jp/minecraft/pages/918.html

3.2.1 コマンド

現状使用できるコマンド一覧

カメラ

- `mc.camera.setNormal([entityId])}`
- `mc.camera.setFixed()`
- `mc.camera.setFollow([entityId])`
- `mc.camera.setPos(x, y, z)`

自キャラ

- `mc.player.getPos() => Vec3`
- `mc.player.setPos(xf, yf, zf)`
- `mc.player.getTilePos() => x, y, z`
- `mc.player.setTilePos(x, y, z)`

World

- `mc.getBlock(x,y,z) --> blockTypeId`
- `mc.getBlockWithData(*args):`
- `mc.getBlocks(*args):`
- `mc.setBlock(x, y, z, blockTypeId)`
- `mc.setBlock(x, y, z, blockTypeId, blockData)`
- `mc.setBlocks(x1, y1, z1, x2, y2, z2, blockTypeId)`
- `mc.setBlocks(x1, y1, z1, x2, y2, z2, blockTypeId, blockData)`
- `mc.getHeight(x, z) --> Integer`
- `mc.getPlayerEntityIds() -- >id:int`
- `mc.saveCheckpoint()`
- `mc.restoreCheckpoint()`
- `mc.postToChat(msg)`
- `mc.setting(setting, status)`

Events

- `mc.events.clearAll()`
- `mc.events.pollBlockHits()`

ブロックの種類

`setBlock`, `setBlocks` で指定できるブロックの種類の番号（`blockTypeId`）一覧（次ページ）

番号	ブロックの種類	番号	ブロックの種類
0	AIR	1	STONE
2	GRASS	3	DIRT
4	COBBLESTONE	5	WOOD PLANKS
6	SAPLING	7	BEDROCK
8	WATER FLOWING	9	WATER STATIONARY
10	LAVA FLOWING	11	LAVA STATIONARY
12	SAND	13	GRAVEL
14	GOLD ORE	15	IRON ORE
16	COAL ORE	17	WOOD
18	LEAVES	20	GLASS
21	LAPIS LAZULI ORE	22	LAPIS LAZULI BLOCK
24	SANDSTONE	26	BED
30	COBWEB	31	GRASS TALL
35	WOOL	37	FLOWER YELLOW
38	FLOWER CYAN	39	MUSHROOM BROWN
40	MUSHROOM RED	41	GOLD BLOCK
42	IRON BLOCK	43	STONE SLAB DOUBLE
44	STONE SLAB	45	BRICK BLOCK
46	TNT	47	BOOKSHELF
48	MOSS STONE	49	OBSIDIAN
50	TORCH	51	FIRE
53	STAIRS WOOD	54	CHEST
56	DIAMOND ORE	57	DIAMOND BLOCK
58	CRAFTING TABLE	60	FARMLAND
61	FURNACE INACTIVE	62	FURNACE ACTIVE
64	DOOR WOOD	65	LADDER
67	STAIRS COBBLESTONE	71	DOOR IRON
73	REDSTONE ORE	78	SNOW
79	ICE	80	SNOW BLOCK
81	CACTUS	82	CLAY
83	SUGAR CANE	85	FENCE
89	GLOWSTONE BLOCK	95	BEDROCK INVISIBLE
98	STONE BRICK	102	GLASS PANE
103	MELON	107	FENCE GATE
246	GLOWING OBSIDIAN	247	NETHER REACTOR CORE

3.3 Turtle グラフィックス

Minecraft Pi 用にタートルグラフィックスという亀（Turtle）を操作してグラフィックスを操作するプログラムも用意されている。

まずは、インターネット上から Minecraft Turtle のプログラムをダウンロードしておく：

```
Minecraft Turtle のプログラムをダウンロード。
pi@raspberrypi ~ $ cd ~
pi@raspberrypi ~ $ git clone https://github.com/martinohanlon/minecraft-turtle.git

Minecraft Turtle へカレントディレクトリを移動
pi@raspberrypi ~ $ cd minecraft-turtle

サンプルを実行
pi@raspberrypi ~/minecraft-turtle/ $ python example_squares.py
```

サンプルが起動できていることが確認できたら、Python の「対話モード」に入り（カレントディレクトリが minecraft-turtle にあることを確認)、実際に動かせることを確認：

```
Minecraft Turtle のモジュールをインポート
>>>import minecraftturtle
>>>import minecraft
>>>import block

Minecraft に接続。
>>>mc = minecraft.Minecraft.create()

自キャラの位置を取得。
>>>pos = mc.player.getPos()

自キャラの位置に Minecraft Turtle を作成。
>>>steve = minecraftturtle.MinecraftTurtle(mc, pos)

Turtle を前へ 15 ブロック移動。
>>>steve.forward(15)

Turtle を 90 度右に向かせる。
>>>steve.right(90)
```

```
Turtle を 45 度上に向かせる。
>>>steve.up(45)

Turtle を（今向いている方向の）前へ 25 ブロック移動。
>>>steve.forward(25)
```

このプログラムを実行後 Turtle が 2 本の線を描く。1 つはまっすぐな直線で、もう 1 つは右に 90 度、上に 45 度向いた線になる。他にも以下のような命令ができる:

```
Turtle を作成。位置はオプションで、指定しない場合は原点（0,0,0）に作られる。
>>>steve = minecraftturtle.MinecraftTurtle(mc, pos)

右や左に角度を指定して向きを変更。
>>>steve.right(90)
>>>steve.left(90)

上や下にも角度を指定して向きを変更。
>>>steve.up(45)
>>>steve.down(45)

前や後ろに指定したブロック数分移動。
>>>steve.forward(15)
>>>steve.backward(15)

Turtle の位置を取得。
>>>turtlePos = steve.position
>>>print turtlePos.x
>>>print turtlePos.y
>>>print turtlePos.z

Turtle の位置を設定。
>>>steve.setposition(0,0,0)
>>>steve.setx(0)
>>>steve.sety(0)
>>>steve.setz(0)

Turtle の頭の向きの変更。
>>>steve.setheading(90)
>>>steve.setverticalheading(90)
```

ペンの上げ下げ。

```
>>>steve.penup()
>>>steve.pendown()
```

ペンが下がっているか確認。

```
>>>print steve.isdown()
```

ペンとして使うブロックの変更。

```
>>>steve.penblock(block.DIRT.id)
```

Turtle のスピードの変更（１が最遅、１０が最速、０は動かない）

```
>>>steve.speed(10)
```

Turtle をスタート位置に戻す。

```
>>>steve.home()
```

この Turtle グラフィックスのプログラムは下記のサイトに詳細が載っている。
http://www.stuffaboutcode.com/2014/05/minecraft-graphics-turtle.html

4 参考文献

本講義の資料として、オライリー・ジャパンの『Python チュートリアル　第 2 版（Guido van Rossum 著、鴨澤 眞夫 訳)』の第 2 章から第 4 章の一部までを参考にした。

本書は Python の入門書として言語設計者が書いていることもあり、とてもおすすめできる本である（オライリー・ジャパンの書籍としては比較的安価でもある)。

この他にもオライリー・ジャパンより

- 『初めての Python』
- 『Python クックブック』

なども発売されている（ただし、これらは高い！)。